World Wide Web: Internet and Web Information Systems, 8, 233–259, 2005

DOI: 10.1007/s11280-005-1317-7

An XML-Based Approach to Publishing and Querying the History of Databases

FUSHENG WANG AND CARLO ZANIOLO
Department of Computer Science, University of California, Los Angeles, CA 90095, USA

Published online: 1 August 2005

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 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 difficult to express in SQL can be easily expressed in standard XQuery. This approach is very general, insofar as it can be used to store the version history of arbitrary documents and, for relational databases, it also supports queries on the evolution of their schema.

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.

Keywords: relational database history, temporal databases, data archiving, XML publishing, temporal queries, XQuery

1. Introduction

Information systems have yet to address satisfactorily the problem of how to preserve the evolving content and structure 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 as of today, commercial DBMSs do not provide effective means for archiving their data and supporting historical queries on their past contents. Given the strong application demand and the significant research efforts spent on these problems [25,30,43], the lack of viable 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 [37,38,50].

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 [49] and XSLT [44]. 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 [7,20].

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 [13], 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 investigate the two approaches and identify clear tradeoffs of query versus storage performance presented by the two approaches.

2. Related work

2.1. Time in XML

Some interesting research work has recently focused on the problem of representing historical information in XML. In [8] an annotation-based object model is proposed to manage historical semistructured data, and a special Chorel language is used to query changes. In [26] 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 XSLT. In [24], a dimension-based method is proposed to manage changes in XML documents, however how to support queries is not discussed.

In [1,2], 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 [17] is another extension of the 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.)

A τXQuery language is proposed in [22] to extend XQuery for temporal support, which has to provide new constructs for the language. An archiving technique for scientific

data was presented in [6], based on an extension of the SCCS [36] 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 [6]. The scheme we use here to publish the histories of relational tables present several similarities to that proposed in [6], but it also provides full support for XML query languages such as XPath and XQuery.

2.2. *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 [35]. Clifford et al. [13] classified them as two main categories: *temporally ungrouped* and *temporally grouped*. For temporal grouping, value equivalent attribute histories are grouped if the intervals are adjacent or overlap. The second representation is said to have more expressive power and to be more natural since it is history-oriented [13]. TSQL2 [50] tries to reconcile the two approaches [13] 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 [42], and a formal temporal object-oriented data model is proposed in [4]. The problem of version management in object-oriented and CAD databases has received a significant amount of attention [3,12]. However, support for temporal queries is not discussed, although query issues relating to time multigranularity were discussed in [27].

Recently Oracle implemented Flashback [34], an advanced recovery technology that allows users to roll back to old versions of tables in case of errors. Flashback only provides limited queries, and efficient support of temporal queries is not provided, where retrieval of historical data is through reading update logs.

2.3. *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 [16], 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 [19] and XPERANTO [7,20], 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 [19]; 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 commercial databases [15,33].

3. History of DB tables as XML documents

Tables 1 and 2 describe the history of employees and departments as they would be viewed in traditional transaction-time databases [35]. 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. Furthermore, throughout this section, we assume that relation keys remain invariant. For simplicity, we also assume names of employees and departments do not change. 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 2, we will use the representation shown in Figures 1 and 2. We will call these *H-documents*. 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. Note that there is a covering constraint that the interval of a parent node always covers that of its child nodes, which is preserved in the update process. 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 `tstart` set to *current timestamp* and `tend` set to *now*; and *update* can be implemented as a delete followed by an insert.

Table 1. The snapshot history of employees.

empno	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	Tech Leader	d02	1996-02-01	1996-12-31

Table 2. The snapshot history of departments.

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

```
<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`.

Our H-documents use a temporally grouped data model [13]. Clifford et al. [13] show that temporally-grouped models are more natural and powerful than temporally-ungrouped ones. Temporally grouped data models 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 several problems, including the coalescing problem [50]. TSQL2's approach [50] attempts to achieve a compromise between these two [13], and is based on an implicit temporal model, which is not without its own problems [9].

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. XML and XQuery support an adequate set of built-in temporal types, including datetime, date, time, and duration [49]; they also provide a complete set of comparison and casting functions for duration, date and time values, making snapshot and period-based queries convenient to express in XQuery. Furthermore, whenever more complex temporal functions are needed they can be defined using XQuery functions that provide a native extensibility mechanism for the language. User-defined functions are further discussed in Section 4.1.

We next show how to express temporal projection, snapshot, temporal slicing, history queries, and joins on `employees` and `departments`.

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 [40]. 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.

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

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

Query 1. Temporal projection: retrieve the salary history of "Bob":

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

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

```
for $d in doc("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 4.1.

Query 3. Temporal slicing: find employees history from 1995-05-01 to 1996-04-30:

```
for $e in doc("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 function is defined as follows:

```
define function overlapinterval($a, $b){
 if($a/@t end< $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))}
 }
}
```

The next query is a containment query:

Query 4. Containment: find employee(s) who worked in the "QA" department throughout the history of that department:

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

Query 5. Temporal Join: find the manager's empno for each employee:

```
for $e in doc("employees.xml")/employees/employee
for $d in doc("depts.xml")/depts/dept[deptno=$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. Temporal Join (restructuring): find the history of employees in each department:

```
element depts{
 for $d in doc("depts.xml")/depts/dept
 return
  element dept {$d/*, $d/*,
    element employees {
     for $e in doc("employees.xml")/
           employees/employee
     for $ed in $e/deptno
     where $ed = $d/deptno and
```

```
      not(empty(overlapinterval($ed, $d/deptno)))
      return <employee>
              {$e/empno, overlapinterval($ed, $d/deptno)}
             </employee>
    }
  }
}
```

This query will join departments and employees documents and generate a hierarchical XML document of employees' history in a department, grouped by department.

Alternative representations for table histories were studied and compared in [21]. 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 [21]. However IDs can be added to this representation to make some join queries easier [21]. Yet another approach consists of representing the joined tables by a hierarchically structured XML document. This approach simplifies some queries but complicates others [21]. 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 [21]. In summary, the representation used in this paper is the overall approach of choice, according to [21].

4. Complex temporal queries

In the previous sections we have discussed temporal queries such as snapshot queries, slicing queries, historical queries and temporal join queries. Next we show that more complex temporal queries can also be expressed with relative ease with the help of XQuery user-defined functions.

In the next two queries, we show that it is easy to support *since* and *until* connectives of first-order temporal logic [11].

Query 7. A Since B: find the employee who has been the manager of the dept since he/she joined the dept "d007":

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

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

Query 8. A Until B: find the employee who only worked in dept "d01" until he/she was appointed as the manager of dept "d07":

```
for $e in doc("employees.xml")/employees/employee
let $d1:=$e/deptno[1][.="d01"]
let $d2:=$e/deptno[.="d07" and tstart(.)=tend($d1)]
let $m:=$e/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/name}</employee>
```

Query 9. Continuous Periods: find employees who worked in dept (deptno= "d07") for more than 3660 consecutive days (approximately 10 years):

```
for $e in doc("employees.xml")/employees
    /employee[deptno="d07"]
for $d in $e/deptno[.="d07"]
let $duration := subtract-dates(tend($d),
    tstart($d))
where dayTimeDuration-greater-than($duration,"P3660D")
return
 <result>{$e/empno, $e/name}</result>
```

Here "P3660" is a duration constant of 3660 days in XQuery.

Query 10. Period Containment: find employees with same employment 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 doc("employees.xml")/employees
     /employee[empno = "10112"]
for $e2 in doc("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.

4.1. Built-in temporal functions

The user-defined functions `tstart($e)` and `tend($e)` 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:
`snapshot ($e, T)` will return the snapshot of node `e` at time `T`.

Restructuring functions:
`coalesce($l)` will coalesce a list 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; and `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;
`rtend($e)` recursively replaces all the occurrence of "now" with the value of CURRENT_DATE;
`externalnow($e)` recursively replaces all the occurrence of *now* with the string "now".

4.2. Support for now

This is an important topic that has received considerable attention in temporal databases [14]. 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 answer the query correctly.

Internally, we use the "end-of-time" value 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 (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()` 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 [14], 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 descendant nodes.

5. Temporal transformations and visualization with XSLT

The temporally grouped representation used here also provides a good platform for other XML-based technologies and tools. For instance, XSLT [44] is a widely used XML processing language for transforming XML documents into other XML documents or HTML documents. Indeed, H-documents can be easily queried and transformed using XSLT, enabling visualization on the Web of historical documents. A typical temporal transformation is snapshot retrieval, as shown in Figure 3. By providing a snapshot timestamp as the parameter, the XSLT transformation will return the snapshot at the timestamp. Other tem-

```
<xsl:stylesheet t0="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output method="xml" t0="1.0" encoding="UTF-8" indent="yes"/>
 <xsl:param name="t0" select="'1995-01-01'"/>
 <xsl:template match="*">
  <xsl:if test="@tstart&lt;=$t0">
   <xsl:if test="@tend&gt;=$t0">
    <xsl:element name="{name(.)}">
     <xsl:for-each select="@*[ (name(.)!='tstart') and  (name(.)!='tend' ) ] ">
      <xsl:attribute name="{name(.)}"><xsl:value-of select="."/></xsl:attribute>
     </xsl:for-each>
     <xsl:apply-templates/>
    </xsl:element>
   </xsl:if>
  </xsl:if>
 </xsl:template>
</xsl:stylesheet>
```

Figure 3. Snapshot query with XSLT.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
 <xsl:template match="employee">
  <xsl:if test="empno='400001'">
    <xsl:element name="salary_history">
     <xsl:for-each select="salary">
     <xsl:element name="{name(.)}">
      <xsl:for-each select="@*">
       <xsl:attribute name="{name(.)}"><xsl:value-of select="."/></xsl:attribute>
      </xsl:for-each>
      <xsl:apply-templates/>
     </xsl:element>
    </xsl:for-each>
    </xsl:element>
  </xsl:if>
  </xsl:template>
  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>
</xsl:stylesheet>
```

Figure 4. History query with XSLT.

poral transformations can also be done through XSLT by defining temporal templates, for instance, Figure 4. retrieves the salary history of employee "400001".

In addition, the H-document can be transformed with XSLT directly into HTML documents to be displayed on web browsers. A typical application is to visualize changes between the content at two timestamps, by querying the changes and marking background colors through XSLT. For instance, using the `span` tag of HTML, changed values can be highlighted with different background colors.

Figure 5 shows the XSLT template that, given two successive timestamps `t1` and `t2`, displays the employees at time `t2` as a table, where changes from time `t1` are color-marked. For new added tuples, we compare the starting timestamp of an `employee` with `t1`, and mark tuple values as yellow if it is new. Otherwise, if the value of a field does not change between `t1` and `t2`, no color is marked; if the value is updated, then it is marked as green.

In a summary, our H-document representation of data histories together with XSLT provides a convenient approach to transform and visualize the historical data on the Web.

6. 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 underlying relations. In the past, a variety of temporal data models

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
 <xsl:output method="html" version="1.0" encoding="UTF-8" indent="yes"/>
 <xsl:param name="t1" select="'19950201'"/>
 <xsl:param name="t2" select="'19960501'"/>
 <xsl:template match="/">
  <html><head><style type="text/css">
   .add { background-color: #FFFF99; }
   .del  { text-decoration: line-through;}
   .chg  { background-color: #99FF99; }
  </style></head>
  <body>
   <table border="1">
    <xsl:for-each select="/*/*[@tstart&lt;=$t2 and @tend&gt;=$t2]">
     <tr>
      <xsl:for-each select="*[@tstart&lt;=$t2 and @tend&gt;=$t2]">
       <td>
        <!-- new added tuple -->
        <xsl:if test="../@tstart&gt;$t1 and @tend&gt;=$t2">
          <span class="add"> <xsl:value-of select="."/> </span>
        </xsl:if>
        <!-- updated value -->
        <xsl:if test="../@tstart&lt;=$t1 and @tstart&gt;$t1
              and @tstart&gt;../@tstart and @tend&gt;=$t2">
          <span class="chg"> <xsl:value-of select="."/> </span>
        </xsl:if>
        <!-- unchanged value -->
        <xsl:if test="@tstart&lt;=$t1 and @tend&gt;$t2">
          <span class="nochg"> <xsl:value-of select="."/> </span>
        </xsl:if>
       </td>
      </xsl:for-each>
     </tr>
    </xsl:for-each>
   </table>
  </body></html>
 </xsl:template>
</xsl:stylesheet>
```

Figure 5. The XSLT document to visualize changes between two timestamps.

have been proposed [35], but few of them provide support for schema evolution or schema versioning [38]. Indeed, the flat structure of relational database makes it particularly difficult to support schema changes. In [45], 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 multiple 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, i.e., a relation begins when its first element is inserted or, alternatively, if content-free elements are 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. For example, say that the two tables `employees (empno, name,salary)` and `addresses(empno, address)` are joined into a new table `newemployees (empno, name, salary, address)`. Then, we have three H-documents: H-documents for `employees` and `addresses` respectively (which end at the joining time), and an H-document for `newemployees` (which starts from the joining time). Thus, to query the history we can (i) query each H-document individually and join the results, or (ii) merge the histories from the three H-documents, e.g., publish a new H-document `allemployees`, that includes all the histories before and after the join. Therefore schema changes involving the join or decomposition of relations can be represented in our H-documents.

6.1. 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. We can retrieve the schema history, detect schema changes, and view the schema snapshots at any particular time. Therefore, we can query changes of the database schema and its history, as demonstrated by the following queries:

Schema Q1. Schema snapshot: find all the columns of employees relation on 1995-01-01:

```
<columns>{
for $e in distinct-values(input()
  /employees/employee/&#x002A;[tstart(.)<="1995-01-01"
  and tend(.)="1995-01-01"]/local-name(.))
return <column>{$e}</column>
}</columns>
```

Here the XPath function `local-name($e)` will return the name of the element `e`, and `distinct-values($s)` will return distinct values of element names.

Schema Q2. Schema history of relations: find the history of the relation `employees`.

```
let $rel := input()/employees
return <tstart>{tstart($rel)}</tstart>,
       <tend>{tend($rel)}</tend>
```

Schema Q3. Schema history query: find the history of the attributes in the employees relation:

```
<columns>{
for $name in distinct-values(
     input()/employees/employee/*/local-name(.))
let $e := input("employees")/employees/employee
for $mergelist in mergeIntervals(
    $e/*[local-name(.)=$name])
return <column>{$name,tstart($mergelist),
    tend($mergelist)}</column>
}</columns>
```

where `mergeIntervals($list)` is a user-defined function that coalesces a list of history intervals and returns the merged lists. Note that the transaction-time history of a node has to be continous unless the schema changes. e.g., if the attribute salary is added in the table, then dropped, and later added back, then the `mergeIntervals` function will return a list of two intervals, with each interval representing an existed period of the attribute salary.

Schema Q4. Schema change detection: find when the address column was first added into the table:

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

Schema Q5. Joined relations: Find the salary history of employee "10000". Assume that the tables `employees(empno, name, salary)` and `addresses(empno, address)` are joined into a new table `newemployees (empno, name, salary, address)`.

There are two ways to specify the query, one is using a composed H-document that represents the whole history of employees, and the other is specifying queries on different H-documents and then merge the history in the query results (Indeed such schema evolution information can be captured and stored in some tables).

7. 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.

7.1. 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 from 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, Tamino XML Server [39] and X-Hive XML Database [47]. Both Tamino and X-Hive use text-based storage, and Tamino automatically compresses large XML documents for better storage efficiency.

7.2. RDBMS-based architecture

The H-documents are decomposed into a set of history tables. For each table in the current database, we decompose it as a *key table*, a set of *attribute history tables*, and a *global relational table*.

For example, we have the following relation in the current database:
`employee(empno, name, 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:

- The key table:
  `employee_id(id, tstart, tend)`
  is used to store the keys of employee objects, which is identical to `empno`, thus 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)`

where each table stores the history of an attribute. The values of ids in the above tables are the corresponding key values (e.g., empno), thus indexes on such ids 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

– *Global relational table*:
`relations(name, tstart, tend)`
records all the relations' history of the database schema, such as the timespan covered by tables `employees` and `departments`.

Our design builds on the *unique name* assumption of relational databases, where, e.g., queries will treat tuples denoted by the keys "Clark Kent" and "Superman" as two different entities, and the responsibility to make the connection between the two is left to the user. Here we group temporal information by the keys in the relations; thus the user must deal directly with any change in key values, e.g., by storing the change history of keys in the database.

7.3. Publishing table histories

When the historical data are decomposed and stored into tables, we use a middleware to publish data as XML. XTABLES (formerly XPERANTO) [7,20] 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 shown in Figure 6. Figure 7. shows the XQuery to reconstruct the H-document view of employees history, thus XML queries can be specified on such view through `XTABLES`.

7.4. 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

```
<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>
```

Figure 6. Default XML view of employee history tables.

```
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 7. User-defned XML view of employees that reconstructs the H-document view.

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 (user-defined functions are predefined and manually added into the SQL queries too).

7.5. Updating table histories

Changes in the current databases can be tracked with either update logs or triggers. For instance, in DB2 we built triggers (for update, deletion, and insertion) that successfully track and archive all changes in current tables updated into the historical tables. An alternative,

and possibly more efficient implementation, we are working on, uses database update logs for updating table histories.

8. Native XML DB versus RDBMS

8.1. Experimental setup

We investigate three systems for archiving: native XML database-based: Tamino and X-Hive; and RDBMS-based: DB2. To facilitate our testing, we generate a large amount of temporal data, representing as XML documents the history of 300024 employees over 17 years. The simulation models salary increases, changes of titles, and changes of departments. The size of the temporal data represented in XML documents (H-documents) is 335 MB.

The testing machine is a Pentium IV 3.0 GHz PC with Windows XP Professional, with 512 MB DDR memory and an 80 GB IDE hard drive. The databases we use are DB2 UDB V8.1 Enterprise Edition, Tamino V4.2, and X-Hive/DB V6.0. (All are the latest releases as of our testing in September, 2004.)

8.2. Storage utilization

One of the criteria for archiving systems is storage utilization. Figure 8 shows that X-Hive has the largest storage size among the three systems, since it uses text-based storage and no compression is provided. Tamino has the best storage efficiency, and the data size drops

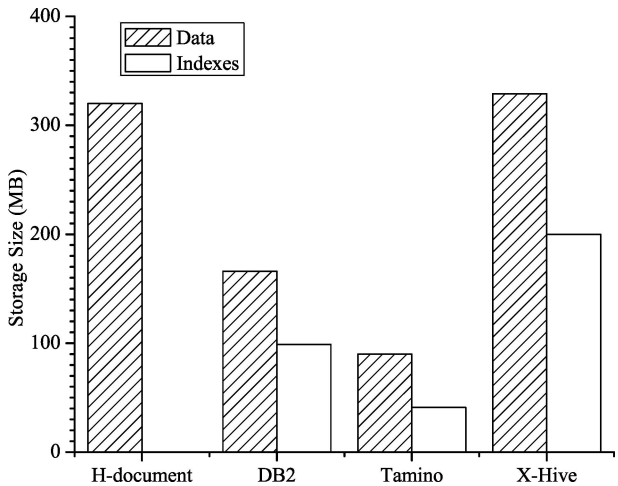

Figure 8. Comparison of database sizes for archiving temporal data.

to 27% 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. DB2 UDB reduces the size to 50% of original XML documents and lies in the middle. DB2 UDB does not provide compression options, while DB2 for OS/390 provides several data compression techniques [5] through either hardware or software, and the storage can be reduced to more than 50%. Oracle has recently announced a table compression feature, with typical compression ratios of 3:1 [33]. Thus, while compression techniques working directly on XML documents can achieve remarkable compression rates [31,39], 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 compressed ones [33].

In addition, a set of indexes are built for later query comparisons: indexes are created for all nodes/attributes which have predicates specified on.

8.3. Query performance

Query performance was tested for the queries shown in Table 3 chosen from the main classes of temporal queries as discussed in [41.50]. (To simplify the comparison, schema evolution queries are not included in our experiments).

Among these, Q1 and Q2 are history queries, Q4 and Q5 are snapshot queries, Q3 is a temporal slicing query, and Q6 is a temporal join. The queries are written in SQL for DB2, and XQuery for Tamino and X-Hive. We only consider the time spent in the database engine and ignore the time spent in returning the results. Each query was executed five times and the average was selected.

The query performance is compared in Figure 9. These experiments show that an RDBMS offers considerable performance advantages over native XML DBs. Typically, DB2 executes the queries one or two orders of magnitude faster than Tamino and X-Hive. Moreover, both X-Hive and Tamino run temporal join Q6 very slowly—indeed, we stopped the executions after 1000 seconds.

Table 3. Sample temporal queries

Q1.	History: find the total number of employees;
Q2.	History(single object): retrieve the history of employee "110000";
Q3.	Temporal slicing: Find all the salary changes throughout 1990-01-01 to 1991-01-01;
Q4.	Snapshot: find the average salary on 1995-01-01;
Q5.	Snapshot: find employees who joined the company on 1995-01-01;
Q6.	Temporal join: find employee pairs who have the same salary in the past year

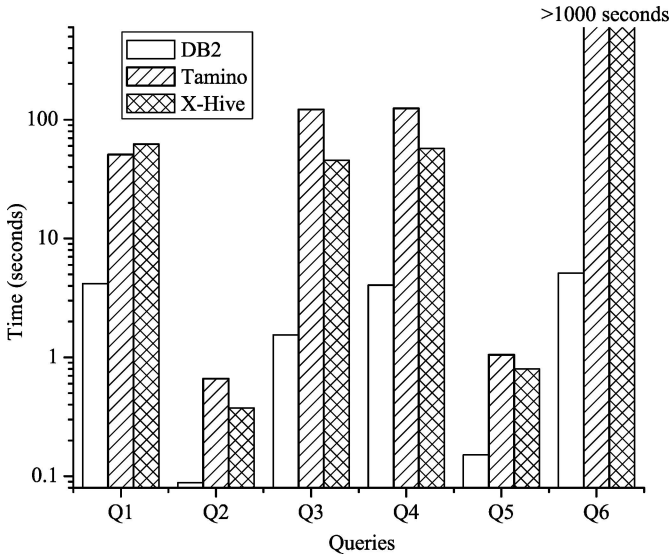

Figure 9. Query performance of DB2, Tamino and X-Hive.

Among the queries on DB2, Q1 is 12 times faster than Tamino, Q3 79 times faster, and Q4 31 times faster. Q2 and Q5 runs much faster on DB2 than on Tamino too. X-Hive and Tamino have comparable performance, and neither one handles well temporal join queries. These differences can be attributed, at least in part, to the fact that the technology of relational databases is more mature than that of XML databases. Furthermore, our experiments focused on preserving the history of relational data; different results could be expected when preserving the history of complex XML documents.

8.4. Temporal clustering

The performance on snapshot queries and temporal slicing queries, such as Q3 and Q4, can be further improved with more effective temporal clustering schemes.

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). The performance on snapshot queries and temporal slicing queries, such as Q3 and Q4, can be improved with more effective temporal clustering schemes; in such schemes, tuples having similar timestamp values are clustered together. We are currently experimenting with such a scheme and also studying various compression techniques that can be combined with it.

9. General XML documents

The temporal modeling and querying approach proposed in the previous sections is general and can be applied to arbitrary XML documents, i.e., ranging from strictly structured data-

```
<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>
 <employees tstart="1995-01-01" tend="1996-12-31">
  <employee tstart="1995-01-01" tend="1996-12-31">
   <empno tstart="1995-01-01" tend="1995-09-30">1001</empno>
   <name tstart="1995-01-01" tend="1995-09-30">Bob</name>
   <salary tstart="1995-01-01" tend="1995-05-31">60000</salary>
   <salary tstart="1995-06-01" tend="1995-09-30">70000</salary>
   <title tstart="1995-01-01" tend="1995-09-30">Engineer</title>
  </employee>
 </employees>
 </dept>
</depts>
```

Figure 10. The history of the `employee` table is published as `employees.xml`.

centric XML documents, to loosely structured text-centric Web documents. Two typical application areas are discussed next.

– *Temporal support of complex data-centric XML documents*
XML is becoming the most popular language for data exchange on the Web, and a large amount of XML standards have been created for data exchange, such as ebXML [18], and GML [23]. These documents often have complex schemas, such as deeply nested structures. As these XML documents are updated and evolved, their histories need to be archived and queried. This can be achieved by using the temporal grouping and query techniques previously described.

  Figure 10 shows a hierarchical representation where departments occupy the upper level, and employees occupy the lower level. All the elements of this temporal document are timestamped with transaction time intervals, where the usual covering relationship holds between the timestamp of an element and those of their children. XQuery can be used to query and update this document. (In fact, this temporal document could also be produced by joining the temporal representations of department histories in Figure 2. with the employee history in Figure 1).
– *Web archiving and version management*
As discussed in [46], an important issue of web archiving is how to preserve efficiently past versions of documents, and query their evolution history. Our temporal grouping scheme can be applied to documents of arbitrary hierarchical structure, and used to archive the version history of arbitrary XML documents [46]. Therefore, e-permanence of XML documents is achieved by representing their version history in XML and using

XQuery to express complex queries on the evolution of the documents and their contents. These important research topics are pursued by UCLA Web Information Systems Laboratory and the San Diego Super Computer Center through the ICAP project [28].

9.1. A user case: W3C standard specification documents

A first case-study we are working on is archiving as one timestamped XML document the successive versions of the UCLA course catalog which is published every year. Then we support queries such as: when was course CS143 introduced? and how many years did it take for nanotechnology topics to migrate from graduate-level courses to undergraduate ones?

A second case-study is the standard specifications published in XML by the W3 Consortium. Since the drafts are revised frequently, e.g., once within several months, we want to provide users with the ability of querying changes between versions. By representing a standard revision history with the approach discussed in previous sections, we can provide users the opportunity to query changes in any part of the specifications.

For instance, the history of the W3C XLink specs [48] is represented as described in (http://wis.cs.ucla.edu/icap/v-xlink.xml). We have automated the process of building the version history of a document (called V-document) from its successive versions by (i) first computing the diff between the previous version and the new version [32] and (ii) using the results of this process to update the V-document, by timestamping each its elements using attributes `vstart` and `vend`. The user can now query the document history as shown below.

Query V1. Snapshot: retrieve the "Introduction" section in the 2001-01-01 version of the XLink document:

```
for $sec in doc("xlinks.xml")/spec/body
 /div1[header="Introduction" and
 [vstart(.)<="2001-01-01" and vend(.)>= "2001-01-01"]
return snapshot($sec,"2001-01-01")
```

Here, `vstart($e)` and `vend($e)` retrieve the version starting and ending timestamps respectively, and `snapshot($node, $versionTS)` is a recursive XQuery function that checks the version interval of the element and only returns the element and its descendants where $vstart \leq versionTS \leq vend$, defined as follows:

```
define function snapshot($e, $v){
  if((date($e/@vstart)<=date($v))
      and (date($e/@vend)>=date($v)))
  then
   elementname($e)  }  {
    $e/text(), $e/@*[ string(name(.)) != "vstart"
       and string(name(.)) != "vend"],
    for $child in $e/*
```

```
    return snapshot($child, $v)
  }
  else ()
}
```

Query V2. Change Detection: check if there is any change on the subsection "Processing Dependencies" in the latest version.

This query needs to identify the beginning timestamp of the latest version, and this can be done in a number of ways. The simplest way consists in using the fact that the timestamp of each version is normally known, and store it in acsending order in another XML document `versions.xml`. (For the example at hand, the three XLink documents' release dates were 2000-07-03, 2000-12-20, and 2001-06-27 respectively). The document is in the format of:

```
<versions>
  <version snapshot="2000-07-03" />
  <version snapshot="2000-12-20" />
  <version snapshot="2001-06-27" />
</versions>
```

Then the beginning timestamp of last version can be retrieved using XPath function `last()`, as shown in the query.

An alternative way to implement our query is to identify a node that changes in every version, and then use the `last()` function in a similar fashion. (For example, in this V-document, "pubdate" changes with each versions.)

A third way to implement our query consists in comparing the `vstart` timestamps of the last versions of the document elements to ensure that they do not come after that of the "Processing Dependencies" element. Although this solution is less efficient than the previous ones, it will work in all cases, and it is simple to express using a function similar to `snapshot`.

```
let $pds := doc("xlinks.xml")/spec/body
  /div1/div2[header="Processing Dependencies"]
let $t := doc("versions.xml")/versions/version[last()]
return
      if (some $pd in $pds satisfies
          vstart($s) < $t and vend($s) > $t)
      then "true"
      else "false"
```

Query V3. History: retrieve the history of the abstract:

```
for $abs in doc("xlinks.xml")/spec/header/abstract
return $abs
```

Here too, we use XSLT to visualize with color highlighting the changes between any two versions—not just the two successive ones—in a way similar to that discussed for relational tables in Section 5.

10. Conclusion and future work

In this paper, we propose a temporally-grouped data model to represent and query the history of a relational database using XML and its languages, without requiring any change in existing standards. The evolution history of the underlying relational schema can also be represented and queried with our XML-based approach. The XML-viewed history of database tables can be stored using a native XML database or using an RDBMS. The technique is general and can be applied to arbitrary XML documents with complex structures, to support web archiving and version management.

Via the representation proposed in this paper, XML provides a more supportive environment to querying and manipulating temporal information than SQL: in particular XQuery proved effective at querying historical information and XSLT at transforming and visualizing the same. On the other hand, RDBMS proved to offer query performance advantages; however, native XML databases can be more effective in terms of storage because of compression techniques.

In general, while the approach proposed in this paper has proven to be quite effective at managing temporal information at a logical level, efficient support for temporal information and queries at the physical level remains an important topic for further research. For instance, we are currently investigating a segment-based archiving scheme based on page usefulness that assures temporal clustering and can boost the performance on many temporal queries significantly [10].

Acknowledgments

The authors would like to thank Shu-Yao Chien, Vassilis Tsotras, and Xin Zhou for many inspiring discussions; we are also grateful to the reviewers for many suggested improvements. This work was supported by a gift from NCR Teradata

References

[1] T. Amagasa, M. Yoshikawa, and S. Uemura, "A data model for temporal XML documents," in *DEXA*, 2000.
[2] T. Amagasa, M. Yoshikawa, and S. Uemura,"Realizing temporal XML repositories using temporal relational databases," in *CODAS*, 2001, pp. 63–68.
[3] D. Beech and B. Mahbod, "Generalized version control in an object-oriented database," in *ICDE*, 1988 pp. 14–22.
[4] E. Bertino, E. Ferrai, and G. Guerrini, "A formal temporal object-oriented data model," in *EDBT*, 1996.
[5] P Bruni and R Naidoo. *DB2 for OS/390 and Data, Compression*. IBM, Redbooks, 1998.
[6] P. Buneman, S. Khanna, K. Ajima, and W. Tan,"Archiving scientific data," in *ACM SIGMOD*, 2002.
[7] M. Carey, J. Kiernan, J. Shanmugasundaram et al.,"XPERANTO: A middleware for publishing object-relational data as XML documents," in *VLDB*, 2000.
[8] S.S. Chawathe, S. Abiteboul, and J. Widom, "Managing historical semistructured data," *Theory and Practice of Object, S,ystems* 24(4), 1999, 1–20.
[9] C. Chen and C. Zaniolo, "Universal temporal extensions for database languages," in *ICDE*, 1999.

[10] S.Y. Chien, V.J. Tsotras, and C. Zaniolo. Efficient schemes for managing multiversion XML documents. *VLDB, J,ournal*, 2003.
[11] J. Chomicki, D. Toman, and M.H. Böhlen,"Querying ATSQL databases with temporal logic," *TODS* 26(2), 2001, 145–178.
[12] H. Chou and W. Kim, "A unifying framework for version control in a CAD environment," in *VLDB*, 1986.
[13] J. Clifford, A. Croker, F. Grandi, and A. Tuzhilin," On temporal grouping," in *Recent, Advances in Temporal, Databases*, Springer, Verlag, 1995, pp. 194–213.
[14] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, and R.T. Snodgrass,"On the semantics of "now" in databases," *TODS* 22(2), 1997, 171–214.
[15] DB2 V8.1 Documentation. http://www.ibm.com/db2/
[16] DB2 XML, Extender. http://www-3.ibm.com/software/data/db2/extenders /xmlext/
[17] C.E. Dyreson,"Observing transaction-time semantics with TTXPath," in *WISE*, 2001.
[18] Electronic, Business using eXtensible, Markup Language(ebXML). http://www.ebxml.org
[19] M. Fernandez, W. Tan, and D. Suciu,"Silkroute: Trading between relations and XML," in *8th Intl. WWW, Conf.*, 1999.
[20] J.E. Funderburk, G. Kiernan, J. Shanmugasundaram, E. Shekita, and C. Wei, "XTABLES: Bridging relational technology and XML," *IBM Systems Journal* 41(4), 2002.
[21] F Wang and C Zaniolo,"Preserving and querying histories of XML-published relational databases," in *ECDM*, 2002.
[22] D. Gao and R. T. Snodgrass,"Temporal slicing in the evaluation of XML queries," in *VLDB*, 2003.
[23] Geography Markup Language (GML).
[24] M. Gergatsoulis and Y. Stavrakas,"Representing changes in XML documents using dimensions," in *Xsym*, 2003.
[25] F. Grandi, "An annotated bibliography on temporal and evolution aspects in the world wide web," in TimeCenter, Technique Report, 2003.
[26] F. Grandi and F. Mandreoli, "The valid web: An XML/XSL infrastructure for temporal management of web documents," in *ADVIS*, 2000.
[27] G. Guerrini M. Mesiti E. Camossi, and E. Bertino, "Automatic evolution of multigranular temporal objects," in *TIME*, 2002.
[28] ICAP: Incorporating, Change Management into Archival, Processes. http://wis.cs.ucla.edu/projects/icap/
[29] Information technology - database languages—SQL part 14: XML-related specifications.
[30] C. S. Jensen and C. E. Dyreson (eds), "A consensus glossary of temporal database concepts—february 1998 version," *Temporal Databases: Research and Practice*, 1998, pp. 367–405.
[31] H. Liefke and D. Suciu, "XMill: An efficient compressor for XML data," in *SIGMOD*, 2000.
[32] Microsoft XML. http:www.microsoft.com/xml/
[33] Oracle documentation, http://otn.oracle.com
[34] Oracle Flashback technology, http://otn.oracle.com
[35] G. Ozsoyoglu and R. T. Snodgrass,"Temporal and real-time databases: A survey," *IEEE, T,ransactions on Knowledge and Data, E,ngineering*, 7(4),1995, 513–532.
[36] M.J. Rochkind,"The source code control system," *IEEE Transactions on Software, Engineering*, SE-1(4),1975, 364–370
[37] J.F. Roddick, "A survey of schema versioning issues for database systems," *Information and Software Technology* 37(7), 1995, 383–393.
[38] J.F. Roddick, "A model for schema versioning in temporal database systems," in *Proc. 19th. ACSC Conf.*, 1996. pp. 446–452.
[39] H. Schning, "Tamino—a DBMS designed for XML," in *ICDE*, 2001.
[40] J. Shanmugasundaram et al., "Efficiently publishing relational data as XML documents," in *VLDB*, 2000.
[41] R. T. Snodgrass, *The TSQL2 Temporal, Q,uery Language*. Kluwer, 1995.
[42] R. Snodgrass, *Temporal, Object-Oriented, Databases: ACritical, Comparision*. Addions-Wesley/ACM, Press, 1995.
[43] R. T. Snodgrass. *Developing, Time-Oriented, Database Applications in SQL*. Morgan Kaufmann, 1999.

[44] The extensible stylesheet language (XSL). http://www.w3.org/Style/XSL/
[45] F. Wang and C. Zaniolo, "Representing and querying the evolution of databases and their schemas in XML," in *Intl. Workshop on Web, Engineering, SEKE*, 2003.
[46] F. Wang and C. Zaniolo, "Publishing and querying the histories of archived relational databases in XML," in *WISE*, 2003.
[47] X-Hive/DB, http://www.x-hive.com.
[48] XML, Linking Language (XLink) Version 1.0. http://www.w3.org/tr/xlink/
[49] XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/
[50] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V.S. Subrahmanian, and R. Zicari, *Advanced, Database Systems*, Morgan, Kaufmann Publishers, 1997.