

# Experimenting with Temporal Relational Databases

Iqbal A. Goralwalla<sup>a</sup>, Abdullah U. Tansel<sup>b</sup> and M. Tamer Özsu<sup>a</sup>

<sup>a</sup>Laboratory for Database Systems Research,  
Department of Computing Science, University of Alberta  
Edmonton, Alberta, Canada T6G 2H1.  
{iqbal,ozsu}@cs.ualberta.ca  
Tel: (403) 492 2860, Fax: (403) 492 1071

<sup>b</sup>Baruch College—CUNY,  
17 Lexington Avenue, Bo x 513,  
New York NY 10010, USA.  
uztbb@cunyvm.bitnet

## Abstract

In this paper we describe an implementation of a temporal relational database management system based on attribute timestamping. The algebraic language of the system includes relational algebra operators, restructuring operators and temporal operators. We then use this system to carry out experiments on the performance of different types of temporal databases: databases using attribute timestamping, databases using tuple timestamping where relations are in temporal normal form and databases using tuple timestamping where a single relation is used. We run sample queries against these types of temporal databases and measure the processing time of these queries. This study verifies that the major performance trade off between different types of temporal databases is between the restructuring (*unpack*) operation needed in temporal databases using attribute timestamping and the *join* operation needed in temporal databases using tuple timestamping. Furthermore, the experiments show that keeping all temporal tuples in one single relation does not prove to be an effective alternative for temporal databases which use tuple timestamping.

**Keywords:** temporal databases, attribute timestamping, tuple timestamping, temporal queries.

## 1 Introduction

A database contains data pertaining to an organization and its activities. It forms a data repository from which information is extracted for various purposes. Conventional databases can be viewed as **snapshot** databases in that they represent the state of an enterprise at one particular time, i.e., they contain only current data. As a database changes, out-of-date information representing past states of the enterprise is discarded.

Time is an attribute of all real-world phenomena. Events occur at specific points in time; objects and the relationships among objects exist over time. The ability to model this temporal dimension of the real world is essential to many information system applications. Examples of these are econometrics, banking, inventory control, medical records, airline reservations, versions in CAD/CAM applications, statistical and

scientific data, etc. The attributes of an object (i.e., an entity or a relationship) assume different values over time. The set of these values form the history of that object. A database which maintains past, present and future data (i.e., object histories) is called a **Temporal Database (TDB)**.

In the last decade there has been extensive research activity on temporal databases. An initial summary of research projects can be found in [24] which also includes a bibliography on temporal databases [15]. Additional bibliographies on temporal databases are given in [27, 26]. Furthermore, Kline provides an update to previous bibliographies on temporal databases in [13]. Snodgrass [25] reports on the status and research directions in temporal databases while Pissinou et al. [19] report on the outcome of a workshop on the infrastructure for temporal databases. A recent book [30] provides a comprehensive treatment of the current state-of-the-art in temporal databases.

Most of the research on temporal databases has concentrated on extending the relational model [5] to handle time in an appropriate manner. These extensions can be grouped into two main categories. The first approach uses First Normal Form (1NF) relations in which special time attributes are added to a relation and the history of an object (attribute) is modeled by several 1NF tuples [1, 14, 16, 20, 23]. This approach is known as tuple timestamping. The second approach uses Non-First Normal Form (N1NF) relations in which time is attached to attribute values of a relation and the history of an object (attribute) is modeled by a single N1NF tuple [4, 9, 28]. This approach is known as attribute timestamping.

Although substantial research activity on temporal databases has been going on, modeling temporal data and temporal query languages have been the topics most investigated. There has been very little work in the implementation of temporal databases. TQUEL [23] is the only exception which is a prototype implementation on INGRES database system. The underlying model is based on attaching time stamps to tuples, hence 1NF relations. Moreover, even though tuple timestamping and attribute timestamping are the two main approaches of modeling temporal relational databases, there has been no practical study done to point out and thereby qualify the theoretical trade-offs in using either approach.

In this paper we describe the implementation of a memory resident prototype temporal relational database management system [10] (TDBMS) which uses attribute timestamping (**AT**). We feel the attribute timestamping approach is closer to user thought process as compared to tuple timestamping. In addition, there is minimum data redundancy as all historical data belonging to an object (attribute) is modeled in one single tuple. We use TDBMS to generate two types of databases which use tuple timestamping. In the first database (**TT**), time varying attributes are distributed over multiple relations, each in the temporal normal form described in [16]. In the second database (**TTL**), a single relation is used to hold all its pertaining time varying attributes. We then investigate the performance of **AT**, **TT** and **TTL** databases

on different types of queries. We see the following as the two main contributions of this paper:

1. Our prototype is the first implementation of a temporal database management system based on attribute timestamping. It demonstrates the feasibility of such databases. Even though it needs more work for incorporating indexing for query optimization, in its present form it provides a useful environment for experimentation with temporal databases.
2. We use the implemented system to study the performance of the two main approaches of modeling temporal databases. This is the first practical study which compares the attribute timestamping and tuple timestamping approaches. The results reported here will provide useful insight into the design of temporal databases.

The rest of the paper is organized as follows. In Section 2, the temporal relational model together with its algebra is given. Section 3 describes the implementation of the temporal relational database management system. In Section 4 various experiments are conducted on the performance of different temporal database types. The paper concludes with a discussion and avenues for future research in Section 5.

## 2 The Temporal Relational DBMS (TDBMS)

In our temporal model, each time-varying attribute value is a  $\langle \text{time}, \text{value} \rangle$  pair. The time part of this pair is taken to be the *interval* in which the value is valid as opposed to the *time point* at which the value became valid. The latter approach creates complications in expressing and interpreting the relational algebra operations since it splits the time interval between two successive pairs, causing the successor pair to be examined if the time duration over which the value is valid is needed. Hence, we opted for the former approach and represent the time component of time-varying attributes as intervals.

Let  $V$  be the set of all possible integers, reals, and strings and  $T$  be a set of time points which is totally ordered under the less than-equal-to ( $\leq$ ) relation. The points are identified relative to an origin  $t_0$  as shown below:

$$\begin{aligned}
 T &= \{t_0, t_1, \dots, t_i, \dots, \text{now}\} \\
 t_0 &< t_1 \dots < t_i < \dots \text{now} \\
 t_i &= t_{i-1} + 1 \text{ and } t_i = t_0 + i
 \end{aligned}$$

$t_0$  is the starting time and *now* is the marking symbol for the current time. Time-varying attribute values are represented as triplets of the form,  $\langle [l, u], v \rangle$ .  $[l, u]$  is the time component (interval), with  $l$

E#	Ename	*\$Department	*\$Salary
111	Tom	{<[Jan 84, Jan 85], Shoe>, <[Jan 85, now], Toys>}	{<[Jan 84, June 85], 20K>, <[June 85, now], 25K>}
122	Ann	{<[Jan 86, now], Sales>}	{<[Jan 86, now], 30K>}
133	John	{<[Jan 89,now], Toys>}	{<[Jan 89, June 90],32K>, <[June 90, now], 40K>}

Figure 1: *Employee* relation.

and  $u$  standing for the lower and upper bounds of the interval respectively.  $v$  ( $v \in V$ ) is the data value which is valid over the time interval  $[l,u)$ .

Sets of triplets represent the history of an attribute. A relation containing such attributes is called a **valid time relation**. In our model, a valid time relation may have four types of attributes:

- *Atomic* attributes: contain atomic values from domains which are subsets of  $V$ .
- *Triplet-valued* attributes: contain triplets as atomic values. A triplet is of the form  $\langle [l, u), v \rangle$  where  $l, u \in T$  and  $v \in V$ .
- *Set-valued* attributes: are sets of atomic values. These values are considered independent of time.
- *Set-triplet valued* attributes: contain sets of triplets as values. Each set is a collection of one or more triplets defined over a subset of the interval  $[t_0, now]$ , and represents the attribute's history.

In order to differentiate different types of attributes we use the following notation. Atomic attributes are referred to by their names, Set-valued attributes are prefixed by a “\*” , e.g., \*A, \*B, triplet-valued attributes are prefixed by a dollar sign, e.g., \$A, \$B, and set-triplet valued attributes are prefixed by a star and dollar, e.g., \*\$A, \*\$B, etc. To refer to the components of a triplet-valued attribute, \$A, we use  $\$A_l$ ,  $\$A_u$  and  $\$A_v$ . They refer to the lower bound, upper bound, and value components of the triplet, respectively. A valid time relation is a nested relation with one level of nesting, i.e., attribute values can be sets. Figure 1 gives an example valid time relation. All tuples of the *Employee* relation are homogeneous [9]. TDBMS however, allows non-homogeneous tuples as well.

## 2.1 Temporal Relational Algebra

The set of Temporal Relational Algebra (TRA) operations includes the five basic operations of the relational algebra with slight modifications for handling temporal data, the two re-structuring operations, **pack** and

E#	Ename	\$Department	*\$Salary
111	Tom	<[Jan 84, Jan 85), Shoe>	{<[Jan 84, June 85), 20K>, <[June 85, now], 25K>}
111	Tom	<[Jan 85, now], Toys>	{<[Jan 84, June 85), 20K>, <[June 85, now], 25K>}
122	Ann	<[Jan 86, now], Sales>	{<[Jan 86, now), 30K>}
133	John	<[Jan 89,now], Toys>	{<[Jan 89, June 90), 32K>, <[June 90, now], 40K>}

Figure 2:  $Employee_{unpack}$  relation.

**unpack** [17] which directly apply to temporal relations, and new operations for manipulating temporal data. The formal definitions of the TRA operations can be found in [28, 29]. We now describe the TRA operations we have implemented and illustrate the workings of each operation using the Employee relation given in Figure 1:

1. **Set operations (union, intersection, difference)/Cartesian product/Project/Natural Join:**

These operations are the same as the relational algebra operations.

2. **Selection (select  $A$  from  $R$  where  $F$ ):** The selection operator selects the tuples from the valid time relation  $R$  which satisfy the formula  $F$  and projects out the attribute(s)  $A$ .  $F$  is a formula which could contain any type of attribute, comparison operators (i.e., =,  $\neq$ , >,  $\geq$ , <,  $\leq$ ) and connectives (and, or, not). These comparison operators have been used instead of temporal operators like *overlap*, *during*, etc. to keep the implementation simple. Note that temporal operators add to the user-friendliness of the query. However, omitting them does not diminish the expressive power of TRA. In the Selection operation when a triplet-valued attribute  $\$X$  is used, its components can be referenced. Note that our selection operation includes the selection and projection operations of relational algebra.

3. **Unpack (unpack  $R$  on  $A$ ):** The unpack operation creates a family of tuples for each tuple of  $R$  when it is applied on one of  $R$ 's set-valued or set-triplet valued attributes,  $A$ . One tuple is created for each element of the set in the attribute value. As an example, the result of **unpack**  $Employee$  **on**  $Department$  is shown in Figure 2.

4. **Pack (pack  $R$  on  $A$ ):** The pack operation when applied to an atomic or triplet-valued attribute  $A$  of a valid time relation  $R$ , collects the values in attribute  $A$  into a single tuple component for tuples whose remaining attributes agree. For example, applying pack on the  $Department$  attribute of  $Employee_{unpack}$  yields back the  $Employee$  relation.

E#	Ename	Department	From	To	*\$Salary
111	Tom	Shoe	Jan 84	Jan 85	{<[Jan 84, June 85), 20K>, <[June 85, now], 25K>}
111	Tom	Toys	Jan 85	now	{<[Jan 84, June 85), 20K>, <[June 85, now], 25K>}
122	Ann	Sales	Jan 86	now	{<[Jan 86, now), 30K>}
133	John	Toys	Jan 89	now	{<[Jan 89, June 90), 32K>, <[June 90, now], 40K>}

Figure 3:  $Employee_{tdec}$  relation.

5. **Triplet-decomposition** ( $tdec R$  on  $\$A$ ): This operation breaks the triplet valued attribute  $\$A$  of the valid time relation  $R$  into its components. It adds two new attributes for representing the upper and lower bounds of the time interval to  $R$ . The value field replaces  $\$A$ . For example,  $tdec Employee_{unpack}$  on  $Department$  gives the relation shown in Figure 3.
6. **Triplet-formation** ( $tform R$  on  $V, L, U$ ):  $tform$  creates a triplet-valued attribute from the three attributes  $V, L$ , and  $U$  of the valid time relation  $R$  which correspond to the value component, lower bound, and upper bound of the triplet respectively. Triplet-formation is the inverse operation of triplet-decomposition. For example,  $tform Employee_{tdec}$  on  $Department, From, To$  gives back the  $Employee_{unpack}$  relation.
7. **Slice** ( $slice R \$A$  by  $\$B$ ): Let  $\$A$  and  $\$B$  be two triplet-valued attributes. The slice operator trims the time of  $\$A$  with respect to the time of  $\$B$ . In other words, the intersection of intervals of  $\$A$  and  $\$B$  is assigned to  $\$A$  as its time reference. There are two other versions of this operation. **Uslice** operation forms the *union* of the time intervals of  $\$A$  and  $\$B$  and assigns it as the time component of  $\$A$ . **Dslice** operation is similar to *slice*, except the *difference* of the time intervals of  $\$A$  and  $\$B$  is assigned to the time component of  $\$A$ . In both *Uslice* and *Dslice* operations, attribute  $A$  becomes a set-triplet valued attribute, since union (difference) of two intervals is not one single interval. For example, after unpacking the  $Employee$  relation on the  $Department$  and  $Salary$  attributes,  $slice Employee Department$  by  $Salary$  gives the relation shown in Figure 4.
8. **Drop-time** ( $droptime R$  on  $A$ ): This operation gets rid of the time components of the triplet-valued or set-triplet valued attribute  $A$  and converts it into an atomic or a set-valued attribute, respectively. For example, Figure 5 shows the  $Employee$  relation with the time of the  $Department$  attribute dropped.

E#	Ename	\$Department	\$Salary
111	Tom	<[Jan 84, Jan 85), Shoe>	<[Jan 84, June 85) 20K>
111	Tom	<[Jan 85, June 85), Toys>	<[Jan 84, June 85), 20K>
111	Tom	<[June 85, now], Toys>	<[June 85, now], 25K>
122	Ann	<[Jan 86, now], Sales>	<[Jan 86, now], 30K>
133	John	<[Jan 89, June 90), Toys>	<[Jan 89, June 90), 32K>
133	John	<[June 90, now], Toys>	<[June 90, now], 40K>

Figure 4:  $Employee_{slice}$  relation.

E#	Ename	*Department	*\$Salary
111	Tom	{Shoe, Toys}	{<[Jan 84, June 85), 20K>, <[June 85, now], 25K>}
122	Ann	{Sales}	{<[Jan 86, now], 30K>}
133	John	{Toys}	{<[Jan 89, June 90), 32K>, <[June 90, now], 40K>}

Figure 5:  $Employee_{dtime}$  relation.

9. **Enumeration** (ENUM1, ENUM2): The enumeration operation derives a relation of uniform data for a set of specified time points or intervals from a valid time relation. To handle the semantic issues which arise when applying aggregates to historical data, two versions of the enumeration operation have been developed. Let  $X_1$  be a set of atomic attributes and let  $X_2$  be a set of set-triplet valued attributes. ENUM1 ( $\mathbf{enum1} R <\{X_1\} \{X_2\} > \{T\}$ ) returns the values of the designated attributes at the specified time points. For each of the specified *time points* in  $T$ , each tuple of the valid time relation  $R$  is retrieved. The triplets of the set-triplet valued attribute(s)  $X_2$  are examined in turn for intersection with the specified time point. If an intersection is found, the data values of the attribute(s)  $X_1$  and  $X_2$  are retrieved. For example,  $\mathbf{enum1} Employee <\{E\#\} \{Department, Salary\} > \{“010185” “010190”\}$  gives the relation shown in Figure 6.

On the other hand, ENUM2 ( $\mathbf{enum2} R <\{X_1\} \{aggrf(X_2)\} > \{T\}$ ) returns an appropriate aggregated value for an attribute by utilizing all of its values which are valid over a specified *time interval* in  $T$ . For each of the specified time intervals in  $T$  the triplets of the set-triplet valued attribute  $X_2$  are examined in turn for intersection. The aggregation function, *aggrf* is then applied to all qualifying data values of attribute  $X_2$ . The aggregated value and attribute(s)  $X_1$  are then retrieved. For example,  $\mathbf{enum2} Employee <\{E\#\} \{first(Department), sum(Salary)\} > \{“010684010186” “010190010191”\}$  gives

E#	Department	Salary	T
111	Toys	20K	01/01/85
111	Toys	25K	01/01/90
122	Sales	30K	01/01/90
133	Toys	32K	01/01/90

Figure 6:  $Employee_{enum1}$  relation.

E#	Department	Salary	T
111	Shoe	45K	[01/06/84,01/01/86)
111	Toys	25K	[01/01/90,01/01/91)
122	Sales	30K	[01/01/90,01/01/91)
133	Toys	72K	[01/01/90,01/01/91)

Figure 7:  $Employee_{enum2}$  relation.

the relation shown in Figure 7.

### 3 Implementation of the Temporal Relational DBMS

#### 3.1 Representation of the triplets

TDBMS has been implemented on top of ERAM [8], a memory resident database management system which is based on the extended relational model and extended relational algebra [17, 18]. The extended relational model allows relations with set-valued attributes where there is only one level of nesting. It has specifically been formulated for statistical databases, and hence includes powerful aggregation features. The extended relational algebra includes the five basic operations of relational algebra with extensions for handling set-valued relations, and the **pack** and **unpack** restructuring operations.

ERAM has been implemented in the C-programming language on top of the UNIX internal file structure and Unix standard I/O library. It is an operational system and has successfully been used and tested in several universities over the past years. ERAM provides a natural environment for the implementation of TDBMS since it supports (one level) nested relations. It directly supports atomic and set-valued attributes of valid time relations. Additionally, the **pack** and **unpack** operations in ERAM help in making and breaking set-triplet valued attributes of a valid time relation. Moreover, with a coding method triplet-valued and set-triplet valued attributes of a valid time relation can be translated to the corresponding attributes of ERAM. Two alternatives are considered:



1. The first alternative is to convert triplet-valued attributes into *three* separate *atomic* attributes such that  $\langle [l, u), v \rangle$  becomes  $l$ ,  $u$  and  $v$ . For the set-triplet valued attributes, we create three set-valued attributes. As an example, consider the following set of triplets:

$$\begin{array}{c}
 \{ \langle 1, 2, a \rangle, \langle 3, 4, b \rangle, \langle 5, 6, c \rangle \} \\
 \downarrow \\
 \begin{array}{ccc}
 \text{*L} & \text{*U} & \text{*V} \\
 \hline
 \{1, 3, 5\} & \{2, 4, 6\} & \{a, b, c\}
 \end{array}
 \end{array}$$

The lower bound values ( $l$ ) of each triplet in the set-triplet valued attribute are combined into a single set-valued attribute i.e.,  $*L$ . The same is the case for the upper bounds i.e.,  $*U$  and the value i.e.,  $*V$  fields of triplets. Naturally, each of the set-valued attributes has components which have the same type which is also required by ERAM. However, the problem in this alternative is to maintain the order of the components in each set-valued attribute. Otherwise it would be impossible to reconstruct the original triplets from these three separate attributes. Furthermore, duplicates are not allowed in set-valued attributes. Duplicate values may appear when a set-triplet valued attribute is broken into its components. This necessitates conversion of duplicates into unique values so that they can be safely manipulated by ERAM. However, this would lead to considerable computational complexity in manipulating and updating the prefixes in each relational operation.

2. In the second alternative, a triplet is coded into a single value. That is,  $\langle [l, u), v \rangle$  becomes  $luv$ , a single string, hence, a triplet-valued attribute becomes an atomic attribute in ERAM. Similarly, a set-triplet valued attribute is converted to a set-valued attribute. Thus, the triplets of the preceding example become 12a, 34b, 56c, and hence the second method, converting triplets into strings is preferred and is used in the implementation.

### 3.2 Data Formats for Time

We use the following data structure in representing a triplet:

lower bound (l)	upper bound (u)	value (v)
-----------------	-----------------	-----------

where the lower and upper bounds are represented as time points. A time point is represented as *ddmmyy*

where *dd* (5 bits) refers to the day, *mm* (4 bits) refers to the month, and *yy* (7 bits) stands for the year. Thus, the system can accommodate 128 years where time granularity is a day. Hence, the lower and upper bounds are compressed into a total of four bytes. The value of a triplet is concatenated to these four bytes to form the final string. When needed, they are uncompressed to their original form, in presenting the results and in implementing TRA operations. To keep the present implementation task simpler we do not consider time points beyond the current time, *now*.

### 3.3 Implementation of the Relational Commands

We have modified ERAM to accommodate TRA operations. We also have added code for the new operations of TRA. Details of the commands and their syntax are given in [10]. Triplets are written into relations in compressed form. Most of the TRA operations require decompression of these values into their components, i.e., the selection operation or the unpack operation which requires a sorting operation to eliminate the duplicate tuples in the result.

We briefly comment on the implementation methodology used for the *unpack*, *join* and *enumeration* operations since they are frequently used in the sample queries. For the *unpack* operation, a sequential scan is made of the operand relation and each tuple of the original relation is written to a temporary file as many times as the number of instances of the specified attribute. The temporary file is then sorted to eliminate duplicates before loading the tuples back to the relation. *Unpack* operation does not create duplicates if one or more atomic attributes of the operand relation form a key. To accommodate this case, we implemented another version of the *unpack* operation without sorting. In the *join* operation, only the tuple identifiers and the join attributes of the operand relations are written to temporary files. These files are then sorted to eliminate duplicates after which a sequential scan is made to get the matching tuples on the join attributes. The tuple identifiers are used to fetch the qualifying tuples of the original relations and load them into the new relation. The **enum1** operation returns the values of designated attributes at specified time points in one sequential scan over the operand relation. The triplets of the designated attributes are examined in turn for intersection with the specified time point. If an intersection is found, the data values of the attributes are selected.

### 3.4 Example Queries

In this section we give example queries and their temporal algebra expressions, **TAE**. The queries use the *Employee* relation of Figure 1 which is created in TDBMS. We also provide the results produced by the system in Figure 8.

**Query 1:** What are the E#s of employees currently making more than 30K?

**TAE:** select E# from (**enum1** Employee<{E#} {Salary}>{now}) where Salary > 30

The **enum1** operation selects the salary values at the current time, *now*. The *select* operation then selects the E#s of employees whose salary is more than 30K. Note that an *unpack* operation on the *salary* attribute, followed by a selection is an alternative way to obtain the same result. However, for large relations, the **enum1** operation is more efficient.

**Query 2:** What are the E#s and department history of employees who make more than 30K in their salary history?

**TAE:** select E# Department from (unpack Employee on Salary) where SalaryV > 30

In this query, the entire history of salary values is searched for a match. As opposed to Query 1, we have to use the *unpack* operation here. After the *unpack*, the *value* field of each salary triplet (salaryV) is checked for a match.

**Query 3:** What are the E#s and current department of employees who make between 25K and 30K in their salary history?

**TAE:** **enum1** (select E# Department from (unpack Employee on Salary) where SalaryV ≥ 25 and SalaryV ≤ 30)<{E#} {Department}>{now}

This query is similar to Query 2. It tests a condition in the history of salary values and retrieves the current department values of the qualifying tuples.

**Query 4:** What were the E#s and salaries of employees when they worked for the Toys department?

**TAE:** select E# Salary from (slice (unpack (unpack Employee on Department) on Salary) Salary by Department) where DepartmentV = “Toys”

After the *department* and *salary* attributes are unpacked, the time component of the *salary* attribute is aligned with respect to that of the *department* attribute. The E#s and salary values of the employees who worked in the Toys department are then selected.

**Query 5:** What is the average salary each employee earned?

**TAE:** (unpack Employee on Salary)<E# avg(Salary)>

E#
133

(Q1)

E#	*\$Department
133	{<[Jan 89, now], Toys>}

(Q2)

E#	Department
111	Toys
122	Sales

(Q3)

E#	\$Salary
111	<[June 85, now], 25K>
133	<[Jan 89, June 90), 32K>
133	<[June 90, now], 40K>

(Q4)

E#	Salary
111	22.50
122	30.00
133	36.00

(Q5)

Figure 8: Results of the Example Queries

## 4 Experiments with the Temporal Relational DBMS

### 4.1 Overview

In this section we show how the TDBMS can be used for measuring the performance of queries on temporal databases which support tuple timestamping and those that support attribute timestamping. A valid time relation consisting of only atomic attributes is a regular 1NF relation. In this relation, certain atomic attributes can be designated for the time reference of tuples. Such a relation in TDBMS simulates tuple timestamping. Hence, our implementation supports both tuple timestamping and attribute timestamping and provides a natural environment for evaluating their performance. In this respect queries of both database types (tuple timestamping and attribute timestamping) have the same overhead due to the structure of TDBMS. However, the system will be slightly biased towards attribute timestamping since set-valued attributes and some compression are not required in the implementation of tuple timestamping.

Our aim is to measure the performance of different types of temporal databases using various queries in terms of processing time. We have chosen a set of sample queries with varying characteristics such as, comparison between atomic attributes, atomic and triplet-valued attributes and triplet-valued and triplet-valued (time-varying) attributes. A list of these queries is provided in Appendix 2. This list covers many of the sample queries included in [12], except those that are defined for testing the user friendliness of temporal query languages. A detailed listing of the algebraic expressions, expressed in TDBMS for these queries can be found in [11]. We have executed these queries against the following database types:

- **AT:** A temporal database using attribute timestamping.
- **TTL:** A temporal database using tuple timestamping where a single relation is used.

- **TT**: A temporal database using tuple timestamping where relations are in temporal normal form.

Note that we consider two cases for tuple timestamping. This will enable us to see how the performance of temporal databases using tuple timestamping changes depending on whether a single relation is used to hold all the temporal data belonging to similar objects (**TTL**), or several relations are used to hold the same temporal data (**TT**). To ensure consistency, **AT**, **TTL**, and **TT** databases contain the same data.

## 4.2 Generating the AT Data

### 4.2.1 Database Schema

A test database, *eval* has been created for the evaluation of the database types mentioned in the previous section. It consists of four relations, namely, *Emp*, *Dept*, *Proj* and *Assigned*. These relations are:

- (a) **Emp**(ssno, name, address, \*\$salary, \*\$skills, \*\$dname)
- (b) **Dept**(dno, dname, \*\$budget, \*\$manager)
- (c) **Proj**(pno, pname, \*\$budget)
- (d) **Assigned**(ssno, pno, \*\$type of work, \*\$rating)

The *Assigned* relation shows employees and the projects they are assigned to. Attribute *type of work* denotes the different types of work carried out by the employee. For each assignment, the employee gets a rating, say, 0–100 for different periods of time. The rest of the relations and their attributes are self-explanatory. This database is created as a **AT** database under TDBMS. Temporal databases using tuple timestamping are generated from it as described in Sections 4.3 and 4.4.

### 4.2.2 Populating the Database

Each of the relations mentioned in the previous section is populated with data generated randomly from a uniform distribution. The random function, *drand48()* which returns non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0), is used to generate random data. The random number is then converted to an attribute value. The range of the attribute values is listed in Table 1. It should be noted that for each of the set-valued and set-triplet valued attributes in TDBMS, two bytes are allocated in each tuple. These hold pointers to a bucket of set values. For convenience, two assumptions have been made while creating the historical data:

Attribute	Relation	Value Range	Time Range	Size (bytes)
ssno	Emp	1–500	–	2
name	Emp	E1–E500	–	4
address	Emp	A1–A500	–	4
salary	Emp	50K–120K	01/01/70– <i>now</i>	6
skills	Emp	S1–S10	01/01/70– <i>now</i>	7
dname	Emp	D1–D20	01/01/70– <i>now</i>	7
dno	Dept	1–20	–	2
dname	Dept	D1–D20	–	3
budget	Dept	5000K–6000K	01/01/70– <i>now</i>	6
manager	Dept	M1–M20	01/01/70– <i>now</i>	7
pno	Proj	1–50	–	2
pname	Proj	P1–P50	–	3
budget	Proj	5000K–6000K	01/01/70– <i>now</i>	6
ssno	Assigned	1–500	–	2
pno	Assigned	1–50	–	2
type	Assigned	W1–W10	01/01/70– <i>now</i>	7
rating	Assigned	0–100	01/01/70– <i>now</i>	6

Table 1: Attribute Ranges.

1. All the tuples of a relation are homogeneous, i.e., each set-triplet valued attribute of a tuple has the same starting and ending times. Note that TDBMS supports non-homogeneous tuples as well.
2. The instances of each set-triplet valued attribute are continuous, i.e., we assume no null values or discontinuities in the tuples of these relations.

### 4.3 Generating the TTL Data

The test database *eval* and the relations given in Section 4.2.1 are used to generate the relations which use tuple timestamping where the whole temporal data of a relation is contained within a single relation. These temporal relations are generated by first unpacking the relations on each set-triplet valued attribute and then intersection slicing each (triplet-valued) attribute by the other (triplet-valued) attribute. This will ensure that each attribute has the same time reference. A droptime operation is then carried out on each temporal attribute except one. Triplet-decomposition on this attribute gives the desired relation. For

example, the **Deptttl** is created using the following algebra expression:

$$\mathbf{Deptttl} := \text{tdec} (\text{droptime} (\text{slice} (\text{slice} (\text{unpack} (\text{unpack Dept on budget}) \text{ on manager}) \\ \text{budget by manager}) \text{ manager by budget}) \text{ on budget}) \text{ on manager}$$

For ease of reading, we rename the attributes  $A_v$ ,  $A_l$ , and  $A_u$  of the result of triplet-decomposition operation with the attribute names  $A$ ,  $from$  and  $to$  respectively. The resulting **TTL** relations are as follows:

- (a) **Empttl**(ssno, ename, address, salary, skills, dname, from, to)
- (b) **Deptttl**(dno, dname, budget, manager, from, to)
- (c) **Projttl**(pno, pname, budget, from, to)
- (d) **Assignedttl**(ssno, pno, type of work, rating, from, to)

#### 4.4 Generating the TT Data

The test database *eval* and the relations given in Section 4.2.1 are used to generate the relations which use tuple timestamping and are in temporal normal form. From each relation using attribute timestamping, a family of relations is created. The non-time varying attributes are collected into a relation whereas each time-varying attribute and the key are kept in a separate relation. These temporal relations are created by first projecting out the desired attributes from the original relations, unpacking the set-triplet valued attributes and then performing a triplet-decomposition operation. For example, the **Empsl** is created using the following algebra expression:

$$\mathbf{Empsl} := \text{tdec} (\text{unpack} (\text{project Emp on ssno salary}) \text{ on salary}) \text{ on salary}$$

We rename the names of attributes created as a result of triplet-decomposition operation similar to the **TTL case**. The resulting **TT** relations are as follows:

- (a) **Empa**(ssno, ename, address)
- (b) **Empsl**(ssno, salary, from, to)
- (c) **Empsk**(ssno, skills, from, to)
- (d) **Empd**(ssno, dname, from, to)
- (e) **Depta**(dno, dname)
- (f) **Deptb**(dno, budget, from, to)

- (g) **Deptm**(dno, manager, from, to)
- (h) **Proja**(pno, pname)
- (i) **Projb**(pno, budget, from, to)
- (j) **Assignedt**(ssno, pno, type of work, from, to)
- (k) **Assignedr**(ssno, pno, rating, from, to)

## 4.5 Queries

The list of example queries used for the performance evaluation is given in Appendix 2. Each query is executed against the database types **AT**, **TTL** and **TT**. We consider two types of queries: **AT<sub>C</sub>**, **TTL<sub>C</sub>** and **TT<sub>C</sub>** stand for queries involving only current data. On the other hand, **AT<sub>H</sub>**, **TTL<sub>H</sub>** and **TT<sub>H</sub>** are the queries which involve part or all of the historical data represented in a temporal database.

## 4.6 Database Parameters

Maintaining longer histories increases the database size. Similarly, the frequency of update and insertion operations also affect the database size. However, the performance degradation is obvious upon increasing the database size. Thus, we use the length of history kept as the determining factor of the database size and perform our experiments by changing the number of triplets in set-triplet valued attributes. To see the effect of the length of history on the performance of different databases, we consider two cases by changing the number of triplets in a set-triplet valued attribute. In case **A1**, a set-triplet valued attribute contains 5 triplets, whereas in case **A2**, it contains 10 triplets. Thus, case **A2** represents a database which carries a longer history and **A1** represents a relatively shorter history. If the length of history is the same, **A2** represents a database where there are frequent changes. Table 2 gives the relation sizes for these two cases. The cardinality of the Emptttl relation in case **A2** could not be calculated since the intermediate relations resulting from the unpack operations were too large.

Query processing times for parameter set **A1** and **A2** are given in Appendix 1. In the table of run **A1**, the second and third columns represents the results for the temporal databases using attribute timestamping. The second column (**AT<sub>C</sub>**) gives processing time of the queries involving current data. The third column (**AT<sub>H</sub>**) is the processing time for the historical queries. The next two columns show processing time of queries in the case of temporal databases using tuple timestamping (**TT<sub>C</sub>** and **TT<sub>H</sub>**) where relations are in temporal normal form. The last two columns show processing time of queries in the case of temporal databases using tuple timestamping (**TTL<sub>C</sub>** and **TTL<sub>H</sub>**) where a single temporal relation is used. We did not run the queries for the **TTL** type database for run **A2** since the resulting relations are



Relations	Case <b>A1</b>		Case <b>A2</b>	
	Tuple length	Cardinality	Tuple length	Cardinality
Emp	16	500	16	500
Dept	9	20	9	20
Proj	7	50	7	50
Assigned	8	2000	8	2000
Empttl	22	6487	22	–
Deptttl	14	180	14	380
Projttl	11	250	11	500
Assignedttl	13	17980	13	37947
Empa	10	500	10	500
Empsl	8	2500	8	5000
Empsk	9	2500	9	5000
Empd	9	2500	9	5000
Depta	5	20	5	20
Deptb	8	100	8	200
Deptm	9	100	9	200
Proja	5	50	5	50
Projb	8	250	8	500
Assignedt	11	10000	11	20000
Assignedr	10	10000	10	20000

Table 2: Relation sizes for Case **A1** and **A2**.

too large and obviously its performance would be much slower.

#### 4.7 The Computing Environment

All the experiments were carried out on a Sparc ELC workstation on a local area network. Some (marginal) interference is possible since the machine was connected to network (Ethernet). However, most of Ethernet work is done by the controller, not by the CPU. Hence the interference can be ignored. Moreover, since all three database types are subjected to the same (marginal) interference, and therefore, we do not expect that the interference would unfavourably affect a database type.

The processing times are given in terms of the time the process spent in *user* mode. Since we are

concerned primarily with retrieval type of queries, we do not consider the time spent in *system/kernel* mode.

## 4.8 Experiments and Results

All experiments are based on databases given in Section 4.6, and sample queries listed in Appendix 2. We conducted three experiments. In the first experiment, the sample queries were run against the sample **AT** database. In the second experiment the queries were run on the **TT** database and the third experiment was run on the **TTL** database. In the following interpretation, we mostly refer to the results of parameter set **A1**. The table for case **A2** gives the results for larger databases and confirms the results obtained in case **A1**.

**Q1.0**, **Q1.1** and **Q1.2** are point queries which select one tuple i.e., the salary of a particular employee. **AT** databases perform better than **TT** and **TTL** databases since the salary history of an employee is contained within a single tuple. On the other hand, **TT** and **TTL** databases have the salary history of an employee spread over multiple tuples requiring multiple accesses. Additionally, **TTL** databases perform slower than **TT** databases due to the longer tuple length and increased cardinalities of **TTL** relations.

**Q1.3** and **Q1.4** are also point queries selecting a tuple but retrieving two and three attributes, respectively. There is not much difference between these queries and **Q1.0**, **Q1.1** and **Q1.2** in the performance of **AT** databases since the multiple attributes are retrieved from a single qualifying tuple. Similarly, there is not much difference in the performance of **TTL** databases since the number of qualifying tuples does not vary significantly from those of **Q1.0**, **Q1.1** and **Q1.2**. The performance of **TT** databases however degrades due to one *join* operation for **Q1.3** and two *join* operations for **Q1.4**. The *join* operation is necessary since the attributes to be retrieved are distributed over multiple relations.

**Q1.5**, **Q1.6** and **Q1.7** are range queries which retrieve one set-triplet valued attribute, the salary of employees. **Q1.5** retrieves employees whose names are less than E144. This selects 10% of the employee relation, i.e., 50 tuples since employee names are character strings and string comparison is used in the query. More specifically, employees with names E1, E10-E14, and E100-E143 are selected (Employee names range from E1 to E500 as shown in Table 1). **Q1.6** selects 20% of the employee relation, i.e., 100 tuples and **Q1.7** selects all the salary values in the employee relation, i.e., 500 tuples. The performance of all three types of databases degrades from **Q1.5** to **Q1.6** due to 50 more tuples being selected. **TT** databases do not perform as well as **AT** databases due to the *join* operation. The time required for **AT** databases almost doubles from **Q1.6** to **Q1.7**. This is due to the entire salary values being retrieved and the length

of the salary attribute. In contrast, the time required for **TT** databases decreases from **Q1.6** to **Q1.7**. Although all tuples are selected, the absence of a join improves the performance. The **TT<sub>H</sub>** version of **Q1.7** is a simple projection. In the **TTL** databases case, the time for **TTL<sub>C</sub>** increases from **Q1.6** to **Q1.7** due to more tuples being selected in **Q1.7**. However, **TTL<sub>H</sub>** is a simple projection, hence the time decreases from **Q1.6** to **Q1.7**.

**Q1.8**, **Q1.9** and **Q1.10** are range queries which retrieve two set-triplet valued attributes, the salary and departments of employees. The number of tuples selected are identical to **Q1.5**, **Q1.6** and **Q1.7** respectively. The response times in these queries are higher than their **Q1.5**, **Q1.6** and **Q1.7** counterparts due to the retrieval of an additional set-triplet valued attribute. Again, **TT** databases do not perform as well as **AT** databases due to the *join* operation. Two join operations are required in **Q1.8** and **Q1.9** as compared to a single join in queries **Q1.5** and **Q1.6**. Even though a single join is required for the **TT<sub>H</sub>** version of **Q1.10**, the time spent in processing the query is quite high due to the join being on the entire **Empsl** and **Empd** relations. The explanation for the performance of **TTL** databases is similar to the one given for queries **Q1.5**, **Q1.6** and **Q1.7**.

**Q1.11**, **Q1.12** and **Q1.13** are range queries which retrieve three set-triplet valued attributes, the salary, departments and skills of employees. The number of tuples selected are identical to **Q1.5**, **Q1.6** and **Q1.7** respectively. As expected, the times for these queries are higher than **Q1.8**, **Q1.9** and **Q1.10** due to the retrieval of an additional set-triplet valued attribute. Other trends are similar to those noticed for queries **Q1.8**, **Q1.9** and **Q1.10**. Three join operations are required in **Q1.11** and **Q1.12**. The performance degradation in the **TT<sub>H</sub>** version of **Q1.13** is quite striking due to the joins between the entire **Empsl**, **Empd** and **Empsk** relations.

**Q1.14** selects the current department of the employees who satisfy a condition in the past (i.e., salary = 60K on 01/01/72). The time required in processing the **TT** queries is higher than the **AT** queries due to the requirement of a join operation in **TT**. **TTL** queries take longer to process due to longer tuples and larger cardinality of selected tuples.

**Q2.0** selects ssno's of employees whose salary is greater than 60K. **Q2.1** is similar to **Q2.0** except the skills of employees are retrieved instead of ssno's. The processing time of **AT** and **TT** databases increases from **Q2.0** to **Q2.1** due to the retrieval of a set-triplet valued attribute in the latter query. The processing time of **TTL** however is almost constant due to retrieval of an atomic attribute in both **Q2.0** and **Q2.1**. Processing the **AT<sub>H</sub>** query takes significantly longer than the **TT<sub>H</sub>** query. **AT<sub>H</sub>** query requires an *unpack* operation for reaching the historical values. The *unpack* operation has heavy I/O overhead since it reads/writes entire historical tuples which are longer into temporary relations. Elimination of duplicates

also requires sorting which is very costly. **Q2.1** retrieves skills of employees. It requires a *join* operation in processing **TT** queries, thus increasing their processing time. However, **TT** database still performs much better than the **AT** database.

**Q3.0** is a range query which returns ssno's of employees whose salary is between 50K and 70K. **Q3.1** is similar but retrieves skills of employees. The performance trends in **Q3.0** and **Q3.1** are similar to **Q2.0** and **Q2.1** respectively.

**Q4.0** is a join query. The historical versions of all types of databases perform better than the current versions due to the extra selection operation required in the current queries. All types of databases require a join, but the length and cardinality of selected tuples is greater in **TTL**, followed by **TT**, and finally **AT** databases. This is substantiated by the increase in processing time from **AT** to **TT** to **TTL** as seen in Table 5. **Q5.0** also shows a similar performance pattern.

**Q6** and **Q7** are join queries. The first one involves a salary value and the second one involves a range of salary values. The explanation for the increase in processing time from **AT** to **TTL** databases in **Q6** is similar to that of **Q4.0**. The performance of **AT** databases degrades in **Q7** due to the unpack operation.

**Q8** involves a *join* operation between two time-varying attributes. Processing times are more or less the same for **AT** and **TT** databases. The processing time for **TTL<sub>H</sub>** query in both **Q8** and **Q8.1** is very high since the entire **Deptttl** and **Projttl** relations are joined. The number of tuples participating in the join and their length is quite large, hence the increasing the processing time.

**Q9** is another join query. Processing times of **AT<sub>C</sub>** and **TT<sub>C</sub>** queries are not significantly different. Although an *unpack* operation is needed in the **AT<sub>H</sub>** query, since only a single tuple is unpacked, the processing time does not increase much.

Queries 11 through 15 are aggregation queries. In larger databases we were not able to complete **Q11** for **AT<sub>H</sub>** since the *unpack* operation created too many intermediate tuples, beyond the capacity of the system. Furthermore, the *unpack* operation required for the **AT<sub>H</sub>** queries increased the processing time significantly in comparison to the other database types. Overall, in queries **Q1.0** to **Q9**, **AT** databases had better performance in 76% of the current queries whereas **TT** databases performed better in 24% of the current queries. In historical queries, **TTL** databases performed better only in 27% of the queries when compared to **AT** databases.

## 5 Conclusions and Future Work

In this work, we described the implementation of a temporal relational database management system (TDBMS) on top of an existing relational database management system (ERAM). ERAM provided a natural environment for the implementation of the methodology introduced in [28] for incorporating the time dimension into the relational model. The implemented system attaches time to attribute values, supports a time granularity of days and allows different types of attributes to coexist in relations: atomic values, triplets and their sets, and handles historical data in a convenient way. TDBMS includes revised versions of the basic set of operations of the extended relational algebra and new operations which convert one attribute type to another and do selection over the time dimension. A statistical interface has also been added to TDBMS. This interface includes aggregate functions and transformations of data into tabular forms suitable for advanced statistical analysis. The *enumeration* operation which derives a table of uniform data for a set of specified time points or intervals from a valid time relation is also included in the interface.

We then used TDBMS to generate two types of temporal databases that use tuple timestamping. The first database contains relations which are in the temporal normal form of [16] while the second database contains a single relation is used to hold all its time varying attributes. Sample queries were run on these three database types and the processing time of each query was measured. We can derive several useful conclusions from the experiments of Section 4:

1. As one would expect, large volume of data in temporal databases increase the processing time of temporal queries. However, as the size of the database decreases from parameter set **A2** to **A1** and shorter history is maintained in the database the performance of temporal databases would converge to the performance of snapshot databases.
2. In case of point queries selecting one attribute value of fewer tuples, the **AT** database performs better than the **TT** database. **AT<sub>C</sub>** queries executing on a single relation take less processing time than their **TT<sub>C</sub>** counterparts since **AT** databases contain fewer tuples than **TT** databases. Additionally, in **AT** databases, the set-membership operator is used to select the desired triplet(s) from the history of a set-triplet valued attribute. Moreover, **TT** database requires *join* operations when several temporal attributes are requested by the query. It can be seen that **AT<sub>C</sub>** queries involving more than one temporal attribute require less processing time. In the **TT<sub>C</sub>** queries however, the *join* operation increases the processing time. Similarly, **AT<sub>H</sub>** queries involving more than one temporal attribute require less processing time since a simple *select* operation is used to fetch the entire history values of the desired attribute. **TT<sub>H</sub>** queries on the other hand require a *join* operation to fetch the desired

values. Thus, as the number of qualifying tuples increase and one or more attributes are retrieved, **TT** queries become more expensive.

3. *Unpack* is a costly operation to reach the values of historical attributes. Though the *unpack* operation can be completed in one scan, it still needs sorting to eliminate duplicates. Furthermore, the intermediate tuples are written to temporary files. This creates an extra overhead for **AT** databases since the tuples are longer and require more I/O time. Queries requiring *unpack* operation degrades the performance of **AT** databases faster than the performance of **TT** queries requiring *join* operations. In TDBMS, the *join* operation is carried out in an efficient way. Optimization procedures for the *unpack* operation are also needed. Furthermore, an operation which allows access to the elements of a set-triplet valued attribute can be added to TRA. This operation will be used in most of the queries requiring an *unpack* on a set-triplet valued attribute. Such an operation is similar to **enum1** and can be processed by a sequential scan of the operand relation.
4. It was also seen that having all temporal attributes within a single relation (**TTL**) gives better performance than **TT** only in a few queries not involving joins of temporal attributes. In general, **TTL** databases require more processing time than **AT** and **TT** databases due to the large size of **TTL** relations. For joins involving temporal attributes, the performance of **TTL** database degrades substantially as seen by queries **Q8.0** and **Q8.1**. Therefore, for tuple timestamping **TTL** is not a better choice than **TT** relations.
5. As the size of the temporal database increases, we observe degradation in the performance of **AT** and **TT** databases. The length of history kept in the database as well as the frequency of changes in attribute values directly affects the size of the database. However, magnitude of the performance degradation is more than the proportional change in the database size. Furthermore, processing time of queries for **AT** and **TT** databases in general are stable from one query to another with occasional jumps. These jumps occur in queries requiring an *unpack* operation in **AT** databases whereas they occur in queries requiring excessive *join* operations in **TT** databases.

The TDBMS prototype is the first implementation where attribute timestamping is used. It will be useful in testing the theory developed for temporal databases. It will also form the foundation for future implementation of temporal query languages, temporal query optimization, and performance evaluation of alternative temporal models.

Our study verifies that the major performance trade off between different types of temporal databases

is between the restructuring (*unpack*) operation needed in temporal databases using attribute timestamping and the *join* operation needed in temporal databases using tuple timestamping. Furthermore, the experiments show that keeping all temporal tuples in one single relation does not prove to be an effective alternative for temporal databases which use tuple timestamping.

Performance evaluation of temporal DBMSs, especially of temporal query processing has received marginal interest in the research community. The results of our experiments emphasize the need of a detailed performance study of temporal databases using attribute timestamping and those using tuple timestamping. The work of Ahn and Snodgrass [2, 3] is a good starting point if mirrored to temporal databases using attribute timestamping. In [2], Ahn and Snodgrass, run a benchmark set of queries to study the performance of their prototype system on four types of databases: static, rollback, historical and temporal. Furthermore, they propose an analytical model which analyses the input and output cost of temporal queries on various access methods [3]. We follow an empirical approach and measure the actual performance of representative sample of temporal queries on temporal databases supporting attribute timestamping and those supporting tuple timestamping. The results that we report in the previous section provide useful insight into the performance of temporal databases using tuple timestamping and attribute timestamping.

The absence of a time index and appropriate access paths also account for the performance variation in our experimental study. An actual performance study of different types of temporal databases would entail using time indexes to better optimize the *unpack*, *join* and other temporal operations. Quite a number of studies have been done on the development of time indexes for temporal databases [6, 7, 21, 22]. However, all of them assume an underlying temporal model based on tuple timestamping. It is clear that further research should also be carried out for the addition of a time index to improve processing of certain class of temporal queries in a temporal model using attribute timestamping. Once this is done, a comprehensive performance study on the different approaches of modeling temporal databases can be carried out.

In this work, we obtained attribute values from uniform distributions. Our results can be verified by analytic cost estimations. However, in real life attribute distributions are not uniform, on the contrary most of the time they are skewed. We focussed our attention on retrieval queries and we have not considered update operations. We do not expect much of a difference for simple update operations requiring few tuples. However, certain update patterns as well as update operations affecting larger number of tuples may cause a performance difference in **AT** and **TT** databases. We plan to consider these issues, possible optimization of TRA operations and their effect on the performance of **AT** and **TT** databases in a separate work. Other database functions such as concurrency control, crash recovery, etc., are also among the

topics that could be incorporated to the prototype implemented. Problems created by temporal databases in handling these functions also deserves further investigation.

## Acknowledgement

We would like thank Professors Gultekin Ozsoyoglu and Meral Ozsoyoglu for permitting us to use the ERAM database package. The work of Abdullah Uz Tansel is partially supported by the PSC/CUNY research grant 665307. The work of Iqbal A. Goralwalla and M. Tamer Özsu has been supported by the Natural Sciences and Engineering Research Council of Canada under research grant OGP0951.

## References

- [1] Ariav, G., "A Temporally Oriented Data Model", ACM TODS, Vol.11, No.4, 1986.
- [2] Ahn, I., Snodgrass, R., "Partitioned Storage for Temporal Databases", Information Systems, Vol.13, No.4, pp 369-391, 1988.
- [3] Ahn, I., Snodgrass, R., "Performance Analysis of Temporal Queries", Information Sciences, Vol.49, 1989, pp 103-143.
- [4] Clifford, J., Croker, A., "HRDM: A Historical Relational Data Model", Proc. of the Third IEEE International Conf. on Data Engineering, 1987.
- [5] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", Comm. of ACM. Vol.13 No.6 (1970) pp 377-387.
- [6] Elmasri, R., Wu, G.T.J., Kim, Y., "The Time Index - An Access Structure for Temporal Data," in Proc. of the 16th VLDB Conference, Brisbane, Australia, August 1990.
- [7] Koramajiam, V., Kamel, I., Elmasri, R., Waheed, S., "The Time Index+: An Incremental Access Structure for Temporal Databases," in Proc. of the CIKM Conference, 1994.
- [8] Hou, Wen-chi., "The Implementation of the Extended Relational Database Management System", MS thesis, Case Western Reserve University, 1985.
- [9] Gadia, S.K., "A Homogeneous Relational Model and Query Languages for Temporal Databases", ACM TODS, Vol.13, No.4, 1988.
- [10] Goralwalla, I.A., "An Implementation of a Temporal Relational Database Management System", MS thesis, Bilkent University, June 1992.
- [11] Goralwalla, I.A., Tansel, A., Özsu, M.T., "An Empirical Study of the Performance of Temporal Relational Databases," Technical Report TR94-15, Department of Computing Science, University of Alberta, November 1994.
- [12] Jensen, C.S., et al., "A Consensus Test Suite of Temporal Database Queries", Aalborg University, Technical Report R93-2034, November 1993.
- [13] Kline, N., "An Update of the Temporal Database Bibliography," ACM SIGMOD Record, Vol.22, No.4, pp. 66-80, December 1993.



- [14] Lorentzos, N.A., Johnson, R.G., “Extending Relational Algebra to Manipulate Temporal Data”, Information Systems, Vol.15, No.3, 1988.
- [15] McKenzie, M., “Bibliography: Temporal Databases,” ACM SIGMOD Record, Vol.15, No.4, pp. 40-52, December 1986.
- [16] Navathe, S.B., Ahmed, R., “A Temporal Relational Model and A Query Language,” Information Sciences, Vol.49, No.2, pp. 147-175, 1989.
- [17] Ozsoyoglu, Z.M., Ozsoyoglu, G., “An Extension of Relational Algebra for Summary Tables”, Proc. Second LBL Workshop on Statistical Database Management, 1983.
- [18] Ozsoyoglu, G., Ozsoyoglu, Z.M., Matos, V., “Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions”, ACM Trans. Database Syst., Vol.12, No.4, Dec. 1987, pp 566-592.
- [19] Pissinou, N., et al., “Towards an Infrastructure for Temporal Databases: Report of an Invitational ARPA/NSF Workshop,” ACM SIGMOD Record, Vol.23, No.1, pp. 35-51, March 1994.
- [20] Sarda, N., “Extensions to SQL for Historical Databases”, IEEE Trans. on Knowledge and Data Engineering, Vol.2, No.2, June 1990, pp 220-230.
- [21] Segev, A., Gunadhi, H., “Efficient Indexing Methods for Temporal Relations,” IEEE Transactions on Knowledge and Data Engineering, Vol.5 , No.3, June 1993.
- [22] Shen, H., Ooi, B.C., Lu, H., “The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases,” in Proc. of the International Conference on Data Engineering, 1994.
- [23] Snodgrass, R., “The Temporal Query Language, TQuel”, ACM Trans. Database Syst., Vol.12, No.2 pp.247-298, June 1987.
- [24] Snodgrass, R. (Ed.), “Research Concerning Time in Databases: Project Summaries”, ACM SIGMOD Record, Vol.15, No.4, December 1986.
- [25] Snodgrass, R., “Temporal Databases: Status and Research Directions”, ACM SIGMOD Record, Vol.19, No.4, pp 83-89, December 1990.
- [26] Soo, M.D., “Bibliography on Temporal Databases”, ACM SIGMOD Record, Vol.20, No.1, pp. 14-23, March 1991.
- [27] Stam, R., Snodgrass, R.T., “A Bibliography on Temporal Databases,” IEEE Database Engineering, Vol.7, No.4, pp. 231-239, December 1988.
- [28] Tansel, A.U., “Adding Time Dimension to Relational Model and Extending Relational Algebra”, Information Systems Vol.13 No.4 (1986) pp 343-355.
- [29] Tansel, A.U., “A Statistical Interface for Historical Relational Databases”, Proc. of the Third IEEE International Conf. on Data Engineering. 1987, pp 538-546.
- [30] Tansel, A., et al., “Temporal Databases: Theory, Design, and Implementation”, Benjamin/Cummings, 1993.

## Appendix 1: Processing times (in seconds) for runs A1 and A2

Q	AT <sub>C</sub>	AT <sub>H</sub>	TT <sub>C</sub>	TT <sub>H</sub>	TTL <sub>C</sub>	TTL <sub>H</sub>
1.0	0.5	0.47	1.08	1.01	3.19	3.20
1.1	0.51	0.49	1.02	1.01	3.21	3.19
1.2	0.51	0.55	1.04	0.99	3.16	3.13
1.3	0.51	0.49	2.10	1.93	3.37	3.25
1.4	0.52	0.57	3.03	3.08	3.18	3.15
1.5	0.53	0.56	1.57	0.74	3.52	3.36
1.6	0.59	0.65	1.61	0.88	3.53	3.67
1.7	1.05	1.07	1.29	0.29	3.84	1.23
1.8	0.61	0.66	2.76	1.75	3.44	3.41
1.9	0.75	0.83	2.81	2.40	3.56	3.47
1.10	1.57	1.91	2.50	5.90	3.84	1.36
1.11	0.68	0.77	4.30	5.78	3.45	3.35
1.12	0.85	1.00	4.01	10.31	3.44	3.50
1.13	2.09	2.64	3.93	38.17	3.95	1.98
1.14	1.22	1.38	2.50	2.27	4.19	3.69
2.0	1.40	12.37	1.40	1.23	3.94	3.63
2.1	3.00	14.32	2.41	2.33	3.83	3.55
3.0	1.39	12.81	1.34	1.26	3.98	3.87
3.1	2.51	14.17	1.97	2.24	4.16	3.87
4.0	4.09	1.14	5.16	2.53	9.36	5.13
5.0	N/A	0.98	N/A	2.04	N/A	4.07
6.0	4.91	2.57	6.06	3.70	12.67	8.07
7.0	6.41	18.19	6.31	5.96	13.46	13.04
8.0	0.19	0.13	0.21	0.17	0.26	70.06
8.1	0.22	0.11	0.24	0.21	0.28	72.77
9.0	1.13	0.48	1.10	1.55	3.48	3.78
10.0	1.68	1.13	2.26	1.59	3.21	3.71
11.0	1.65	–	2.79	6.72	3.79	1.67
12.0	N/A	0.87	N/A	1.69	N/A	3.05
13.0	0.09	0.33	0.13	0.06	0.14	0.06
14.0	4.38	26.76	5.76	2.24	10.54	4.30
15.0	N/A	11.14	N/A	0.51	N/A	1.71

**A1**

Q	AT <sub>C</sub>	AT <sub>H</sub>	TT <sub>C</sub>	TT <sub>H</sub>
1.0	0.74	0.76	2.03	1.97
1.1	0.8	0.82	1.98	1.95
1.2	0.84	0.84	2.01	2.01
1.3	0.73	0.70	4.00	4.01
1.4	0.83	0.74	5.87	6.12
1.5	0.94	0.98	2.70	1.21
1.6	0.98	1.10	2.69	1.31
1.7	1.79	2.09	2.52	0.54
1.8	0.99	1.06	5.28	3.97
1.9	1.17	1.38	5.17	6.09
1.10	2.83	3.73	4.90	22.37
1.11	1.05	1.26	7.65	31.62
1.12	1.33	1.78	7.37	59.52
1.13	3.75	5.13	7.31	266.79
1.14	2.05	2.59	4.58	4.39
2.0	2.06	42.78	2.67	2.46
2.1	5.39	49.97	4.71	4.67
3.0	2.03	43.41	2.41	2.44
3.1	3.81	46.37	3.09	4.34
4.0	6.55	1.91	10.21	6.18
5.0	N/A	1.74	N/A	4.23
6.0	8.23	4.86	12.53	7.39
7.0	9.65	53.14	12.58	11.83
8.0	0.31	0.18	0.37	0.30
8.1	0.40	0.21	0.44	0.39
9.0	1.80	0.49	2.06	3.99
10.0	2.88	2.12	4.42	3.46
11.0	2.85	–	5.40	23.77
12.0	N/A	1.42	N/A	3.11
13.0	0.12	1.00	0.16	0.06
14.0	6.95	89.26	10.77	4.32
15.0	N/A	39.61	N/A	1.03

**A2**

## Appendix 2: Sample Queries

In the following we give six versions of a query for the query **Q1.0**, current and historical for **AT**, **TT** and **TTL** databases. The other queries follow a similar format. To save space, we only list current and historical versions in a generic format.

### **Q1.0**      **Point Query**

**AT<sub>C</sub>**    : What is the current salary of the employee with a ssno of 1?

**AT<sub>H</sub>**    : What is the salary history of the employee with a ssno of 1?

**TT<sub>C</sub>**    : What is the current salary of the employee with a ssno of 1?

**TT<sub>H</sub>**    : What is the salary history of the employee with a ssno of 1?

**TTL<sub>C</sub>** : What is the current salary of the employee with a ssno of 1?

**TTL<sub>H</sub>** : What is the salary history of the employee with a ssno of 1?

### **Q1.1**      **Point Query**

Same as **Q1.0**, except the ssno of the employee is 250.

### **Q1.2**      **Point Query**

Same as **Q1.0**, except the ssno of the employee is 500.

### **Q1.3**      **Point Query**

**C**      : What are the current salary and manager values of the employee with a ssno of 250?

**H**      : What is the salary and manager history of the employee with a ssno of 250?

### **Q1.4**      **Point Query**

**C**      : What are the current salary, manager and skills values of the employee with a ssno of 250?

**H**      : What is the salary, manager and skills history of the employee with a ssno of 250?

### **Q1.5**      **Range Query**

**C**      : What are the current salaries of employees whose names are less than E144?

**H**      : What are the salary histories of employees whose names are less than E144?

### **Q1.6**      **Range Query**

**C**      : What are the current salaries of employees whose names are less than E199?

**H**      : What are the salary histories of employees whose names are less than E199?

### **Q1.7**      **Range Query**

**C**      : What are the current salary values of employees?

**H**      : What are the salary histories of employees?

### **Q1.8**      **Range Query**

**C**      : What are the current salaries and departments of employees whose names are less than E144?

**H**      : What are the salary and department histories of employees whose names are less than E144?

### **Q1.9**      **Range Query**

**C**      : What are the current salaries and departments of employees whose names are less than E199?

**H**      : What are the salary and department histories of employees whose names are less than E199?

**Q1.10 Range Query**

- C** : What are the current salary and department values of employees?
- H** : What are the salary and department histories of employees?

**Q1.11 Range Query**

- C** : What are the current salaries, departments and skills of employees whose names are less than E144?
- H** : What are the salary, department and skill histories of employees whose names are less than E144?

**Q1.12 Range Query**

- C** : What are the current salaries, departments and skills of employees whose names are less than E199?
- H** : What are the salary, department and skill histories of employees whose names are less than E199?

**Q1.13 Range Query**

- C** : What are the current salary, department and skill values of employees?
- H** : What are the salary, department and skill histories of employees?

**Q1.14 Point Query**

- C** : What is the current department of the employee whose salary was 60K on 01/01/72?
- H** : What is the current department of the employee whose salary was 60K at any time?

**Q2.0 Range Query**

- C** : What are the ssnos' of employees currently making more than 60K?
- H** : What are the ssnos' of employees who make more than 60K in their salary history?

**Q2.1 Range Query**

- C** : What are the skills of employees currently making more than 60K?
- H** : What are the skills of employees who make more than 60K in their salary history?

**Q3.0 Range Query**

- C** : What are the ssnos' of employees currently making between 50K and 70K?
- H** : What are the ssnos' of employees who make between 50K and 70K in their salary history?

**Q3.1 Range Query**

- C** : What are the skills of employees currently making between 50K and 70K?
- H** : What are the skills of employees who make between 50K and 70K in their salary history?

**Q4 Join between two Atomic attributes**

- C** : What is the current budget of project number 7, and what are the ssnos' and current types of work of the employees assigned to it?  
**H** : What is the budget history of project number 7, and what are the ssnos' and histories of the types of work of the employees assigned to it?

**Q5 Join between two Atomic attributes**

- H** : What are the ssnos' of employees who worked in project number 1?

**Q6 Join between two Atomic attributes**

- C** : What is the current type of work done by employees whose current salary is 60K?  
**H** : What is the history of type of work done by employees who made 60K at any time?

**Q7 Join between two Atomic attributes**

- C** : What is the current type of work done by employees whose current salary is  $\geq$  60K?  
**H** : What is the history of type of work done by employees who made  $\geq$  60K at any time?

**Q8.0 Join between two set-triplet attributes**

- C** : What are the department and project numbers of the departments and projects whose current budget is the same?  
**H** : What are the department and project numbers of the departments and projects which have the same budget values at the same time?

**Q8.1 Join between two set-triplet attributes**

- C** : What are the department numbers, managers and project numbers of the departments and projects whose current budget is the same?  
**H** : What are the department numbers, managers and project numbers of the departments and projects which have the same budget values at the same time?

**Q9 Join between two set-triplet attributes**

- C** : What is the current budget of the departments for which the employee whose ssno is 123 is currently working?  
**H** : What is the budget history of the departments for which the employee whose ssno is 123 worked at any time?

**Q10 Selection on a set-triplet attribute and retrieval from another set-triplet attribute**

- C** : What are the current salaries of employees currently working in department number 7?  
**H** : What were the salaries of employees when they were working in department number 7?

**Q11 Aggregation (1)**

**C** : What is the current average salary of employees currently working in each department?

**H** : What was the average salary of employees in each department?

**Q12 Aggregation (2)**

**H** : What are the number of projects each employee has been assigned to?

**Q13 Aggregation (3)**

**C** : Which department has currently the maximum budget?

**H** : Which department has the maximum budget?

**Q14 Aggregation (4)**

**C** : What was the current average rating of each employee for the projects he was assigned to?

**H** : What was the average rating of each employee for the projects he was assigned to?

**Q15 Aggregation (5)**

**H** : What was the highest salary earned by each employee?