

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

Fusheng Wang
Integrated Data Systems Department
Siemens Corporate Research
Princeton, NJ 08540, USA
fusheng.wang@siemens.com

Xin Zhou Carlo Zaniolo
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095, USA
{xinzhou,zaniolo}@cs.ucla.edu

Abstract

In this paper, we present the ArchIS system that achieves full-functionality transaction-time databases without requiring temporal extensions in XML or database standards. ArchIS' architecture uses (a) XML to support temporally grouped (virtual) representations of the database history, (b) XQuery to express powerful temporal queries on such views, (c) temporal clustering and indexing techniques for managing the actual historical data in a relational database, and (d) SQL/XML for executing the queries on the XML views as equivalent queries on the relational database. The performance studies presented in the paper show that ArchIS is quite effective at storing and retrieving under complex query conditions the transaction-time history of relational databases.

1. Introduction

The interest in and user demand for temporal databases have only increased with time [1, 2]; unfortunately, DBMS vendors and standard groups have not moved aggressively to extend their systems with support for transaction-time or valid-time. The lack of viable solutions suggests that (i) the technical challenges posed by the problem are many and severe, (ii) their severity is exacerbated by the inflexibility of the relational data model and the lack of extensibility of SQL, and (iii) the addition of temporal extensions to SQL standards is not to be expected any time soon. In this paper, we instead introduce a novel low-cost solution, by showing how XML and its query languages can be used to overcome most of these difficulties, and propose transaction-time extensions for database systems that require no modification of existing standards. Indeed, unlike the relational model, XML provides excellent support for temporally grouped data models, which have long been advocated as the most natural and effective representations of temporal information [3]. Moreover, unlike SQL, XQuery [4] is Turing-complete and natively extensible [5]. Thus many additional constructs needed for temporal queries can be defined in

XQuery itself, without having to depend on difficult-to-obtain extensions by standard committees. Therefore, while the challenge of expressing and supporting complex temporal queries should never be underestimated, in this paper we will show that the additional complexity of going from standard queries into temporal ones is much less when starting from XML/XQuery than when starting from relational tables and SQL.

This situation creates the unique opportunity of bringing much needed temporal database support to the users, since database vendors, while torpid on temporal extensions for RDBMS, are moving feverishly to add support for XML and XQuery to their systems. For instance, most database systems support the viewing of the database through XML views that can be queried using XQuery and other languages. These queries are then supported by mapping them into equivalent queries on the underlying database [6, 7]. Database vendors and standard groups are adding these capabilities to SQL through the SQL/XML initiative [8].

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

2. Viewing Relation History in XML

Table 1 describes the history of employees as they would be viewed in traditional transaction-time databases [1] using a temporally ungrouped representation. With this approach, any change of an attribute value will lead to a new history tuple, thus causes redundancy information between tuples. Moreover, temporal queries need to frequently coalesce tuples. Temporal coalescing is a source of complications in temporal databases, which is complex and hard to scale in RDBMS [9].

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

Table 1. The snapshot history of employees

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

Figure 1. Temporally grouped history of employees

These problems can be overcome using a representation where the timestamped history of each attribute is grouped under the attribute (Figure 1) [3], i.e., with the values of each attribute, we associate a set of time intervals denoting their validity—intervals that are adjacent or overlap for the same attribute value are coalesced. While this nested representation is hard to be represented in a flat table, it can be naturally represented by an XML-based hierarchical view shown in Figure 2. We call these *H-documents* or *H-views* when these are virtual representations. The root element in an H-document represents the corresponding table’s history (the creation and deletion of a table), and its child elements represent the grouped history of attribute values. Each element is assigned two attributes *tstart* and *tend*, to represent the inclusive time-interval of the element.

This approach can significantly reduce the need for coalescing, since an attribute history is already grouped. Another significant benefit is the effectiveness of expressing complex temporal queries with XQuery, as discussed next.

3. Temporal Queries using XQuery

The key advantage of our approach is that powerful temporal queries can be expressed in XQuery without requiring the introduction of new constructs in the language. We next show how to express the main classes of temporal queries on *employees.xml* document (Figure 2) and *depts.xml* document (which is the H-document of the department table, including the histories of attributes deptno, deptname, and mgrno).

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

```

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

```

```

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

```

Figure 2. The history of the employee table viewed in XML (employees.xml)

This query shows the benefit of removing coalescing in the query result.

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

```

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

```

tstart(\$e) and *tend*(\$e) are user-defined functions (expressed in XQuery) that get the starting date and ending date of an element respectively, thus the implementation is transparent to users.

QUERY 3: A Since B. Find the employee who has been a Senior Engineer in dept “d001” since he/she joined the dept:

```

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

```

Here *tcontains*(\$p, \$c) is a user-defined function to check if one interval contains another.

4. The ArchIS System

Two approaches are possible for storing and querying H-documents: one is to use a native XML DBMS such as Tamino XML Server [10]; the other is to use RDBMSs and provide mappings of queries and query results between the XML views and the underlying database systems.

We will next discuss the solutions of these problems used in our *Archival Information System—ArchIS* (*ArchIS-DB2* on DB2 and *ArchIS-ATLaS* on ATLaS [14]), which uses the RDBMS-based approach. The architecture of ArchIS is shown in Figure 3.

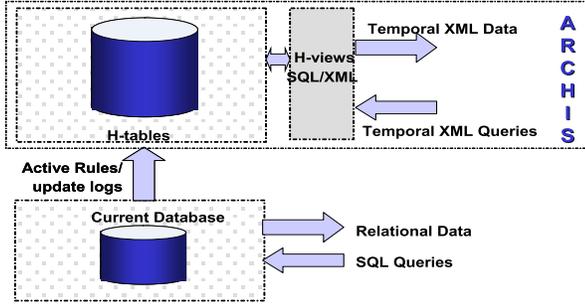


Figure 3. ArchIS: Archival Information System

4.1. H-tables

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

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

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

where *id* is the key. The history of the table is viewed as an H-document, which is then stored as (i) a key table, (ii) the attribute history tables, and a global relation table:

The Key Table:

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

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

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)
```

Global Relation Table:

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

will record all the relations history in the database schema.

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

4.2. Updating Table Histories

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

4.3. Query Mapping

Instead of using general query mapping approaches [11, 6], for ArchIS we developed specialized techniques and software, thus obtaining significant performance improvements. ArchIS implements XQuery on H-views, by translating them into equivalent SQL/XML expressions on H-tables. SQL/XML [8] is now a standard efficiently supported in commercial DBMS.

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

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

The mapping of queries on H-views to H-tables can be summarized as five main steps: i) identification of variable range; ii) generation of join conditions; iii) generation of the **where** conditions; iv) translation of built-in functions; and v) output generation. The implementation is based on Galax [12], an open source parser of XQuery. The translation algorithm is given in [13].

4.4. Usefulness-Based Clustering

We develop a segment-based archiving scheme which has better temporal clustering, and will boost the performance of most temporal queries.

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

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

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

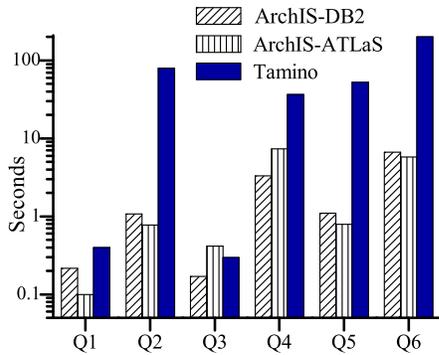


Figure 4. Query performance of segment-based archiving on RDBMS vs native XML DB

4. All live tuples in SEG_{live} are copied into a new live segment $SEG_{live'}$, and the old live segment is dropped.

After these operations are completed, segment $SEG_{live'}$ becomes the new live segment for updates, and the process repeats.

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

5. Performance Study

We investigate three systems for archiving: native XML database Tamino (Enterprise Edition V4.1); ArchIS-DB2 built on RDBMS DB2 (DB2 Enterprise Edition V7.2), and ArchIS-ATLaS built on ATLaS [14]. Tamino stores H-documents directly, and the latter two systems map and store H-views into H-tables clustered on segments (with U_{min} as 0.4 and 9 segments). The experiments are performed on a Pentium IV 2.4GHz PC with RedHat 8.0, with 256MB memory and an 80GB ATA hard drive. We prepare a set of typical temporal queries such as snapshot, temporal slicing, history, and temporal join.

Figure 4 shows the query performance on the three systems. The results suggest that RDBMSs offer substantial performance advantage over a native XML DB for most queries. The difference of snapshot queries between RDBMS and native XML databases is more significant. For instance, snapshot query Q2 on ArchIS-ATLaS is 102 times faster, and temporal slicing query Q5 is 66 times faster. History query Q4 on ArchIS-ATLaS is nearly 4 times faster, and temporal join Q6 is 35 times faster.

6. Conclusion

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

References

- [1] G. Ozsoyoglu and R.T. Snodgrass. Temporal and Real-Time Databases: A Survey. *TKDE*, 7(4):513–532, 1995.
- [2] R. T. Snodgrass. Developing Time-Oriented Database Applications in SQL. *Morgan Kaufmann*, 1999.
- [3] J. Clifford, A. Croker, F. Grandi, and A. Tuzhilin. On Temporal Grouping. In *Recent Advances in Temporal Databases*, pages 194–213. Springer Verlag, 1995.
- [4] XQuery 1.0: An XML Query Language. <http://www.w3.org/XML/Query>.
- [5] S. Kepser. A Simple Proof for the Turing-Completeness of XSLT and XQuery. In *Extreme Markup Languages*, 2004.
- [6] J.E. Funderburk, G. Kiernan, J. Shanmugasundaram, E. Shekita, and C. Wei. XTABLES: Bridging Relational Technology and XML. *IBM Systems Journal*, 41(4), 2002.
- [7] Oracle XML. <http://otn.oracle.com/xml/>.
- [8] SQL/XML. <http://www.sqlx.org>.
- [9] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. In *VLDB*, 1996.
- [10] H. Schöning. Tamino - a DBMS Designed for XML. In *ICDE*, 2001.
- [11] D. DeHaan, D. Toman, M. P. Consens, and M. T. Ozsu. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Encoding. In *SIGMOD*, 2003.
- [12] Galax—an Open Source XQuery Implementation. <http://www.galaxquery.org>.
- [13] F. Wang, X. Zhou, and C. Zaniolo. Using XML to Build Efficient Transaction-Time Temporal Database Systems on Relational Databases. Technical Report 81, TimeCenter, www.cs.auc.dk/TimeCenter, Mar. 2005.
- [14] ATLaS. <http://wis.cs.ucla.edu/atlas>.