REGULAR PAPER

ArchIS: an XML-based approach to transaction-time temporal database systems

Fusheng Wang · Carlo Zaniolo · Xin Zhou

Abstract Effective support for temporal applications by database systems represents an important technical objective that is difficult to achieve since it requires an integrated solution for several problems, including (i) expressive temporal representations and data models, (ii) powerful languages for temporal queries and snapshot queries, (iii) indexing, clustering and query optimization techniques for managing temporal information efficiently, and (iv) architectures that bring together the different pieces of enabling technology into a robust system. In this paper, we present the ArchIS system that achieves these objectives by supporting a temporally grouped data model on top of RDBMS. ArchIS' architecture uses (a) XML to support temporally grouped (virtual) representations of the database history, (b) XQuery to express powerful temporal queries on such views, (c) temporal clustering and indexing techniques for managing the actual historical data in a 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, and can also assure excellent storage efficiency by providing compression as an option. This approach achieves full-functionality transaction-time databases without requiring temporal extensions in XML or database standards, and provides critical support to emerging application areas such as RFID.

Keywords Temporal database · XML database · Temporal grouping · Temporal query · XQuery

1 Introduction

The interest in and user demand for temporal databases have only increased with time [1]. For example, the emerging RFID applications generate large amount of history data for tracking and monitoring physical objects [2]. Unfortunately, DBMS vendors and standard groups have not moved aggressively to extend their systems with support for transaction-time or valid-time. Given the strong application demand and the significant research efforts spent on these problems [3], 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 [4]. Moreover, unlike SQL, XQuery [5] is Turing-complete and natively

F. Wang (✉)
Integrated Data System Department, Siemens Corporate Research,
Princeton, NJ 08540, USA
e-mail: fusheng.wang@siemens.com

C. Zaniolo
Computer Science Department, University of California,
Los Angeles, Los Angeles, CA 90095, USA
e-mail: zaniolo@cs.ucla.edu

X. Zhou
Teradata Division, NCR Corporation,
Los Angeles, CA 90245, USA
e-mail: xin.zhou@ncr.com

extensible [6]. Thus many additional constructs needed for temporal queries can be defined in XQuery itself, without having to depend on difficult-to-obtain extensions by standard committees. However, in this paper we show that XML/XQuery can play a very useful role in bringing temporal support to database systems while retaining current standards, and such extensions are needed in emerging application areas, such as the management of RFID data [2]. Because of these new applications, and since database vendors who were torpid on temporal extensions for RDBMS are now moving feverishly to add support for XML and XQuery to their systems, we see a window of opportunity for the approach proposed in this paper. In fact, most database systems now support the publishing and viewing of database tables through XML views, which can be queried using XQuery and other languages. These queries are then supported by mapping them into equivalent queries on the underlying database [7,8]. Database vendors and standard groups are adding these capabilities to SQL through the SQL/XML initiative [9].

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

The paper is organized as follows: in Sect. 2 we show that XML provides a natural vehicle for implementing a temporally grouped data model for representing the evolution history of a relational database. In fact, in Sect. 3 we show that complex temporal and snapshot queries can be expressed on such views using XQuery. Next, in Sect. 3.4, we show that the approach can be used quite naturally to support RFID applications. In Sects. 4 and 5, we focus on the efficient implementation of such queries on RDBMS, where queries on the XML views are translated into SQL/XML queries on the relational tables, and various indexing/clustering techniques are used to make the execution of these queries efficient. Query performance study in Section 6 shows that ArchIS is quite effective, and in Sect. 7, we propose database compression as an option, and present a simple but effective technique for compressing archived databases. Related work is discussed in Sect. 8, followed by conclusions and future work.

2 Viewing relation history in XML

Table 1 describes the history of employees as they would be viewed in traditional transaction-time databases [3] using a temporally ungrouped representation, where **id** is the key

Table 1 The snapshot history of employees

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

Fig. 1 Temporally grouped history of employees

of the table and remains invariant in the history.[1] With this approach, any change of an attribute value will lead to a new history tuple, thus causes redundancy information between tuples, e.g., the name of Bob appeared the same but was stored in all the 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. For instance, a temporal coalescing query can take more than 20 lines of SQL with SQL92, and the best performance of coalescing on RDBMS is quadratic [11].

These problems can be overcome using a representation where the timestamped history of each attribute is grouped under the attribute (Fig. 1)[4], 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 Fig. 2. We call these *H-documents* or *H-views* when these are virtual representations. Here for simplicity, we only show examples for employee table, other tables like dept table can be modeled in similar way. The root element in an H-document represents the corresponding table's history (the creation and deletion of a table), and its child elements

---

[1] In the remainder of this paper, our granularity for time is a day; however, all the techniques we present are equally valid for any granularity used by the application. For finer granularity, techniques in [10] can be used. Furthermore, throughout this paper, we assume that relation keys remain invariant.

```
<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>
```

Fig. 2 The history of the `employee` table is viewed as `employees.xml`

represent the grouped history of attribute values. Each element in an H-document is assigned two attributes `tstart` and `tend`, to represent the inclusive time-interval of the element. The value of `tend` can be set to *now*, to denote the ever-increasing current time. (This will be further discussed in Sect. 3.3.) Note that there is a *temporal covering constraint* that the interval of a parent node (table history) always covers that of its child nodes (attribute histories).

In general, consider a relation **R** and let K denotes its primary key. Then the corresponding H-document contains an entity E for each maximal period of validity of a value of K. Thus in Fig. 2, **id** is our key and the period of validity of **id** = **1001** is T = (1995-01-01–1996-12-31). Thus, our H-document contains an entity timestamped with this validity period. Then the successive values of the attributes for tuples with **id** = **1001** are represented as subelements of E timestamped with their periods of maximal validity for the values of the corresponding attributes. The attribute in K always has one period of validity, while the others have one or more; in our example, Bob has only one period, salary and deptno have two and title has three. Thus, H-documents have a simple and well-defined schema which can be easily generated from that of the relation whose history they want to preserve.

Our H-documents use a temporally grouped data model [4]. Clifford et al. [4,12,13] show that temporally grouped models are more natural and powerful than temporally ungrouped ones, and can significantly reduce the need for coalescing, since an attribute history is already grouped. For instance, in Fig. 2 we can project out the salary entries without having to change the title ones, and vice versa. If we instead use a relational representation, such as that of Table 1, we will have to perform a coalescing operation after these projections. Using the relational model, an alternative solution consists in decomposing Table 1 into four tables sharing the **id** column. This solution will avoid coalescing, inasmuch as H-documents do, but requires frequent joins on such tables. Therefore, H-views combine the benefits of both

representations and support queries on the whole view and its projections without requiring neither joins nor coalescing operations. On the other hand, these benefits are restricted to non-key attributes: coalescing is still required when we project out attributes from the primary key. For instance, a query such as "For how long has department d01 been without a Sr. Engineer?", would require projecting out name, salary, title and id—and a coalescing operation will then be needed because of the elimination of the key attribute **id**.

However, even for such cases, other XML's advantages remain, particularly since we can now use XQuery—a query language that is significantly more powerful than SQL [6]—to express coalescing and other temporal queries. In fact, as discussed next, complex historical queries can be expressed in standard XQuery, thus avoiding the need for a special-purpose temporal query language.

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 as discussed in [14,15]: *temporal projection, temporal snapshot, temporal slicing, temporal join, temporal aggregate,* and *restructuring*, on `employees.xml` document(Fig. 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}
```

This query returns all `title` elements of employee "Bob", using a single XPath `/employees/employee/title`, with a predicate on the `name` element. This query illustrates the benefits of avoiding the need for coalescing the query results. Since the history of titles is grouped, the projected result is already coalesced—whereas coalescing on the results would have to be performed in a temporally ungrouped representation.

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
```

This query returns all managers in the history whose interval overlap with the timestamp `1994-05-06`. Note that

xs is the namespace of XML Schema (the declaration of namespaces is ignored here). tstart($e) and tend($e) are user-defined functions that respectively extract the starting date and ending date of an element. By using such functions, the implementation details of our timestamping schemes can be made transparent to users. This will be further discussed in Sect. 3.2.

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

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

This query checks if the interval of an employee element overlapped with a given interval. Here, toverlaps ($a, $b) is a function that returns true if the temporal intervals of elements a and b overlap, and false otherwise; telement($x, $y) constructs an element with x and y as its attributes.

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

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

This query joins the depts.xml and employees.xml documents and generates a hierarchical XML document grouped by dept and manager: for each mgrno in depts.xml, all employees in employees.xml are returned if they were in the same department as and during periods that overlapped with those of the manager. Here, overlap-interval($a, $b) is a user-defined function that returns an interval element describing the overlapping period. If this period is empty, then the XQuery built-in function empty($e) is satisfied and nothing is returned.

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

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

Here tavg($s) is a temporal aggregate function that can be efficiently computed in a single scan of the database.

Other temporal aggregates such as TSQL2 *rising* [15] and *moving window* aggregates, can also be supported through user-defined functions.

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

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

The user-defined function restructure takes two lists and returns all their overlapping intervals.

3.1 More complex queries

Here, we discuss more advanced temporal queries, such as *until*, *since*, and *contain*, which are often used as a test for the expressive power of temporal languages [16]. For instance, the following is a *since* query:

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

```
for $e in doc("employees.xml")/employees/employee
let $m:= $e/title[.="Sr Engineer" and
tend(.)=current-date()]
let $d:=$e/deptno[.="d001" and
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. The first let clause returns current "Sr Engineers", and the second let clause returns those currently in dept "d001", with the period containing the title period.

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

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

Here `tequals($d1,$d2)` is a user-defined function that checks whether two nodes have the same time periods (i.e., same `tstart` and `tend`). For this query, we check that for each department Bob worked, if there is a person who once worked in that department as well, and with exactly the same period, and vice versa.

3.2 Temporal functions

The use of functions such as `tequals($d1,$d2)`, `tstart($e)`, and `tend($e)` offers the advantage of divorcing temporal 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 interval functions as defined Allen's temporal operators; during/date/time functions that will extract time from the database by hiding the implementation details; functions to handle now, and restructuring function for handling coalescing, which is very common for temporal databases.

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

Interval functions: `tequals($a,$b)`,`tmeets($a,$b)`, `toverlaps($a,$b)`, `tprecedes($a,$b)` and `tcontains($a,$b)` return true or false according to the positions of the two intervals. The `overlapinterval ($a,$b)` returns the overlapping portion of the intervals, if this is not empty, with the following form: `<interval tstart= "d1" tend="d2"/>`.

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

All these temporal functions must also support the special 'now' timestamp, which is discussed in Sect. 3.3, below.

In particular, because of the internal representation we use for 'now', the function `tcontains($a,$b)` can simply be defined as follows:

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

3.3 Support for "now"

An important issue in temporal databases is how to handle *now* or *UC* (until changed) [17,18]. In a transaction-time database, *now* means that the values in the tuple are still current at the time the query is asked. In our strategy, we replace the symbol "now" with the value `current_timestamp` (or `current_date`, depending on the used time granularity). Such instantiation is performed conservatively only when needed.

Internally, we use the "end-of-time" value (e.g., "9999-12-31" for date) to denote the "now" symbol; this representation can assure that the current search techniques based on indexes and temporal ordering can be used without any change. The user does not access this value directly, he/she will access it through built-in functions `tstart($e)` and `tend($e)`. While the function `tstart($e)` returns the start of the interval, the `tend($e)` function returns its end, if this is different from "9999-12-31" and `current_date` otherwise. Also, many situations call for a different solution. For instance, the XML nodes returned by QUERY 1, use the "9999-12-31" internal representation for *now*, which makes it easier to use this as input in other temporal queries.

However, for data returned to the end-user, two different representations are preferable. One is to return the `current_date` by applying function `rtend($e)` that, recursively, replaces all the occurrence of "9999-12-31" with the value of `current_date`. The other is to return a special string, such as "now" or "until-changed" to be displayed on the end-user screen. As discussed in [17], this is often more intuitive and appealing for users, and is supported by our built-in function `externalnow($e)` that does that for the node e and its sub-nodes.

3.4 RFID applications

The need for supporting temporal queries is pervasive in many new and important application areas, and in the previous section, we have shown how the classical queries from the temporal database literature can be effectively expressed in ArchIS. In this section, we will focus on the emerging application area of RFID data management, since temporal queries play a key role in this important application domain [2].

RFID (radio frequency identification) is a key technology for automatic identification and data capture that uses radio frequency waves to transfer data between readers and EPC. [2] Using RFID, objects can be automatically identified, categorized, and tracked, without requiring a line of sight or contact

---

[2] (EPC—Electronic Product Code—is an identification scheme for universally identifying physical objects, defined by standard committees [19].)
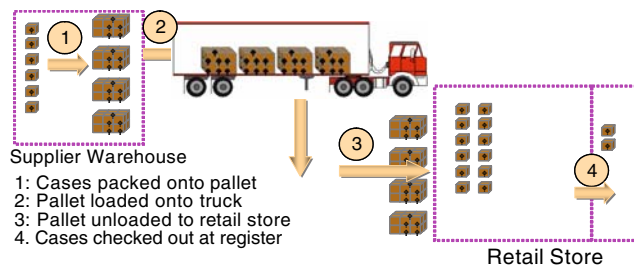
Fig. 3 Location driven changes in RFID-enabled applications

between readers and tagged objects. Because of its significant advantages, RFID technology is being gradually adopted and deployed in a wide area of applications.

There are several important entities in RFID applications, including *EPC-tagged objects*, *RFID readers*, and *locations*. Figure 3 demonstrates an example of an RFID enabled supply chain, where cases are packed onto pallets and shipped through trucks to retail stores to be sold to customers. During the process, these RFID-tagged entities interact with each other and experience various events, such as change of locations, and change of containers. There are also other types of events, such as business operations, and change of legal owners. Thus RFID data are temporally oriented [2]. One major goal of RFID applications is to track and monitor physical objects, thus a temporal database is essential to support such applications. Temporal support, however, is missing in past RFID middleware systems [20,21]. Next, we show that our approach is a perfect fit for tracking and monitoring RFID objects.

3.4.1 Representing RFID data in XML

The history of RFID objects can be directly represented in an H-view, as shown in an example in Fig. 4. In this example, the H-view represents the history of an object's along the

```
<objects tstart="2005-08-01" tend="now">
  <object tstart="2005-08-01" tend="now">
    <epc tstart="2005-08-01" tend="now">A1B2C3</epc>
    <location tstart="2005-08-01" tend="2005-08-05">warehouse1</location>
    <location tstart="2005-08-06" tend="2005-08-09">routeA</location>
    <location tstart="2005-08-10" tend="2005-08-15">retailerW</location>
    <location tstart="2005-08-16" tend="2005-08-17">sold</location>
    <location tstart="2005-08-18" tend="2005-08-19">retailerW</location>
    <location tstart="2005-08-20" tend="UC">sold</location>
    <containerepc tstart="2005-08-01" tend="2005-08-05">A1EFGD</containerepc>
    <containerepc tstart="2005-08-06" tend="2005-08-09">A1C3DF</containerepc>
    <owner tstart="2005-08-01" tend="2005-08-04">Supply A Co.</owner>
    <owner tstart="2005-08-05" tend="2005-08-15">Retail W Co.</owner>
    <owner tstart="2005-08-16" tend="2005-08-17">Customer</owner>
    <owner tstart="2005-08-18" tend="2005-08-19">Retail W Co.</owner>
    <owner tstart="2005-08-20" tend="UC">Customer</owner>
  </object>
  <!-- more ... -->
</objects>
```

Fig. 4 RFID object history in XML (rfid.xml)

three dimensions of location, containment, and ownership. Thus, the `<location>` history of our object identified by the EPC tag A1B2C3 is generated by the steps 1–4 of Fig. 3; its `<containerpc>` history instead records the placement first into a case and then into a pallet that also occurred during the steps shown in Fig. 3.

Next, we show that powerful RFID queries for tracking and monitoring can be supported in ArchIS with XQuery.

3.4.2 Temporal RFID queries with ArchIS

QUERY R1: Track the location history of object with EPC "A1B2C3":

```
element locationhistory{
for $l in doc("rfid.xml")/objects/
object[epc="A1B2C3"]/location
return $l }
```

QUERY R2: This case (with EPC "A1B2C3") of product is damaged. What other cases have ever been put in the same pallet with this case at the same time?

```
for $s in doc("rfid.xml")/objects/object
[epc="A1B2C3"]
for $c in $s/containerepc
for $o in doc("rfid.xml")/objects/object
[epc != "A1B2C3" and containerepc=$c]
where toverlaps($c, overlapinterval($s, $o))
return $o
```

This query checks if there are other objects in the same container of object "A1B2C3" and with overlapping periods. *QUERY R3: Find how many items were returned by customers (with location "sold") to the store (with location "retailerW") on 2005-11-11:*

```
let $o := doc("rfid.xml")/objects/object
[location ="retailerW" and tstart(location)
=xs:date("2005-11-11")]
where  every $i in $o satisfies
fn:exists($o[location = "sold" and
tend(location)=xs:date("2005-11-11")])
return count($o)
```

Since items sold to customers would begin a new location, the query checks all objects whose locations started at location "retailerW" on 2005-11-11. *QUERY R4: find missing objects at current location "L", which were available at location "S" at time "T".*

```
for $o in doc("rfid.xml")/objects/object
[location="S" and tcontains(location,
telement(xs:date("T"), xs:date("T")) )]
where empty($o/location[. ="L" and
tend($o/location) = "UC"]
return $o
```

This query will search objects which passed through location "S" at time "T", but are not available on "L" at present.

The XML-based representation and query support also facilitate the exchange of RFID data in distributed RFID

applications, such as large distributed supply chain systems. RFID observations are frequently exchanged among multiple sites with PML (Physical Markup Language) [22]; therefore, the RFID data history can be easily queried and exchanged in our H-document format.

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 [23]; the other is to use RDBMSs and provide mappings of queries and query results between the XML views and the underlying database systems. The query performance and the storage efficiency of the two approaches are compared in Sect. 6.

The main design issues that must be addressed for an efficient realization of the second approach include i) how to map (shred) the XML views representing the H-documents into tables (which we call *H-tables*), ii) how to translate queries on the XML views to the H-tables, and iii) which indexing, clustering and query mapping techniques should be used for high performance.

We will next discuss the solutions of these problems used in our Archival Information System—ArchIS, which uses the RDBMS-based approach (*ArchIS-DB2* on DB2 and *ArchIS–ATLaS* on ATLaS [24]—a compact implementation of an RDBMS at UCLA. The architecture of ArchIS is shown in Fig. 5. In our implementation, the "current database" and H-tables are implemented as tables in a same database, but the results are easily generalized to the situations where these two are separate, or even the case where the current database is a view containing the *now* snapshot of the H-tables.

4.1 H-tables

In ArchIS, 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
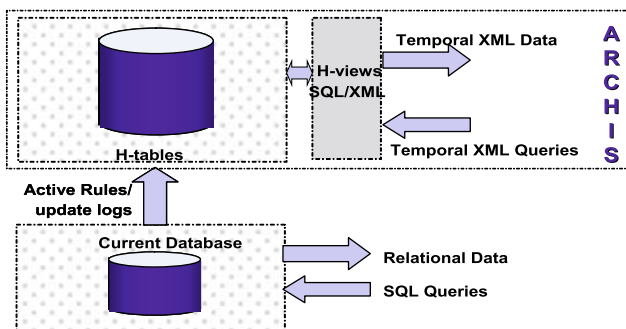

Fig. 5 ArchIS: Archival Information System

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.

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 easy joining of all attribute histories of an object such as an employee.

Attribute history tables:

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

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

A sample content of the `employee_salary` table is

ID	SALARY	TSTART	TEND
100022	40000	02/20/1988	02/19/1989
100022	42010	02/20/1989	02/04/1990
100022	42525	02/20/1990	02/04/1991
100022	42327	02/20/1991	02/19/1992
...			
100023	43162	07/13/1988	07/13/1989
...			

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

Global relation table:
`relations(relationname, tstart, tend)`
will record all the relations history in the database schema, i.e., the time spans covered by the various tables in the database. This corresponds to the root elements of H-documents.

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

Algorithm 1 XQuery to SQL/XML translation algorithm

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

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

Very general translation mechanisms from XML documents to RDBMS have been studied in [25] and middleware such as XTABLES [7] can be used to publish our H-tables into XML and support queries on such tables. After investigating these general-purpose translation techniques and software, for ArchIS we developed specialized techniques and software, thus obtaining significant performance improvements (see Sect. 6.2).

Therefore, ArchIS implements XQuery on H-views, by translating them into equivalent SQL/XML expressions on H-tables. SQL/XML [9] is now a standard efficiently supported in commercial DBMS. We also implemented it in our experimental system ATLaS using the techniques described in [26].

The expressions on H-tables use the SQL/XML constructs `XMLElement`, `XMLAttributes`, and `XMLAgg`, which are discussed next. The `XMLElement` and `XMLAttributes` constructs are used to return elements and their attributes. `XMLAgg` is an aggregate function, which constructs an XML value from a collection of XML value expressions. For instance, to return 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"
```

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

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

These SQL/XML constructs simplify the translation from queries expressed on H-views to equivalent queries on H-tables. For instance, the SQL/XML translation of QUERY 1 in Sect. 3 is shown below:

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

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

As another example, QUERY 3 will be translated to:

```
select XMLElement (Name "emp",
XMLElement (Name "id", XMLAttributes (
e.tstart as "tstart",e.tend as "tend"),e.id),
XMLElement (Name "name", XMLAttributes(
n.tstart as "tstart",n.tend as "tend"),n.name))
from employee_id e, employee_name n
where e.id = n.id  and toverlaps( e.tstart,
e.tend, "1994-05-06", "1995-05-06")
```

Here a join condition is needed to join `e.id` with `n.id`, which is implied in the XPath expression $\$e/name$. The translation of UDF (user-defined function) `toverlaps` takes in the `tstart` and `tend` values, and returns true or false. More on built-in function translation is discussed in Sect. 4.4.

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

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

attribute in such tables. For each distinct tuple variable in the original query, a distinct tuple variable is created in the `from` clause of the SQL/XML query.

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

– Generation of join conditions: There is a join condition `T.id` and `N.id` for any pair of distinct tuple variables.
– Generation of the `where` conditions: these are the conditions that are contained in the `where` clause of the XQuery or specified in the path expression (e.g., `[name="Bob"]` in QUERY 1).
– Translation of built-in functions: Temporal functions (such as `toverlaps($a,$b)`) are simply mapped into the corresponding built-ins we have implemented for ArchIS. We will have more discussion for function mapping in the next section.
– Output generation: This is achieved through the use of the `XMLElement` and the `XMLAgg` constructs previously described. Through expression of these constructs the ArchIS compiler supports simple expressions, such as `return $t` of QUERY 1, and more complex expressions such as QUERY 4. Meanwhile, users have the opiton to specify a `table` construct in the return clause to bypass the SQL/XML transformation, so the results can be returned as tables.

Algorithm 4.3 summarizes the overall translation algorithm, which is implemented on Galax [27], an open source parser of XQuery. The SQL generation at step 19 of Algorithm 4.3 produces a statement which lists in the FROM clause all the T_i tables, and lists in the SELECT clause all the entities E_i and subentities E_{ij} generated in the algorithms, with the WHERE clauses including the concatenation of all conditions from both C_{ij-id} and $C_{ij-where}$ conditions.

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

4.4 Function mapping

The temporal functions discussed in Sect. 3.2 were implemented as C++ user-defined functions imported into the SQL DBMS trough their standard UDF mechanism. These functions are collected in a special library registered with the ArchIS compiler which, therefore, embeds calls to these functions in the equivalent SQL/XML code generated by translating the temporal XQuery statements. This simplifies the mapping from XQuery to SQL. However, the mapping must

be modified according to the different situations of node types in H-views, discussed next.

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

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

Extending ArchIS via a library of user-defined function imported into the DSMS is similar to the approach widely used in commercial DBMS—under a babel of names such as DB extenders, snapins, cartridge, etc. The popularity of such libraries is due to the fact that they enable software houses and domain specialists to extend the DBMS without requiring changes to the standards. ArchIS uses a similar approach to bring about transaction-time databases.[3]

5 Temporal clustering and indexing

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

5.1 Usefulness-based clustering

Assume that an attribute history is stored in a segment. For each segment, we define its usefulness as $U = N_{\text{live}}/N_{\text{all}}$, 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:

---

[3] In our Tamino-based implementation we instead used temporal functions written for and imported into Tamino. To limit the impact of the different implementations, only one of the queries of our experiments uses a function.

Fig. 6 Segment-based archiving of history data
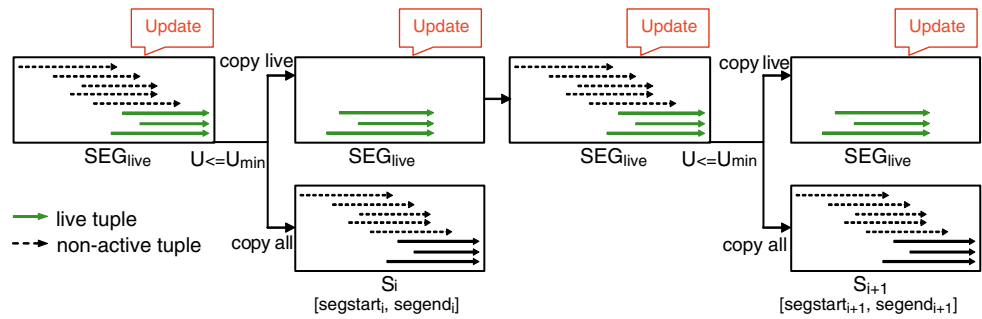
1. A new segment S_i is allocated;
2. The interval of this segment is recorded in the table segment(segno, segstart, segend), where segstart and segend record the starting and ending time for the segment respectively;
3. All tuples in SEG_{live} are copied into the new segment S_i, sorted by ID;
4. All live tuples in SEG_{live} are copied into a new live segment $SEG_{\text{live}'}$, 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. The process is illustrated in Fig. 6.

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

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

Indexing Since the table is generated by bulk-copying sorted by SegNo and ID, the physical clustering of the data is guaranteed. By building traditional B+-Tree indexes on SegNo and ID, we obtain a clustered B+-Tree index on the segmented table. In this way, for history queries, only segments involved in the queries need to be retrieved.

And the content in segment table will be

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

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

$$tstart_{\text{tuple}} \geq segstart_{\text{SEG}} \qquad (1)$$

$$tend_{\text{tuple}} \leq segend_{\text{SEG}} \qquad (2)$$

There are several advantages for segment-based clustering: First, the current live segment always has a high usefulness, which assures effective updates; second, records are globally temporally clustered on segments; third, for snapshot queries, only one segment is used, and for temporal slicing queries, only segments involved are used, thus such queries can be more efficient, as discussed in Sect. 6.3; and last, we have the flexibility to control the number of redundant tuples in segments by U_{min}, as discussed next.

5.2 Storage usage

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

$$\frac{N_{\text{seg}}}{N_{\text{noseg}}} \leq \frac{1}{1 - U_{\text{min}}} \qquad (3)$$

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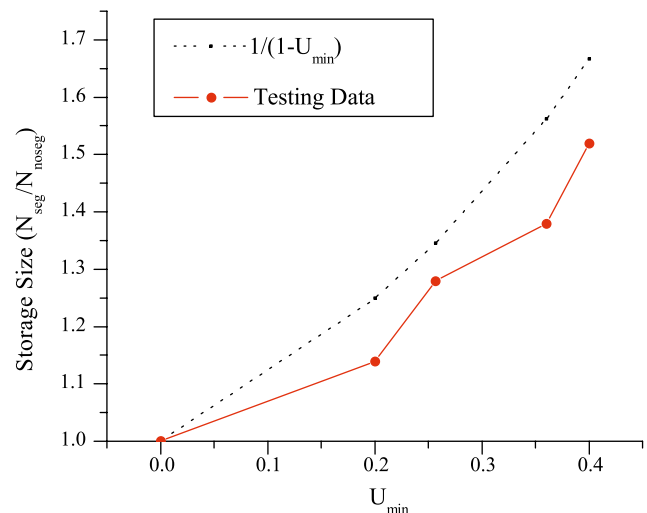

Fig. 7 Storage size for different U_{min}

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

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

The frequency with which new segments are generated is determined by $U_{\min}$ and also the update rates. Suppose the rates (number of tuples per second) for insertion, deletion, and update are R_{ins}, R_{del}, and R_{upd} respectively, and the count of tuples at the beginning of a segment is N_0 (with $U = 100\%$). An insertion will add a new live tuple, a deletion will convert a live tuple to an invalid tuple, and an update will generate a live tuple and an invalid tuple. Throughout its life time a segment must satisfy the condition that the ratio of its live tuples over its total tuples must exceed $U_{\min}$. Thus we must have: $N_{live} \geq U_{\min} \times N_{total}$, where $N_{live} = N_0 + R_{ins} \times T_{seg} + R_{del} \times T_{seg}$ and $N_{total} = N_0 + R_{upd} \times T_{seg} + R_{ins} \times T_{seg}$, with T_{seg} as the age of the segment. In the situations where the rate of insertions exceeds that of deletes and updates, then the inequality $N_{live} \geq U_{\min} \times N_{total}$ is always satisfied, and no new segment is ever generated while the age of the current one keeps increasing. Typically, however, this is not the case, and the usefulness of the segment diminishes as time passes, and soon the old segment must be terminated and a new one will be initiated. We can estimate the time when this occurs, by letting $N_{live} = U_{\min} \times N_{total}$, and thus compute the average age of a segment as follows:

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

Thus higher update rate and/or deletion rate will lead to segments of shorter duration, and higher insertion rate will lead to longer segments. Higher usefulness thresholds will lead to shorter segments.

5.3 Query mapping with clustering

In Section 4.3, we have discussed the general mapping between XQuery and SQL/XML, whereby XQuery upon H-document is translated to SQL/XML upon H-tables. We can now modify our queries in such a way that, when the tstart and tend conditions are specified, we first find the segment number satisfying those conditions and then we use that to restrict the search to only segment(s) of the historical database involved in the query. This operation is made very

efficient by the fact that all indexes are now augmented with a segno information.

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

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

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

6 Performance study

The objective of this study was to determine the effectiveness of alternative implementation approaches in terms of execution and storage cost. Therefore, we evaluated direct implementations of H-documents on native XML databases versus implementation on RDBMS using shredding techniques. We also evaluated the effectiveness of temporal clustering and compression techniques in improving the performance and the storage utilization of our system.

Therefore, in our performance study, we implemented ArchIS on three different platforms: (i) the native XML database Tamino (Enterprise Edition V4.1), (ii) ArchIS–DB2 built on IBM DB2 (Enterprise Edition V7.2), and (iii) ArchIS–ATLaS built on ATLaS [24]. ATLaS is a compact RDBMS developed at UCLA that uses BerkeleyDB as the storage manager and builds an SQL query engine on top of it. Both ArchIS–DB2 and ArchIS–ATLaS use the approach discussed in Sect. 4, but access to the source code of ATLaS allowed us to exercise finer control of optimization. The experiments were performed on a Pentium IV 2.4 GHz PC with RedHat 8.0, with 256 MB memory and an 80 GB ATA hard drive.

We used a temporal employee data set for our testing. The data set archives the history of employees over 17 years, and describes salary raises, title changes, and the department transfers occurred during those years. The total size of the published H-document is 334 MB. As discussed later in this section, an additional data set of 2.28 GB (seven times larger) was also tested to evaluate the scalability of our system.

In our experiments, we made sure to eliminate the effects of OS caching and database buffer pool caching, as follows. To disable Linux OS caching, we first unmounted the hard drive containing the data, and remounted it before running each query. This guarantees the invalidation of Linux page-cache [28]. To disable database buffer caching, the databases

Table 2 Temporal queries on archived history

Q1:	Snapshot(single object): find the salary of an employee 100002 on 05/16/1993;
Q2:	Snapshot: find the average salary of employees on 05/16/1993;
Q3:	History(single object): find the salary history of employee '100002';
Q4:	History: find the total number of salary changes;
Q5:	Temporal slicing: find the number of employees whose salary was more than 60K between 05/16/1993 and 05/16/1994;
Q6:	Temporal join: find the maximum salary increase over a 2 years period after 04/01/2001.

were restarted for each query. We then tested the queries on the following memory sizes: 256, 512 MB and 1 GB. No difference was measured in the query performance, confirming that caching was effectively disabled. We next report the results of our experiments, obtained by taking the average of each query executed seven times.

6.1 Query performance

We investigated three systems to test the query performance: Tamino with H-documents, ArchIS–DB2 with segmented data, and ArchIS–ATLaS with segmented data (9 segments, with U_{min} as 0.4). On Tamino, the documents were automatically compressed for performance (data compression will be further discussed in Sect. 7). On ArchIS–DB2 and ArchIS–ArchIS, the data were stored as H-tables clustered on segments.

We prepared were a set of typical temporal queries such as snapshot (on a single object and on all objects), temporal slicing (on a single object and on all objects), history, and temporal join, as shown in Table 2. In addition, a set of indexes were built for later query comparisons: indexes were created for all nodes/attributes to which selection conditions were applied.

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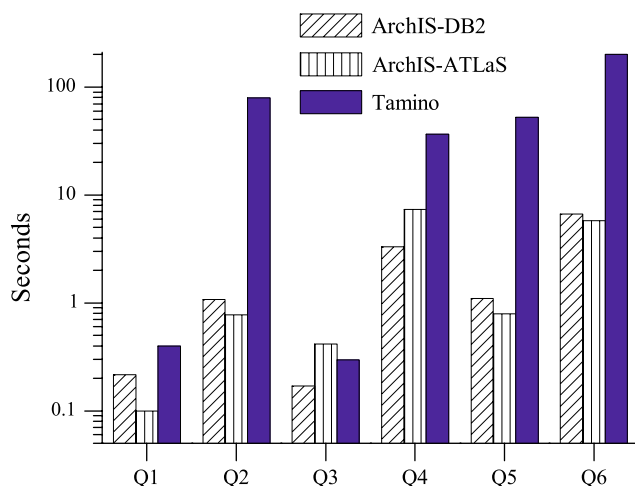
performance advantage over a native XML DB for most queries. The difference of snapshot queries between RDBMS and native XML databases is more significant. For instance, snapshot query Q2 on ArchIS–ATLaS is 102 times faster, and temporal slicing query Q5 is 66 times faster. History query Q4 on ArchIS–ATLaS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

The better results obtained from relational systems reflect the general performance and scalability edge that these still hold over native XML database systems, but it is also due to the fact that no segment-based clustering was used in our Tamino-based implementation. Indeed, we briefly experimented with temporal clustering schemes in Tamino, but we could not find any simple way to cluster the different attributes independently and obtain performance improvements of any significance. The problem of introducing effective temporal clustering and indexing schemes into native XML systems was left for further research.

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

Scalability of ArchIS We used another data set with seven times larger size (2.28 GB), and loaded it into RDBMS as clustered segments. Figure 9 shows that the query execution time of most queries increases approximately linearly. For temporal queries on single object −Q1 and Q3, the time increase is even much less.

6.2 ArchIS versus XTABLES

We also explored the use of XTABLES [7] to support our temporally grouped historical views on the stored H-tables. With XTABLES, users could query the history of database relations using existing commercial software, rather than ArhcIS: however, they would also encounter major problems. The first is that ArchIS' library of special functions designed
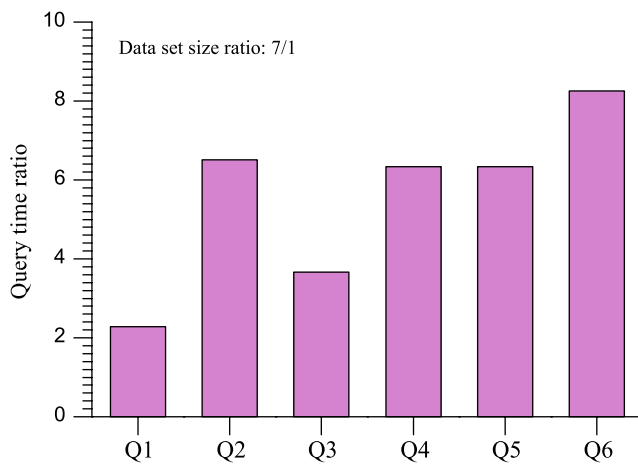

Fig. 8 Query performance of segment-based archiving on RDBMS vs native XML DB

Fig. 9 Query time comparison on ArchIS-DB2 between two data sets (with size ratio: 7/1)

Fig. 11 Query performance with and without segment-based clustering
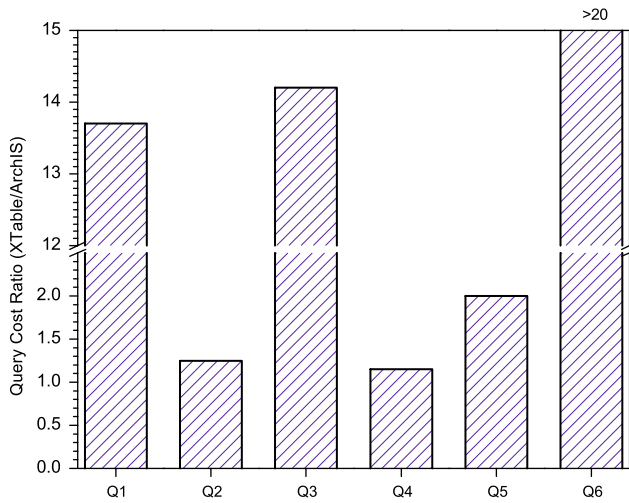

Fig. 10 Comparing XTABLES and ArchIS: query execution time ratios

for temporal queries can not be easily used in this approach. Thus, some temporal queries become very difficult to express and less efficient to execute. The second problem is that even the queries that can be easily expressed without temporal-library functions are significantly less efficient to translate and execute. On the average, XTABLES takes 100 times longer to translate our temporal XQuery statements. More importantly, the execution time also increases dramatically; this is shown in Fig. 10 that compares the execution time on DB2 for queries produced by XTABLES and ArchIS. The ratios between their execution time range between one and two for queries Q2, Q4 and Q5; but XTABLES is more than 13 times slower than ArchIS on queries Q1, Q3 and Q6.

Our study shows that the reasons for this loss of performance are many; however, the main reason is that XTABLES produces additional joins that are redundant given the
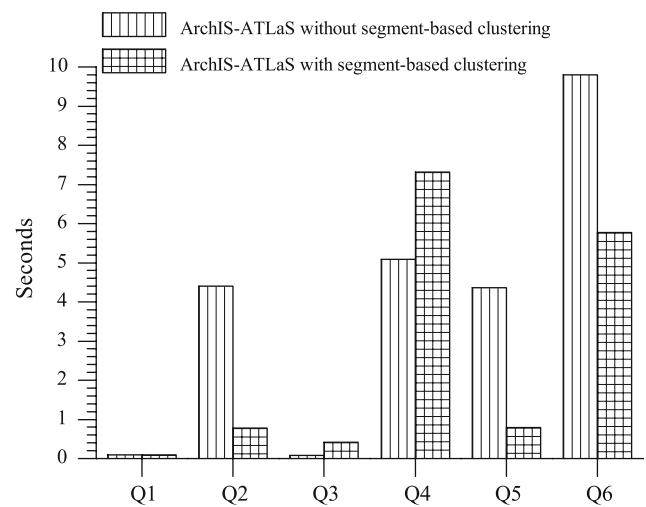
particular semantics of the application at hand. These joins are optimized but not eliminated by the query optimizer.

ArchIS provides additional performance benefits, in addition to the significant ones just discussed; these include the improvements from (i) segment-based temporal clustering, and (ii) storage compression techniques, which are discussed next.

6.3 The Effect of segment-based clustering

Figure 11 compares the query performance with and without segment-based clustering on ArchIS–ATLaS. In our experiments we used $U_{min} = 0.4$ whereby the data was stored in nine segments. This shows that the segment-based clustering scheme significantly boosts the speed for snapshot and temporal slicing queries, e.g., snapshot query Q2 is 5.7 times faster on clustered data than non-clustered data, while temporal slicing query Q5 is 5.5 times faster. Temporal join Q6 is 1.7 times faster with segment-based clustering. The speeds of temporal queries (both Q1 and Q3) on a single object are close for clustering and non-clustering due to the effectiveness of B+ tree index on object IDs. An exception is Q4, which is slower due to the scanning of the whole historical data, and the clustered scheme has a storage redundancy. The value of $U_{min} = 0.4$ was selected on the basis of the experience gained with similar usefulness-based storage scheme [29], which suggests that values in this range deliver a good compromise between the better speed produced by smaller values of U_{min} and the larger space required by such values.

6.4 Storage utilization

We also investigated the storage utilization on the three systems, and discovered that Tamino is very efficient in this
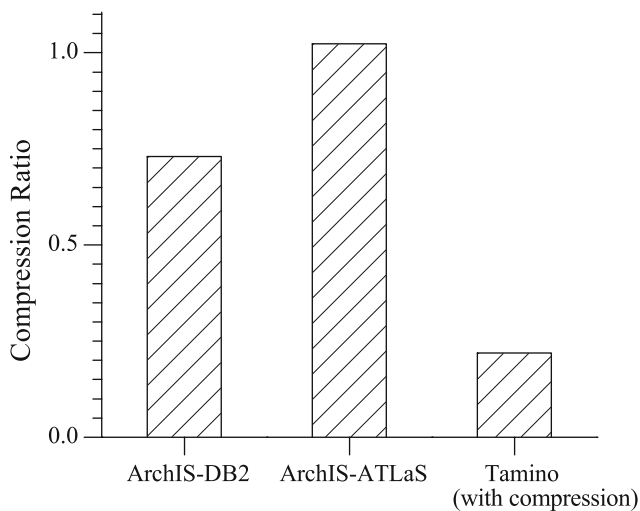
Fig. 12 Compression ratios of H-document storage on different systems

Fig. 13 BlockZIP compresses data into blocks

respect, since Tamino automatically compresses documents with an algorithm similar to gzip.

The compression ratio is defined as the final storage size over H-document size. The compression ratios are 0.22 on Tamino, 0.75 on ArchIS–DB2, and 1.02 on ArchIS–ATLaS respectively (Fig. 12). The storage size in H-tables is half of the H-document size. But with further segment-based clustering, there are redundant tuples among different segments, and the clustering index will take an additional overhead. As a result, ArchIS–DB2 has a compression ratio of 0.75, and ArchIS–ATLaS of 1.02 (ArchIS' storage manager BerkeleyDB uses clustered index which causes extra overhead on storage).

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

7 Database history compression

Next, we propose a block-based compression technique BlockZIP.

7.1 Block-based compression: BlockZIP

BlockZIP is based on zlib [30] (the library version of gzip). Zlib uses the deflation algorithm(an LZ77 variant) and Huffman encoding for data compression. The difference between BlockZIP and zlib is that instead of compressing the data as a whole, it compresses the data as block-sized blocks, and after compression with BlockZIP, the output consists of a set of block-sized compressed blocks concatenated together (Fig. 13). Thus if we know which blocks to
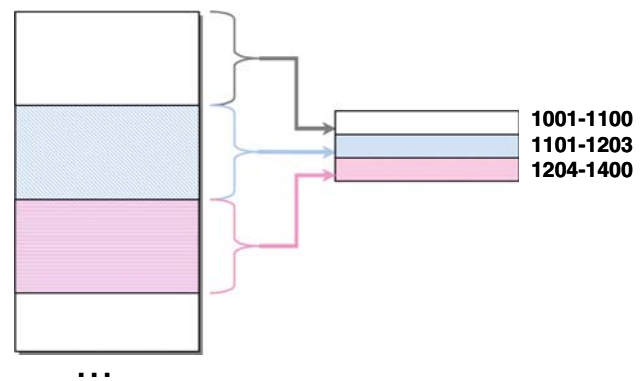
access, we only need to read and uncompress those specific blocks, so uncompressing of the whole file is not needed. The uncompression of such blocks uses exactly the same zlib library functions.

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

7.2 Storage utilization with compression

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

We then compared the storage of the three systems with compression and without compression: Tamino (H-documents), ArchIS–DB2 and ArchIS–ATLaS (the latter two with segment clustered data). Figure 14 compare the compression ratios on different systems, where compression ratio is defined as the data size in the database systems over the size of original H-Documents as stored in the file system. With compression, the storage sizes of ArchIS–DB2 and ArchIS–ATLaS drop significantly and the compression ratio for ArchIS–DB2 (0.23) and ArchIS–ATLaS (0.23) reach very closely to that of Tamino (0.22). Without
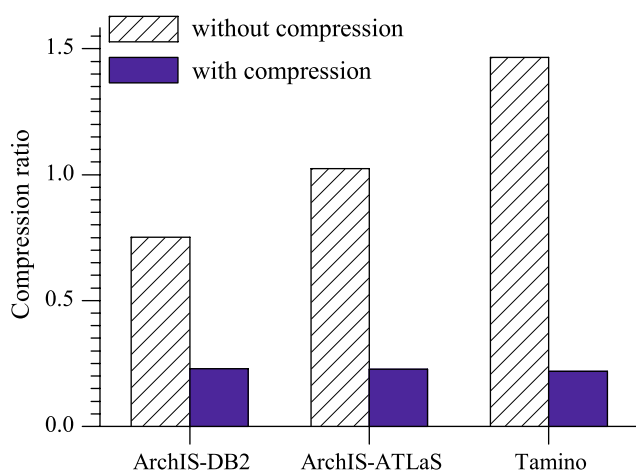
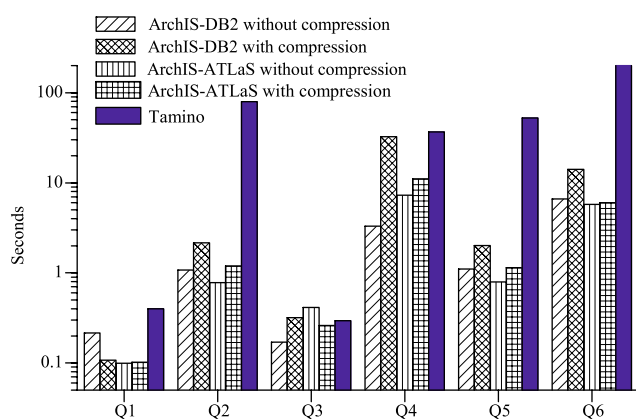**Fig. 14** Compression ratios of historical XML storage on different systems

Fig. 15 Query performance with compression

compression, Tamino's compression ratio is 1.47, a 47% increase from original XML documents.

7.3 Query Performance with compression

We then investigated the query performance on three sets of systems: (a) ArchIS–DB2 and ArchIS–ATLaS with clustered data and with compression; (b) ArchIS–DB2 and ArchIS–ATLaS with clustered data and without compression; and (c) Tamino with non-clustered data and with compression. We use the same queries from Table 2.

Figure 15 shows that RDBMSs without compression have significant performance advantage over a native XML DB, and the benefit of RDBMSs remains for compressed data. With compression, ArchIS–ATLaS and ArchIS–DB2 run the snapshot queries much faster than Tamino, e.g., for snapshot query Q2, ArchIS–ATLaS is 67 times faster than Tamino,

and ArchIS–DB2 is 37 times faster than Tamino. Temporal slicing queries are also much faster on both ArchIS–ATLaS and ArchIS–DB2: Q5 on ArchIS–ATLaS is 46 times faster than on Tamino, and ArchIS–DB2 is 26 times faster. All other historical queries are also faster on ArchIS–ATLaS than on Tamino. For temporal join Q6 on ArchIS–ATLaS, we effectively optimized the join through a user-defined aggregate [31] in one scan, and it takes only 6 s for compressed data.

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

Compression decreases the amount of data being read from the disk—uncompressing one page only costs 0.26 ms, while reading it costs 14ms. However, this doesn't necessarily improve the efficiency of queries, since database performance is determined by many factors. For example, uncompression of a block incurs in the overhead of having to parse the block to extract the records, and DMA reading of continuous blocks can be much faster than seek reading. In queries involving small number of block readings, the performance of compressed mode and uncompressed mode is very close, and the former can even be faster (as shown in Q1).

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

7.4 Update performance

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

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

As another example, for a simulated daily update, the cost is 15 s for Tamino, and 1.52 s for ArchIS–DB2. While normally updates are significant faster in ArchIS–ATLaS than in Tamino, with ArchIS–ATLaS we also have the occasional situations where the current segment's usefulness is below the threshold, and all current data are archived into a new segment. This takes 39 s, and if such segment is to be output

and compressed, it takes an additional 36 s. However, the archiving of each segment only occurs once.

8 Related work

8.1 Temporal XML

The growing recognition of the importance of supporting time in XML and representing the version history of XML documents has produced a rich vein of recent research. Thus, Grandi [32] provides a bibliography on temporal aspects in the Web, while Ali and Pokorny [33] provide a comparison of XML-based data models.

Marian et al. [34] propose an approach for managing web data warehouse for the Xyleme project. Their approach is based on structured diff algorithms for XML documents. Efficient storage schemes based on structured diff representations and durable node numbers are presented by Chien et al. [29], [35]. In [36], Buneman et al. present an archiving technique for scientific data, where semantic continuity of each data element in the archive is preserved through keys. The paper by Buneman et al. must also be credited with introducing hierarchical timestamping in XML documents, an idea used in our paper. However, the temporal query aspects of the problem are not discussed in [36]. In [37,38], Gergatsoulis and Stavraska propose a multidimensional XML model where dimensions are applied to elements and attributes to keep temporal information; however, they do not discuss how to effectively query these temporal extensions. In [39], Grandi et al. discuss the problem of managing temporal normative texts along the temporal dimensions of publication, validity, efficacy and transaction times; their approach relies on Oracle XML extensions to manage XML documents as CLOB type. In [40], Manukyan et al. analyze the temporal components of XMl document, while Currim et al. [41] propose to extend XML Schema to support temporal XML information.

While the before mentioned approaches do not delve deep into query aspects, Amagasa et al. [42] introduce a temporal data model that is based on XPath. Extensions of XPath based on the addition of special temporal axes are proposed by Dyerson [43] for transaction-time support, and then by Dyerson and Zhang [44] for valid-time support. Gao and Snodgrass [45] introduce a valid-time generalization of XQuery, called τXquery, which does not require extending the XML data model: τXquery statements can be translated into equivalent XQuery statements [45], but questions remain on the efficiency of the rewritten queries. Generally speaking, achieving good performance for temporal queries on large complex XML documents represents an unsolved technical challenge, which researchers have been addressing by special indexes or storage structures, or by recasting the documents and the queries into relational DBMS and O-R

DBMS. For instance, an ORDBMS-based implementation of temporal XML management with a temporal language TXSQL is discussed by Norvag et al. [46,47]. In passing, we observe that all these approaches confirm the native ability of XML and its query languages to deal with temporal information at the logical level.[4]

Among the various indexing techniques used to improve performance of temporal queries we will mention the ToXin approach by Mendelzon et al. [48], and the techniques for the efficient support of temporal slicing of Mandreoli et al. [49]. In their recent work, Rizzolo and Waisman [50] propose a new approach for summarizing and indexing temporal documents. The temporal XML representation used in [50] is not strictly hierarchical since a node can point to a subelement that, over time, has migrated to another node. Simple temporal queries and the complex operations needed to maintain the semantic constraints on this non-hierarchical structure are also discussed in [50].

Therefore, as described in [50], for deeply nested XML documents there are situations in which it might be desirable to go beyond hierarchies by allowing the sharing of common subelements. However, this is not the situation of our H-documents, since the temporally grouped representation of the history of flat relations produces shallow hierarchies. Likewise, while the temporal indexing techniques above are very desirable for general XML documents, ArchIS' approach of shredding back into relations performs much better on the regular and shallow structure of our H-documents, as it is clearly demonstrated by the results of our experiments.

8.2 Semistructured and Object-Oriented DBMS

Early work on semistructured and O-O DBMS contains the seeds of ideas that were then used in XML. For instance, Chawathe et al. [51] describe a model for managing historical semistructured data where updates are represented by their "deltas". Oliboni et al. [52] propose a graph-based model and query language for transaction-time semistructured DBMS that was then refined in [53]. In [54], Dyreson et al. show how annotations on the edges of the database graphs can be used to capture transaction time and valid time, with other kinds of information.

The problem of version management in object-oriented and CAD databases has received a significant amount of attention [55,56]. A formal temporal object-oriented data model is proposed in [57] and a comparison between different object-oriented temporal models was presented by Snodgrass [58].

---

[4] Unfortunately, the situation is quite different at the physical level, since good query performance and scalability represent elusive goals for native XML—and even more so for its temporal extensions.

8.3 Relational databases

Relational databases provided the first battleground for temporal database research—producing numerous advances and retreats that we will not recount in this paper since they are already covered in the authoritative survey by Ozsoyoglu and Snodgrass [3]. Among the work discussed in [3], we would like to acknowledge the contributions by Snodgrass[14] and Chomicki et al. [16], which present a comprehensive set of examples and a taxonomy of temporal queries that we have used in this paper to verify the completeness of our approach in terms of temporal query expressiveness. In terms of data models, we have much relied on the seminal work by Clifford et al. [4] who classify temporal representations into the two main categories of *temporally ungrouped* and *temporally grouped* data models. The second representation has more expressive power and is more natural since it is history-oriented [4]. TSQL2 [15] tries to reconcile the two approaches [4] within the severe limitations of the relational tables. Our approach is based on a temporally grouped data model, which dovetails perfectly with the hierarchical structure of XML documents, although temporal implementations based on nested relations in ORDBMS are also possible as discussed in [59].

While the survey presented in [3] illustrates how the design space for temporal extensions has been exhaustively explored at the logical level, much work remains to be done at the physical level. Thus, TimeDB [60] proposes a layered architecture that translates temporal queries into RDBMS, where temporal data is represented as tuples with intervals, thus temporally ungrouped. Renewed interest on temporal databases is now surfacing in the commercial world. For instance, Flashback [61] and ImmortalDB [10] allow users to rollback to old versions of tables (e.g., to correct old errors)—although they do not provide support for complex temporal queries.

While previous research on temporal databases has mostly focused on extending database data model and query languages, we have shown here that the existing primitives of XML and relational DBMS can be skillfully combined to realize a transaction-time temporal database that supports a simple logical model, powerful temporal queries, and good performance on current DBMS. The use of XML in publishing and querying database history was previously proposed in [62]. No efficient system implementation was however discussed in [62], and neither were the key pieces of the enabling technology that make it run, including SQL/XML, temporal indexing, clustering and compression.

9 Conclusion

In this paper, we have shown the history of a relational database 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 approach realized by ArchIS is efficient and general, and can be used to add a transaction-time capability to existing RDBMS and applications (as in the employee/departmet example), and to support fast growing new application areas, such as RFID. The approach is also complete, since its realization does not require the invention of new techniques, nor costly extensions of existing standards. This paper elucidates the query mapping, indexing, clustering, and compression techniques used to achieve performance levels well above those of a native XML DBMS, as demonstrated by several experiments presented in the paper.

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

At the physical level, many clustering and indexing techniques have been proposed for temporal databases [63] and deserve further investigations. Also other efficient data compression techniques proposed for XML data deserve further study [64].

Many interesting research questions also arise if we consider natural generalizations of our approach, and its possible applications to (i) valid-time databases and bitemporal databases, (ii) O-R DBMSs, and (iii) arbitrary XML documents. The question on whether the approach here proposed was also applicable to valid-time and bitemporal databases was studied in [65], where it was concluded that ArchIS's temporally grouped XML model and temporal queries at the logical level. At the physical level, however, the segment-based temporal indexing and clustering used in ArchIS will no longer be effective, and the query optimization and data compression techniques presented also need changes. Thus the extension of ArchIS for valid-time and bitemporal databases represents a challenging topic for future research.

The second question involves the applicability of approaches similar to the one we have proposed to other data models, including object-oriented models and semistructured data models other than XML. Our intuition suggests that, not only the answer to these questions is largely positive, but, surprisingly enough, much of our approach to temporal information management is applicable to SQL itself. Indeed, the most recent SQL:2003 standards support nested relations [66] that can be used to support a temporally grouped data model. Simple temporal queries can be expressed in SQL itself, while more complex queries could require the use of a library of temporal functions and aggregates similar to those that we have developed for ArchIS. This suggests that standard database systems of the future will be able to

manage efficiently temporal information, and also give users a choice on whether to operate under XML standards or SQL standards—while their support is unified and optimized at the internal level.

The temporally grouped data model and timestamping scheme used here can be generalized to support evolution queries on multi-version XML documents [67]. That scheme makes it possible to ask interesting temporal queries on the evolution of standards, e.g., the successive revision of XLink standards, or, from the history of university catalogs, when a new course was first introduced. Thus the XML-based approach here introduced represents a significant first step towards adding historical information management and query capabilities to databases and web information systems, since the preservation of digital artifacts represents a critical open issue for the information age.

References

1. Snodgrass, R.T.: Developing Time-Oriented Database Applications in SQL. Morgan Kaufmann, San Francisco (1999)
2. Wang, F., Liu, P.: Temporal Management of RFID Data. In: VLDB (2005)
3. Ozsoyoglu, G., Snodgrass, R.T.: Temporal and real-time databases: a survey. TKDE **7**(4), 513–532 (1995)
4. Clifford, J., Croker, A., Grandi, F., Tuzhilin, A.: On temporal grouping. In: Recent Advances in Temporal Databases, pp. 194–213. Springer, Heidelberg (1995)
5. XQuery 1.0: An XML Query Language. http://www.w3.org/XML/Query
6. Kepser, S.: A Simple Proof for the Turing-Completeness of XSLT and XQuery. In: Extreme Markup Languages (2004)
7. Funderburk, J.E., Kiernan, G., Shanmugasundaram, J., Shekita, E., Wei, C.: XTABLES: bridging relational technology and XML. IBM Syst. J. **41**(4), (2002)
8. Oracle XML. http://otn.oracle.com/xml/
9. SQL/XML.http://www.sqlx.org
10. Lomet, D., Barga, R., Mokbel, M.F., Shegalov, G., Wang, R., Zhu, Y.: Transaction time support inside a database engine. In: ICDE (2006)
11. Böhlen, M.H., Snodgrass, R.T., Soo, M.D.: Coalescing in temporal databases. In: VLDB (1996)
12. Clifford, J.: Formal Semantics and Pragmatics for Natural Language Querying. Cambridge University Press, Cambridge (1990)
13. Clifford, J., Croker, A., Tuzhilin, A.: On completeness of historical relational query languages. ACM Trans. Database Syst. **19**(1), 64–116 (1994)
14. Snodgrass, R.T.: The TSQL2 Temporal Query Language. Kluwer, Dordrecht (1995)
15. Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R.T., Subrahmanian, V.S., Zicari, R.: Advanced Database Systems. Morgan Kaufmann, San Francisco (1997)
16. Chomicki, J., Toman, D., Böhlen, M.H.: Querying ATSQL databases with temporal logic. TODS **26**(2), 145–178 (2001)
17. Clifford, J., Dyreson, C.E., Isakowitz, T., Jensen, C.S., Snodgrass, R.T.: On the semantics of "Now" in databases. TODS **22**(2), 171–214 (1997)
18. Torp, K., Jensen, C.S., Snodgrass, R.T.: Modification semantics in now-relative Databases. Inf. Sys. (2007, in Press)
19. EPC Tag Data Standards Version 1.1. Technical report, EPCGlobal Inc, (2004)
20. Oracle Sensor Edge Server. http://www.oracle.com/technology/products/iaswe/edge_server
21. WebSphere RFID Premises Server. http://www-306.ibm.com/software/pervasive/ws_rfid_premises_server/, accessed on December 2004
22. Floerkemeier, C., Anarkat, D., Osinski, T., Harrison, M.: PML Core Specification 1.0. Technical report, Auto-ID Center (2003)
23. Schöning, H.: Tamino—a DBMS designed for XML. In: ICDE (2001)
24. ATLaS. http://wis.cs.ucla.edu/atlas
25. DeHaan, D., Toman, D., Consens, M.P., Ozsu, M.T.: A comprehensive XQuery to SQL translation using dynamic interval encoding. In: SIGMOD (2003)
26. Shanmugasundaram, J., et al.: Efficiently publishing relational data as XML documents. In: VLDB (2000)
27. Galax–an Open Source XQuery Implementation. http://www.galaxquery.org
28. Bovet, D.P., Cesati, M.: Understanding the Linux Kernel, 2nd edn. O'Reilly, Cambridge (2002)
29. Chien, S.-Y., Tsotras, V.J., Zaniolo, C.: Efficient schemes for managing multiversion XML documents. VLDB J. **11**(4), 332–353 (2002)
30. Zlib. http://www.gzip.org/zlib/
31. Wang, H., Zaniolo, C.: Using SQL to build new aggregates and extenders for object-relational systems. In: VLDB (2000)
32. Grandi, F.: Introducing an annotated bibliography on temporal and evolution aspects in the world wide web. SIGMOD Record **33**(2), 84–86 (2004)
33. Ali, K., Pokorny, J.: A comparison of XML-based temporal models. In: SITIS (2006)
34. Marian, A., Abiteboul, S., Cobena, G., Mignet, L.: Change-centric Management of versions in an XML warehouse. VLDB J. 581–590 (2001)
35. Chien, S.-Y., Tsotras, V.J., Zaniolo, C., Zhang, D.: Supporting complex queries on multiversion xml documents. ACM Trans. Internet Techn. **6**(1), 53–84 (2006)
36. Buneman, P., Khanna, S., Tajima, K., Tan, W.: Archiving scientific data. TODS **29**(1), 2–42 (2004)
37. Gergatsoulis, M., Stavrakas, Y.: Representing changes in XML documents using dimensions. In: Xsym (2003)
38. Gergatsoulis, M., Stavrakas, Y., Doulkeridis, C., Zafeiris, V.: Representing and querying histories of semistructured databases using multidimensional OEM. Inf. Syst. **29**(6), 461–482 (2004)
39. Grandi, G., Mandreoli, F., Tiberio, P.: Temporal modelling and management of normative documents in XML format. Data Knowledge Engr. **54**(3), 327–254 (2005)
40. Manukyan, M.G., Kalinichenko, L.A.: Temporal XML. ADBIS'01 (2001)
41. Currim, F., Currim, S., Dyreson, C., Snodgrass, R.T.: A tale of two schemas: creating a temporal schema from a snapshot schema with τXSchema. In: EDBT (2004)
42. Amagasa, T., Yoshikawa, M., Uemura, S.: A data model for temporal XML documents. In: DEXA (2000)
43. Dyreson, C.E.: Observing transaction-time semantics with TTXPath. In: WISE (2001)
44. Zhang, S., Dyreson, C.: Adding valid time to XPath. In: DNIS (2002)

45. Gao, D., Snodgrass, R.T.: Temporal slicing in the evaluation of XML Queries. In: VLDB ArchIS 35 (2003)
46. Nørvåg, K., Limstrand, M., Myklebust, L.: TeXOR: temporal XML database on an object-relational database system. In: PSI (2003)
47. Nørvåg, K.: The design, implementation, and performance of the v2 temporal document database system. Inf. Software Technol. **46**(9), 557–574 (2004)
48. Mendelzon, A.O., Rizzolo, F., Vaisman, A.: Indexing temporal XML documents. In: VLDB (2004)
49. Mandreoli, F., Martoglia, R., Ronchetti, E.: Supporting temporal slicing in XML databases. In: EDBT (2006)
50. Rizzolo, F., Vaisman, A.: Temporal xml: modeling, indexing and query processing. Int. J. Very Large Databases (Published Online 7 July 2007)
51. Chawathe, S.S., Abiteboul, S., Widom, J.: Managing historical semistructured data. TAPOS **5**(3), 143–162 (1999)
52. Oliboni B., Quintarelli E., Tanca L. (2001) Temporal aspects of semistructured data. TIME 119–127
53. Combi, C., Oliboni, B., Quintarelli, E.: A graph-based data model to represent transaction time in semistructured data. In: DEXA (2004)
54. Dyreson, C.E., Böhlen, M.H., Jensen, C.S.: Capturing and querying multiple aspects of semistructured data. In: VLDB 290–301 (1999)
55. Chou, H., Kim, W.: A unifying framework for version control in a CAD environment. In: VLDB (1986)
56. Beech, D., Mahbod, B.: Generalized version control in an object-oriented database. In: ICDE 14–22 (1988)
57. Bertino, E., Ferrai, E., Guerrini, G.: A formal temporal object-oriented data model. In: EDBT (1996)
58. Snodgrass, R.T.: Temporal object-oriented databases: a critical comparision. Addions-Wesley/ACM Press, Reading/London (1995)
59. Wang, F., Zhou, X., Zaniolo, C.: Bridging relational database history and the web: the XML approach. In: WIDM (2006)
60. Steiner, A.: A generalisation approach to temporal data models and their implementations. PhD thesis, ETH Zurich (1997)
61. Oracle Flashback Technology. http://otn.oracle.com/deploy/availability/htdocs/flashback_overview.htm
62. Wang, F., Zaniolo, C.: An xml-based approach to publishing and querying the history of databases. World Wide Web **8**(3), 233–259 (2005)
63. Salzberg, B., Tsotras, V.J.: Comparison of access methods for time-evolving data. ACM Comput. Surv. **31**(2), 158–221 (1999)
64. Liefke, H., Suciu, D.: XMILL: an efficient compressor for XML data. In: SIGMOD 153–164 (2000)
65. Wang, F., Zaniolo, C.: XBiT: an XML-based bitemporal data model. ER (2004)
66. Database Languages SQL, ISO/IEC 9075-*:2003
67. Wang, F., Zaniolo, C.: Temporal Queries in XML document archives and web warehouses. In: TIME-ICTL (2003)