Temporal XML? SQL Strikes Back!

Fusheng Wang
Siemens Corporate Research
755 College Road East, Princeton, NJ 08540
fusheng.wang@siemens.com

Carlo Zaniolo and Xin Zhou
Computer Science Department
University of California, Los Angeles
{zaniolo, xinzhou}@cs.ucla.edu

Abstract

While the introduction of temporal extensions into database standards has proven difficult to achieve, the newly introduced SQL:2003 and XML/XQuery standards have actually enhanced our ability to support temporal applications in commercial database systems. We illustrate this point by discussing three approaches that use temporally grouped representations. We first compare the approaches at the logical level using a common set of queries; then we turn to the physical level and discuss our ArchIS system that supports the three different approaches efficiently in one unified physical implementation. We conclude that the approaches of managing transaction-time information using XML and SQL can be integrated and supported efficiently within the current standards, and claim that the proposed approach can be extended to valid-time and bitemporal databases.

1. Introduction

In this paper, we seek to support historical information management and temporal queries without extending current standards. Our insistence on using only current standards is inspired by the lessons learned from the very history of temporal databases, where past proposals failed to gain much acceptance in the commercial arena, in spite of great depth, breadth [1, 2] and technical elegance [3, 4]. An in-depth review of the technical (and often non-technical) reasons that doomed temporal extensions proposed in the past would provide an opportunity for a very interesting and possibly emotional discussion; but such a discussion is outside the scope of this paper. Here, we simply accept the fact that temporal extensions to existing standards are very difficult to sell, in spite of the growing pull by temporal applications; then, we move on from there by exploring solutions that do not require extending current standards. This low-road approach is hardly as glamorous as the "new temporal standards" approach pursued in the past, but it is not without interesting research challenges and opportunities, as we will show in this paper. In particular, new opportunities are offered by two recent developments that have taken information systems well beyond SQL:1992 which, in the past, supplied the frame of reference for temporal database research. The first development is the introduction of XML/XQuery standards, that have gained wide acceptance by all DBMS vendors, and the second is the introduction of SQL:2003 [5, 6, 7], which contains new advanced features such as nested relations and OLAP functions.

The benefits of XML in temporal information management include: i) XML can be used to represent data in a temporally grouped data model, ii) XQuery provides an extensible and Turing-complete language [8], where new temporal functions can be defined in the language itself.

These features make it possible to use XML to represent the history of relational databases by timestamping the grouped attribute histories of each table, and XQuery to express complex temporal queries [9, 10, 11, 12]. This approach requires no extension to current standards, and it is very general, insofar as it can be used to represent and query the transaction-time, valid-time and bitemporal history of databases[11], and arbitrary XML documents [13]. Therefore, contrasting this experience with the past one focusing on SQL, we might simply conclude that XML and its query languages are more supportive to historical information management and temporal queries than SQL.

However, there are many reasons for which we are not prepared to give up on SQL. Indeed, relations represent a simple and intuitive data model which comes with (i) a built-in graphical representation in form of tables, (ii) WYSIWYG query languages such as QBE, and (iii) unique areas of commercial strength, such as OLAP applications and data warehousing. By contrast, (i) there is no built-in graphical rendering for an XML document, and this must be provided by the user via stylesheets, (ii) WYSIWYG XML query languages require further research and development, and (iii) XQuery is more complex than SQL, and its commercial application areas are still emerging.

There is also the critical issue of performance. In particular, in supporting transaction-time history of relational databases in XML [10, 12], we compared the two approaches of (i) implementing temporal queries directly in a native XML system, and (ii) recasting these views into historical tables, whereby the original XQuery statements are then mapped into equivalent SQL (or SQL/XML [14]) queries. Our experiments show that the second approach tends to be significantly more efficient [10, 12].

Therefore, both logical and physical considerations point to the conclusion that SQL is to remain the database language of choice, for a long time to come— particularly in data warehousing and business-intelligence applications— and every effort should be made to assure efficient management for historical information and temporal queries in SQL:2003. Toward this goal, we will take full advantage of the lessons learned in supporting temporal queries in XML and seek an efficient support for temporal queries independent of whether they are expressed in SQL or XQuery.

The paper is organized as follows. After the discussion of related work, in Section 3, we study the problem of representing relational database history in XML, along the lines proposed in [9, 12]. In the core of the paper, we seek to apply the lessons learned on XML to SQL:2003. Thus, in Section 4, we try to use the nested relations constructs provided by SQL:2003 to represent temporally grouped representations of database table histories. We use the same basic temporal queries to compare this approach to the XML-based approach and the SQL:2003-based approach of Section 5, where we support a temporally grouped representation using an OLAP-inspired view based on null values and flat tables. The representation used in Section 5 dovetails with data warehousing and business intelligence applications, and it is also capable of reconciling the event-based and state-based views of temporal databases. Finally, we consider efficient implementation issues and, in Section 6, we describe the architecture of our ArchIS system that unifies the support for both SQL-based and XML-based temporal views and queries at the physical level.

2. Related Work

Time in XML. Interesting research work has recently focused on the problem of representing historical information in XML. Approaches to support temporal XML documents by extending XML or its query languages have been proposed in [15, 16, 17, 18, 19].

For instance in [15], valid time on the Web is supported by introducing a new <**valid**> tag. The τXQuery language proposed in [19], extends XQuery with new constructs for temporal support.

An archiving technique for scientific data was presented in [20], but XML query language support is not provided.

Temporal Databases and Grouped Representations. There is a large number of temporal data models and query languages, including those discussed in [3, 21]; thus the design space for the relational data model has been exhaustively explored [1]. A useful taxonomy was introduced by Clifford et al. [22] who classified them into two main categories: *temporally ungrouped* and *temporally grouped* data models. Clifford had also suggested that the latter representation has more expressive power and is more natural since it is history-oriented [22, 23].

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

Table 1. The table employee_history

The basic representation used in ungrouped data models is tuple timestamping. As shown in Table 1, a new timestamped tuple is generated whenever there is a change in any of the attribute values. The well-know problem with this approach is that coalescing is needed when some of the attributes are projected out [24]. Much research has focused on this problem, and the solutions proposed include the TSQL2 [3] approach, and the point-based temporal model [25].

Versioning of DBMS was discussed in [26], and techniques for accurate time-stamping of transactions were discussed in [27] and Immortal DB [28]. Recently Oracle implemented Flashback [29], which supports the rollback to old versions of tables in case of errors. However, these systems do not provide much support for temporal queries.

SQL:2003. SQL:2003 [5], the latest release of SQL standards, is similar to SQL:1999, but provides significant extensions from SQL:1992. In particular, SQ:2003 O-R features include multiset, nested collection types (supported by both Oracle [7] and Informix [30]), and user-defined types. Another major feature is SQL/XML [14], which defines how SQL can be used together with XML in a database, and is supported by major database vendors. Publishing functions provided by SQL/XML can directly construct query results as XML documents or fragments.

3. Viewing Database History in XML

The use of XML to publish the history of database relations has been discussed in [10, 11, 31, 12], using a temporally grouped representation such as that of Figure 1 (which we call *H-document*) for the employee table as shown in Table 1. Powerful temporal queries on such representations can be expressed using XQuery. (In the remainder of this paper, our granularity for time is a day; however, all the techniques we present are equally valid for any granularity used by the application. For finer granularity, techniques in [27, 28] can be used. Furthermore, throughout this paper, we assume that relation keys remain invariant.)

3.1. Temporal Queries with XQuery

A full spectrum of queries was presented in [10] to illustrate the effectiveness of the approach—including temporal projection, temporal snapshot, temporal slicing, temporal join, and temporal aggregate. Because of space limitations, we restrict ourselves to the following three examples that will be used throughout the paper.

```
<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>
  <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. H-document: XML-based Representation of Employees' History

QUERY Q1. *Temporal Projection. Retrieve the title history of employee "1001":*

```
element title_history{
  for $t in doc("employees.xml")/employees/
        employee[empno="1001"]/title
  return $t }
```

Observe that no coalescing is needed after this projection, since the history of titles is temporally grouped.

QUERY Q2. *Temporal Snapshot. Retrieve titles and salaries of all employees on 1994-05-06:*

```
for $e in doc("employees.xml")/employees/
 employee
let $t:=$e/title[ tstart(.) <=
  date("1994-05-06") and
  tend(.) >= date("1994-05-06") ]
let $s:=$e/salary[ tstart(.) <=
  date("1994-05-06") and
  tend(.) >= date("1994-05-06") ]
return <employee>{$e/empno,$t,$s}</employee>
```

In this query, we need to check only the timestamps of the leaf nodes, since the H-document has a *temporal covering constraint*, i.e., the interval of a parent node always covers that of its child nodes.

Here, **`date()`** is a built-in function of XQuery (for simplicity, we omit the namespace **`fn`**). Instead, **`tstart()`** and **`tend()`** are user-defined functions to shield the user from the complexity of the underlying representation, since, e.g., 'now' [32] requires special representation and special handling (in ArchIS [10] we use the end-of-time to represent 'now'). **`date()`** is a built-in function of XQuery (for simplicity, the namespace **`fn`** is omitted here).

QUERY Q3. *Retrieve the salary history of employees in dept. "d01", while they were in that department.*

```
for $e in doc("employees.xml")/employees/
 employee[deptno="d01"]
for $s in $e/salary
for $d in $e/deptno[.="d01"]
let $ol:=overlapperiod($s, $d)
where not (empty($ol))
return
element salhistory{
  element empno {$e/empno/text()},
  element salary{$s/text()},
  $ol
}
```

Here **`overlapperiod($a, $b)`** is a user-defined function that returns an element **`PERIOD`** with overlapped period as attributes**`(tstart, tend)`**; if there is no overlap, then no element is returned which satisfies the XQuery built-in function **`empty()`**.

3.2. Discussion

The previous examples illustrate that XQuery is capable of expressing complex temporal queries, but the expression of these queries can be greatly simplified by a suitable library of temporal functions. The ArchIS system [10], discussed in Section 6, supports a rich set of functions, including the simple scalar functions described above, and also complex functions, including temporal aggregates and coalesce functions.

A significant benefit offered by the XML/XQuery-based approach to temporal information management is that it is very general and can handle the history of arbitrary XML documents that have evolved through successive versions [13]. The approach can also be extended to valid-time databases and bitemporal databases [11].

On the other hand, the ease of use of XQuery is questionable, and the problem of displaying the results of temporal queries in user-friendly ways can be a real challenge, since the tagged representations, such as that of Figure 1, are not suitable for casual users. To produce visually appealing representations, the query designer might have to code a stylesheet, using XSL [33]—possibly a different one for each query. This problem is far from trivial, and the visual rendering of temporal information poses interesting research challenges.

Finally, the growing popularity of XML in web-oriented applications does not change the fact that SQL remains the cornerstone of database applications, and its importance in areas such as business intelligence and data warehouses is growing every day. For these reasons, efficient support for temporal information and queries in SQL remains critical [**?**]. Therefore, we explore two approaches: one based on nested relations, which is discussed next, and another based on OLAP tables, which is discussed in Section 5.

4. DB History and Nested Relations

Nested relations are part of the latest SQL:2003 standards, and also supported by some commercial database vendors [7, 30]. Therefore a temporally grouped representation, similar to that used with XML, can also be achieved within SQL standard. For instance, for our **`employee_history`** example, we can use the following schema containing the nested table ('n-table' for short, or 'n-view' if it is a nested view) **`n_employee`**:

```
CREATE TYPE salary_typ AS OBJECT(
 salary    NUMBER(7),
 timep     PERIOD
);
...
CREATE TYPE salary_tbl AS TABLE OF
   salary_typ;
 ...
CREATE TABLE
n_employee(
 empno    VARCHAR2(8),
 timep    PERIOD,
 n_name   name_tbl,
 n_salary salary_tbl,
 n_title  title_tbl,
 n_deptno deptno_tbl)
NESTED TABLE n_salary STORE AS n_salary,
NESTED TABLE n_title  STORE AS n_title,
NESTED TABLE n_deptno STORE AS n_deptno;
```

This definition uses the user-defined type **PERIOD**, which can be defined in SQL:2003 as follows:

```
CREATE TYPE PERIOD AS OBJECT(
 tstart    DATE,
 tend      DATE
);
```

The same temporal queries that we have expressed on XML using XQuery can now be expressed on nested tables using SQL:2003, as follows:

QUERY Q1n. *History projection. Retrieve the title history of employee "1001"*:

```
SELECT  t.*
FROM    n_employee e, TABLE(e.n_title) AS t
WHERE   e.empno='1001'
```

QUERY Q2n. *Temporal Snapshot. Retrieve titles and salaries of all employees on 1994-05-06*:

```
SELECT t.title, s.salary
FROM  n_employee e, TABLE(e.n_title) AS t,
      TABLE(e.n_salary) AS s
WHERE tstart(t.timep) <= '1994-05-06'
 AND  tend(t.timep)   >= '1994-05-06'
 AND  tstart(s.timep) <= '1994-05-06'
 AND  tend(s.timep)   >= '1994-05-06'
```

Here too we use the functions **tstart()** and **tend()** to isolate the user for the internal representation of time, including 'now'. (Support for user-defined scalar functions is now available in all commercial OR-DBMSs.)

QUERY Q3n. *Retrieve the salary history of employees in dept."d01", while they were in that department.*

```
SELECT e.empno,
      overlapperiod(d.timep, s.timep),
      s.salary
FROM  n_employee AS e, TABLE(e.n_dept) AS d,
      TABLE (e.n_salary) AS s
WHERE d.deptno = 'd01' AND
      overlaps(d.timep, s.timep)
```

Here **overlaps()** is defined to return true if two periods overlap, and false otherwise; **overlapperiod()** is defined to return the overlapped **PERIOD**.

In addition to scalar functions, such as **overlapperiod()**, temporal aggregates (e.g., the temporal version of min and sum [3]) will be required by temporal queries. These new functions could be easily built into commercial systems by the vendors, or by the users, since commercial OR-DBMSs now support the introduction of new scalar and aggregate functions coded in a procedural language. (In the ATLaS system [34], user-defined aggregates can also be introduced natively in SQL, with no recourse to external PLs.) The new temporal aggregates that must be introduced include, the **rising** function of TSQL2 [3], and also the **tcoalesce** aggregate for temporal coalescing— since the temporally grouped representation made possible by nested tables has greatly reduced the need for coalescing, but not eliminated it all together (and the same is true for XML).

Assuming that a library containing the basic temporal functions is available, the complexity of writing temporal queries in SQL:2003 and nested tables is about the same as writing them in XQuery and XML. Both approaches present users with more alternatives in presenting data than flat relations. For instance, the join of nested employees and departments tables can be represented by a one-level hierarchy where the department and employee attributes are at the same level, or as a hierarchy where employees are grouped inside departments, or vice-versa. We next return to the 'Spartan simplicity' of flat relations, in which the alternatives are fewer and the problem is simplified.

5. An OLAP-Inspired Representation

A temporally grouped representations can also be obtained by using null values in flat tables such as those returned by OLAP aggregates. Thus, the transaction-time history of employees, that was described by tuple timestamping in Table 1, and as an XML document in Figure 1, is now described as a flat table with null values as shown in Figure 2, where the null value is represented by the question mark, "?". This representation can be defined by a ROLLUP operation on Table 1, defined by the following SQL statement (again here we use **timep** to represent the period of **tstart** and **tend**). As in the case of OLAPs, we might also want to represent the null values generated by the rullup operation differently from those representing null values in the original table.

```
CREATE VIEW e_employee AS
SELECT empno,
       tcoalesce(timep,salary,title,deptno)
FROM   employee_history
GROUP BY GROUPING SETS(empno,(empno,salary),
      (empno,title),(empno,deptno) )
```

We refer to the representation shown in Figure 2 as an 'e-table' (or 'e-view' if it is a view) because this captures the event-history for employees, as it will be discussed in Section 5.1. Moreover, temporal queries on e-tables preserve

empno	tstart	tend	salary	title	deptno
1001	1995-01-01	1995-05-31	60000	?	?
1001	1995-06-01	1996-12-31	70000	?	?
1001	1995-01-01	1995-09-30	?	Engineer	?
1001	1995-10-01	1996-01-31	?	Sr Engineer	?
1001	1996-02-01	1996-12-31	?	Tech Leader	?
1001	1995-01-01	1995-09-30	?	?	d01
1001	1995-10-01	1996-12-31	?	?	d02
1001	1995-01-01	1996-12-31	?	?	?

Figure 2. The history view e_employees

the traditional style of SQL queries:

QUERY Q1e. *History projection. Retrieve the title history of employee "1001":*

```
SELECT title, tstart, tend
FROM   e_employee
WHERE  empno= '1001'
AND  title IS NOT NULL
```

QUERY Q2e. *Temporal Snapshot. Retrieve titles and salaries of all employees on 1994-05-06:*

```
SELECT e.empno, e.title, e.salary
FROM   e_employee AS e
WHERE  tstart(e.timep) <= '1994-05-06'
 AND   tend(e.timep)   >= '1994-05-06'
 AND   e.title IS NOT NULL
 OR    e.salary IS NOT NULL
```

This query assumes that we only want to retrieve the information, without reformatting it. However, if we want to reformat the information derived into a join table, then we also want to join the titles and salaries of all employees at that date into a flat relation as follows:

```
SELECT s.empno, t.title, s.salary
FROM   e_employee AS s, e_employee AS t
WHERE  tstart(t.timep) <= '1994-05-06'
 AND   tend(t.timep)   >= '1994-05-06'
 AND   tstart(s.timep) <= '1994-05-06'
 AND   tend(s.timep)   >= '1994-05-06'
 AND   s.empno=t.empno
 AND   t.title IS NOT NULL
 AND   s.salary IS NOT NULL
```

QUERY Q3e. *Retrieve the salary history of employees in dept. "d01", while they were in that department:*

```
SELECT n1.empno, n1.salary,
       overlapperiod(n1.timep,n2.timep)
FROM   e_employee n1, e_employee n2
WHERE  n1.empno = n2.empno
  AND  n1.salary IS NOT NULL
  AND  n2.deptno IS NOT NULL
  AND  n2.deptno = "d01"
  AND  overlaps(n1.timep, n2.timep)
```

This query illustrates the use of temporal joins, with intersection of overlapping periods; these are required for query Q3 in all three representations. While the complexity of queries is similar for our three temporally-grouped approaches, e-tables offer unique advantages that are discussed next.

5.1. Event-Oriented Histories

An advantage of this last representation is that grouping can be easily controlled by the ORDER BY clause in SQL. For instance, the representation of Figure 2, where the history of each employee attribute is grouped together, is produced by the following clause:

```
SELECT empno, timep, salary, title, deptno
FROM   e_employee
ORDER BY empno, salary, title, deptno,
         tstart(timep)
```

Since the null value is assumed to be the last value in each domain, this ORDER BY clause indeed produces the table of Figure 2.

Assume now that we want to view the history of events, pertaining to employees' salaries and departments, that have occurred in the company; then we can just list them in ascending chronological order as follows:

```
SELECT timep, empno, salary,  deptno
FROM   e_employee
WHERE  title IS NULL
ORDER BY tstart(timep),empno,salary,deptno
```

However, in order to visualize the salary history of employees in a given department, we need first to write a query similar to that of Example Q3e to derive a table (or a view) **`depthist(deptno, empno, salary, timep)`**, on which we can write following query:

```
SELECT  deptno, timep, empno, salary
FROM    depthist
ORDER BY deptno,tstart(timep),empno,salary
```

This last statement returns all the events grouped by department and arranged in chronological order.

The visual presentation of historical data and query results is much simpler using e-tables than using n-tables or H-tables (which is discussed later in Section 6.1). This is because flat tables come with their built-in graphical representation, while, e.g., XML requires the user to write a style sheet to visualize data. Moreover, as demonstrated by the previous examples, restructuring on e-tables can be realized by simply reordering the tuple using an ORDER BY clause, whereas it might require complex nesting and unnesting in the other representations.

In most temporal database approaches, including TSQL2 [3], a temporal relation can be either declared as a state table or as an event table but the two views are not easily combined. A simple mapping between the two views is highly desirable since, in everyday life, states and events are two facets of the same evolving reality. Moreover, many advanced applications, such as time-series analysis [35], sequence queries [36], and data stream queries [37], view the database as a sequence of events, rather than a sequence of states.

The e-tables just described, make it possible the unification of state-based and event-based representations by simply using SQL ORDER BY construct. For instance, say that we want to find employees who have been transferred from a department to another, and from this, back to the old one. To answer this query by perusing the history of employees, we would probably start by carefully viewing the results of the following query:

QUERY Q4. *Reordering to detect round-trip transitions between departments:*

```
SELECT empno, timep, depno, salary, title
FROM   e_employee
ORDER BY empno, tstart(timep)
```

Then, the immediate sequence of any three tuples with non-null **deptno** column, would satisfy the query—provided that the first department is equal to the third (and that there was no interruption in the employee's employment).

Although this query is conceptually simple, it requires the detection of three successive tuples—an operation that is rather complex and inefficient to express in standard SQL. A first solution to this problem is to write a user-defined aggregate (UDA); in fact UDAs can easily express state-based computations [38]. Moreover, several event-patterns and sequence languages for time-series analysis have been proposed in the literature [39, 35, 36] and would work very nicely with the representation discussed here. For instance, using SQL-TS [36] our query could be expressed as follows:

QUERY Q5. *From department A to B and back, with no other change in between:*

```
SELECT A.empno, A.title
FROM   e_employee [ORDER BY empno,
       tstart(timep)] AS (A,B,C)
WHERE  A.deptno = C.deptno
  AND  B.deptno IS NOT NULL
```

Here, the FROM clause specifies that, given the ordering described above, **A**, **B** and **C** are three successive tuples that are also related by the conditions specified in the WHERE clause. Space limitation prevents us from delving into languages as SQL-TS [36], although they represent a very interesting and pertinent topic in temporal database research. Here, it suffices to observe that these languages rely on tuples being arranged in a suitable order—which is easier to achieve with e-tables than with H-tables or n-tables.

6. Efficient Implementation

In the previous sections, we have discussed the pros and cons of alternative representations for temporal history. In reality, these are likely to be supported together, rather than as alternatives, since database vendors are gung ho on supporting both SQL and XML in their systems. Practical considerations also suggest that a unified implementation at the internal level should be provided for these multiple external views. At UCLA, we have been developing the ArchIS system that unifies the support for multiple external temporal models into one architecture [10, 12].

The basic architecture of ArchIS [10] is shown in Figure 3. ArchIS is designed to preserve and archive the history of the database by preserving the evolution of its content, either by using active rules attached to the database or by periodically visiting their update logs. ArchIS then supports alternative logical views of the database history described in the previous sections, by mapping queries against these views into equivalent queries against the history database. In our previous work on the implementation of storing H-documents [10, 12], we have compared the use of a native XML DBMS such as the Tamino XML Server [40], against the approach of shredding these documents and storing them into RDBMSs. The second approach was found to offer substantial performance advantages and will be used here. (In our implementation, the 'current database' and the archived one are managed by the same system. But the results are easily generalized to the situations where these two are separate and even remote.)

In the next sections we first discuss the structure of the Key & Attribute History Tables, used at the internal level and then we describe the problem of mapping external queries into internal ones. We finally describe the temporal clustering and indexing techniques used in improving the performance of such queries.

The problem of supporting XML views through stored RDBMS tables is hardly new since it has recently provided a major focus for database research [41, 42, 43]. However, here we do not need to support all XML documents and queries, but only historical views of database tables and temporal queries on such tables; thus, specialized techniques can be used for more efficient storage, and optimized query mapping.

6.1. History Tables

The history of each relation is preserved by a set of tables: one table for each attribute, and an additional table for the primary key of the original relation. Each tuple in the tables is timestamped with the two attributes **tstart** and **tend**. For example, consider our evolving DB relation **employee(empno, salary, title, deptno)** with key **empno**. The history of **employee** is preserved by following tables in ArchIS:

The Key Table:

employee_empno(empno, tstart, tend)

Since **empno** will not change along the history, the period **(tstart,tend)** in the key table also represents the valid period of the employee. The use of keys is for easy joining of all attribute histories of an object such as an employee.

Attribute History Tables:

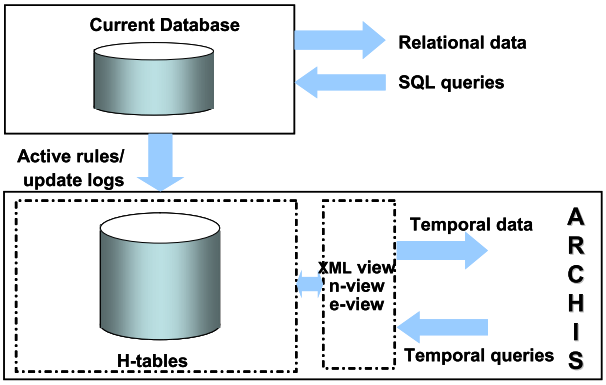

Figure 3. **ArchIS: Archival Information System**

```
employee_salary(empno, salary, tstart,tend)
employee_title (empno, title,  tstart,tend)
employee_deptno(empno, deptno, tstart,tend)
```

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

When a new tuple is inserted, the **tstart** for the new tuple is set to the current timestamp, and **tend** is set to *now*. When there is a delete on a current tuple, we simply change the **tend** value in that tuple as current timestamp. An update can be viewed as a delete followed by an insert. We will later refer to these as key & attribute history tables (H-tables for short). H-tables could also be viewed as yet another candidate representation at the logical level; we have not considered them here because they do not provide real query advantages with respect to e-tables, and they make tasks such as reordering and visualization harder.

In addition to these, we also store information about the schema in a global relation:

```
relations(relationname, tstart, tend)
```

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

6.2. Query Mapping

Mapping from e-views to H-tables. Mapping from H-tables to the e-views (or e-tables) of Figure 2 is simple, since the latter can be obtained taking the union of the H-tables after padding them with null values. This simple correspondence simplifies the translation and optimization of queries expressed on e-tables into equivalent queries on H-tables. The pattern of null values associated with the query plays an important role in the translation. Take for instance QUERY Q1e. There, the condition that **title IS NOT NULL** implicitly determines that salary and department must be null, and attribute table **employee_title** will appear in the WHERE condition of mapped query. Thus our original query is translated into:

```
SELECT T.title, T.tstart, T.tend
FROM   employee_title as T
WHERE  T.empno = '1001'
```

However, this is only the first step of the translation performed by ArchIS which also adds conditions to exploit the temporal clustering and indexing discussed later.

Mapping from n-views to H-tables. In DBMS that support nested relations, n-views (or n-tables) can be supported directly at the physical level. But even so, we might prefer to 'shred' and store them into flat H-tables, to simplify support for alternative external views (in particular, e-views), of for performance reasons, e.g., to take advantage of the clustering techniques available for H-tables, that will be discussed later. A simple approach to achieve this is to define a nested object-view (as defined in SQL:2003) on H-tables, as follows:

```
CREATE VIEW n_employee OF employee_t
WITH OBJECT IDENTIFIER (empno) AS
SELECT e.empno,
 PERIOD(e.tstart, e.tend) AS timep,
 CAST(MULTISET(
       SELECT s.salary, s.tstart, s.tend
       FROM employee_salary s
       WHERE s.empno = e.empno)
      AS salary_tbl)
     ) AS n_salary,
 ...
FROM employee_empno e;
```

With such a mapping, temporal queries on n-views are automatically translated by the DBMS into queries on H-tables through view definitions.

Mapping from XML-views to H-tables. The mapping from XML-views (or H-documents) to H-tables is significantly more complex. The problem of supporting XQuery on H-tables is similar in the sense that we have to generate efficient SQL queries, but more complex insofar as XML documents must be structured as output. Therefore, we use SQL/XML [14], whereby the results of SQL queries can be efficiently assembled into XML documents for output. Many database vendors now support efficient SQL/XML implementations, in which tag-binding and structure construction are done inside the relational engine for best performance [44]. In ArchIS [10], we compile XQuery statements on temporal XML-views, and optimize their translation into SQL/XML on the H-tables in five main steps, as follows:

1. *Identification of variable range*. For each distinct *tuple* variable in the original query, a distinct tuple variable is created in the FROM clause of the SQL/XML query, which refers to a certain key table or attribute table.
2. *Generation of join conditions*. There is a join condition `T.empno` and `N.empno` for any pair of distinct tuple variables.
3. *Generation of the WHERE conditions*. These are the conditions in WHERE clause of XQuery or specified in the XPath expressions.
4. *Translation of built-in functions*. The built-in functions (such as **overlaps($a,$b)**) are simply mapped into

the corresponding SQL built-ins we have implemented for ArchIS.

5. *Output generation*. This is achieved through the use of the XMLElement and XMLAgg constructs defined in SQL/XML [14].

For instance, the SQL/XML translation of Query Q1 is:

```
SELECT XMLElement (Name "title_history",
  XMLAgg (XMLElement (Name "title",
    XMLAttributes (T.tstart as "tstart",
      T.tend as "tend"), T.title)))
FROM employee_title as T
WHERE T.empno = '1001'
```

6.3. Clustering and Indexing

Efficient support for historical queries requires support for temporal clustering and indexing; in ArchIS, this is achieved by a simple usefulness-based scheme whereby the H-tables are partitioned into segments [10]. For each table, the *usefulness* of its current segment is defined as the percentage of the segment tuples that have not expired yet (i.e., whose **tend** timestamp is still 'now'). The usefulness of the current segment is monotonically decreasing with time, and as soon as it falls below a user-specified percentage, the whole segment is archived, and a new segment is started containing only those tuples whose timestamps are 'now'. The segment number then can become part of the search keys supported by the indexes used in the database.

Thus, a request to find the salary of a given employee at certain time, could involve finding the corresponding segment in a small memory-resident index, and then using the (**segment_no, empno**) pair in the index search.

This usefulness-based scheme achieves temporal clustering through redundancy. Since there is no update in the archived tuples of a transaction-time database (unlike valid-time databases), redundancy does not generate additional execution costs. For reasonable usefulness values the extra storage costs are modest (e.g., 30% storage overhead for 33% usefulness [10]); this cost represents a minor drawback, because of the fast decreasing cost of storage, and the applicability of compression techniques which has been proven in [10]. (The cost of re-compressing after updates is not present for archived data, since these are not updated.) On the other hand, the usefulness-based approach expedites archival search in a predictable and controllable fashion. For instance, for usefulness of 33% (1/3) we are assured that, when searching in the corresponding segments for records with a given timestamp, at least one of the three records visited has the right timestamp. Therefore, the time required to regenerate the past snapshot of a relation can be expected to be less than three times of that needed to generate the current snapshot from the current database [10]. Also, observe that the joining of H-tables require little extra time since they are already sorted on **empno**. The architecture and performance of ArchIS is covered in [10].

Scheme / Feature	Ungrouped	XML	Nested Relations	OLAP Tables
External Schema	Flat Tables	XML View	Nested Tables	Null-filled Flat Tables
Temporal Model	Ungrouped	Grouped	Grouped	Grouped
Query Language	SQL	XQuery	SQL:2003	SQL
Temporal Coalescing	Needed Very Often	Seldom Needed	Seldom Needed	Seldom Needed
Event Support	No	No	No	Yes
DBMS Support	ALL	Many	Some	ALL

Figure 4. Temporal Scheme Comparison

6.4. Summary

Only the skeleton of ArchIS is currently operational, and many improvements are planned for the future; even so, its realization confirms the practicality of supporting both SQL-based and XML-based temporal views and queries with a unified and efficient internal representation. ArchIS can now run on top of IBM DB2 and the ATLaS system [34]. We are currently working on extending it to run on commercial DBMS that support nested relations [7, 30], and explore any performance improvement that can be gained with this approach. We also plan to experiment with additional storage structures, such as R-trees, to better support valid-time and bitemporal databases.

7. Conclusion and Future Work

An important conclusion emerges from the research presented in this paper: a unified multi-model support for transaction-time databases can be achieved effectively using a temporally grouped data model. This requires the introduction of new temporal functions and aggregates, but no extension to the current standards. A unified efficient implementation for the three external models relies on well-understood query mapping/optimization techniques, and temporal clustering/indexing techniques at the internal level. In practice, the ArchIS approach is desirable since it provides a low-cost approach to address a wide range of applications. In particular, XML-based views dovetail with web applications, while nested-relations are more natural for object-oriented applications, and the null-filled flat tables are best for traditional database applications, decision-support applications, and event-oriented queries. This last approach provides a simple framework for the presentation of the data, which can require significantly more effort when XML is used. Figure 4 summarizes the features of the temporally-grouped schemes proposed, comparing them to the basic ungrouped scheme.

While we have concentrated here on transaction-time databases, it was recently shown that, for XML, this ap-

proach can be extended to bitemporal representations and queries as well [11]. Support for valid-time and bitemporal views and queries using nested relations and null-filled tables represents an important topic of forthcoming research. Many research issues also remain open at the physical level, including the use of nested relations and of clustering schemes that support updates on historical data (such updates are not present in transaction-time databases).

Acknowledgments

This work was partially supported by a gift of NCR Teradata. The authors would also like yo thank the referees for many useful suggestions.

References

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

[2] F. Grandi. An Annotated Bibliography on Temporal and Evolution Aspects in the World Wide Web. In *TimeCenter Technique Report*, 2003.

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

[4] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Transitioning Temporal Support in TSQL2 to SQL3. *Lecture Notes in Computer Science*, 1399:150–194, 1998.

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

[6] A. Eisenberg, J. Melton, K. Kulkarni, J. Michels, and F. Zemke. SQL:2003 has been published. *SIGMOD Rec.*, 33(1):119–126, 2004.

[7] SQL 2003 Standard Support in Oracle Database 10g, otn.oracle.com/products/database/ application_development/pdf/SQL_2003_TWP.pdf.

[8] S. Kepser. A Proof of the Turing-Completeness of XSLT and XQuery. In *Technical report SFB 441, Eberhard Karls Universitat Tubingen*, 2002.

[9] F. Wang and C. Zaniolo. An XML-Based Approach to Publishing and Querying the History of Databases. *To Appear in World Wide Web: Internet and Web Information Systems*.

[10] F. Wang, X. Zhou, and C. Zaniolo. Using XML to Build Efficient Transaction-Time Temporal Database Systems on Relational Databases. Technical Report 81, TimeCenter, Mar. 2005.

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

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

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

[14] ISO. Information technology - Database languages - SQL Part 14: XML-Related Specifications. 2003.

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

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

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

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

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

[20] P. Buneman, S. Khanna, K. Tajima, and W. Tan. Archiving scientific data. *TODS*, 29(1):2–42, 2004.

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

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

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

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

[25] D. Toman. Point-based Temporal Extensions of SQL. In *DOOD*, pages 103–121, 1997.

[26] M. Stonebraker. The Design of the POSTGRES Storage System. In *VLDB*, 1987.

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

[28] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal DB: Transaction Time Support for SQL Server. In *SIGMOD*, 2005.

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

[30] Informix Universal Server. http://www.ibm.com/informix.

[31] F. Wang, X. Zhou, and C. Zaniolo. Temporal Information Management using XML. In *ER*, 2004.

[32] 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.

[33] The Extensible Stylesheet Language (XSL). http://www.w3.org/Style/XSL/.

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

[35] Chang-Shing Perng and D. S. Parker. SQL/LPP: A Time Series Extension of SQL Based on Limited Patience Patterns. In *DEXA*, 1999.

[36] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and Optimizing Sequence Queries in Database Systems. *TODS*, 29(2):282–318, 2004.

[37] C. Zaniolo Y.-N. Law, H. Wang. Query Languages and Data Models for Database Sequences and Data Streams. In *VLDB*, pages 492–503, 2004.

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

[39] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A Model for Sequence Databases. In *ICDE*, pages 232–239, 1995.

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

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

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

[43] M. F. Fernandez, A. Morishima, D. Suciu, and W. C. Tan. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin*, 24(2):12–19, 2001.

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