Proceedings of International Conference
on Algorithms

Modification and Retrieval Algorithms for Bitemporal Databases*

To-yat Cheung
Department of Computer Science
City University of Hong Kong
Hong Kong

Xinming Ye
Department of Computer Science
Inner Mongolia University
Inner Mongolia, PRC

Abstract

Three system-level algorithms are proposed for time-related modification and retrieval in bitemporal databases. The algorithms adopt several strategies in their design, such as attribute-value timestamping, a special type of multi-lists for data storage and name-independency in operation calls. They also make use of two virtual data values: empty and null. The value empty makes single-valued representation of consecutive time-intervals feasible, whereas the value null provides a limited capability of 'resuming' a value processed previously. The algorithms are superior to some existing ones in both storage space and efficiency.

1. Introduction

In the past, research concerning time-related operations on temporal databases has mostly been at the user language level, such as TSQL or HSQ [10,12,14]. Each user-level operation, such as insertion, retrieval of old values, etc. , is mapped onto a sequence of system-level modifications and/or retrievals. At the system level, study has been confined mainly to data representations [2,5,6,9] and indexing methods [3]. Recently, the *algorithmic aspects* of system-level supports draws the attention of researchers [13].

This paper presents three system-level algorithms for supporting time-related operations on bitemporal databases, one for modification and two for retrieval. Examples of such operations include inserting a value between two previous time instants and retrieving an old value. Design of these algorithms is based on a multi-list model, two special values and several well-accepted database implementation strategies. In the following, we briefly describe the advantages of our strategies as compared with others.

Factors Affecting the Design of System-level Modification and Retrieval Algorithms

A. *Kinds of Time and Types of Temporal Databases Supported*

A *data object* d is a database constituent identifiable by a key (e.g., a tuple of a relational database). A *value* of d is a set of values of those *attributes* of d involved with time. The *valid-time* (*e-time*, also called *effective time*) e of a value v is the real-life time at which a data object obtains v. The *transaction-time* [4] (*t-time*, also called *processing time*) t of v is the computer-clock time at which v is processed by the TDBMS. The expression "data object d has value v *at* e-time e *as of* t-time t" means "according to the record of the TDBMS at computer time t, d possesses the value v at real-life time e". On the basis of valid-time and transaction-time, temporal databases can be classified into four types [6]: *snapshot, historical, rollback and bitemporal*, involving no time, only valid-time, only transaction-time and both valid-time and transaction-time, respectively. Obviously, a bitemporal database is most informative in comparison with the other types.

B. *Timestamps and Timestamping* [2,6,9]

One of the timestamp design issues is whether time-intervals should be consecutive or not. Using non-consecutive intervals has at least three disadvantages:

(1) It is less efficient, as the operations have to take care of the interval 'gaps'.
(2) Each interval has to be represented by a pair of time values, possibly duplicating some boundary values.
(3) Value inconsistency may arise in the common parts of overlapping intervals.

A *scheme for timestamping* describes how a timestamp is associated with data values. Two schemes are often used:

* This research was financially supported by the University Grant Council of Hong Kong

(1) *Attribute-value timestamping* [2], in which every key instance appears in one and only one tuple. The timestamps and relevant values are then associated with this tuple by some indexing techniques.

(2) *Tuple timestamping* [6,8], in which the same key is repeated in as many tuples as necessary. Obviously, attribute-value timestamping requires less storage space than tuple timestamping.

C. *Data Structure and Strategies for Handling Operation Names*

The methods of storing data and handling operation names directly affect the design and implementation of an algorithm [4,13]. Several strategies will be discussed below:

a. Splitting each relation into a core portion and a backlog:
   The core portion is for tuples accessed more frequently (e.g., current tuples) whereas the backlog is for tuples accessed less frequently (e.g., outdated tuples).

b. Name-driven versus time-driven operations:
   An operation may be called in two ways:

   1) *Name-driven:*
      The name is explicitly mentioned in the call and stored with the data. In particular, for a retrieval operation (e.g., rollback), past valus may have to be recreated by reversing a sequence of stored operations according to their names.
   2) *Time-driven*:
      Instead of operation's name, the relevant time values are specified in a call for determining the actual operation. The name-driven approach has several disadvantages: Firstly, for TDBM, since any modification always just adds new values to the database, its name is not essential. Secondly, besides wasting the extra space for storing the names, the data values become overly tied with the internal operations. Any alteration may affect the recoverability or validity of the data. Thirdly, it may lead to unnecessary complexity and restrictions on the interface between the user and system levels.

c. Two approaches are often used to handle the states of a temporal database. In the *partially-materializing* approach, only some of the states are stored while the others are recreated on request by some deduction scheme. Data retrieval, however, will then suffer similar shortcomings as the name-driven approach. In the *fully-materializing* approach, all states are stored physically. It is more flexible for accommodating other functions such as view update [6,7], updating an intermediate database entity (e.g., the join of two relations), etc.

2. A multi-list data storage model

This section presents a data model (MD) as the design basis for our algorithms. It makes use of a set of multi-lists and two special data values: *empty* and *null*, which make consecutive intervals and virtual nullification feasible.

The data storage model DM:

In DM, the attributes of a record (top of Figure 1) are divided into three groups:

a. The first group contains the key d of the data object instance.
b. The second group contains all the time-independent attributes pertaining to d.
c. The third group is a header pointing to an *e-time history* (e-history) E(d). E(d) is a multi-list whose header is a sequence of pairs of the form (e, pointer) arranged in ascending order of value of e, where pointer is the address of a t-history T(d, e, t).
   The *transaction history* (t-history) *of d at e-time e as of t-time t*, in notation T(d, e, t) or simply T(e) if d and t are known from the context, is the following sequence:

$$T(d,e,t)=T(e)=\{(v_1,t_1),\ldots,(v_j,t_j),\ldots(v_{h-1},t_{h-1}),(v_h,t_h)\}, \qquad (1)$$

   where $t_1 \leq \ldots \leq t_j \leq \ldots \leq t_{h-1} \leq t_h \leq t$, and (v_j, t_j) represents the value v_j of d as processed at t-time t_j. T(e) contains all values which occur at the same e-time e but are processed at different t-time instants.

An *empty* interval implies that the data object d has no real-life value in that interval. When a *null* ($\aleph$) value at e-time e is assigned to d in t-time interval [t, t'), it means that the current value of d is nullified and d resumes the value just before the current one. A value v is *valid* if it is neither empty nor null. A *valid interval* of v is an e-time interval in which v is valid. *now* is the computer-clock time when an operation is processed.

In our model, a data value remains constant until being modified at the next time instant. By allowing the value empty, intervals can be considered as consecutive and represented by single discrete values of their boundary points. The null value $\aleph$ is used to make a data object resume a previous value. Detail of how to do this is given in Section 3.A. Null values have been used in conventional DBMSs but for a different purpose [11].

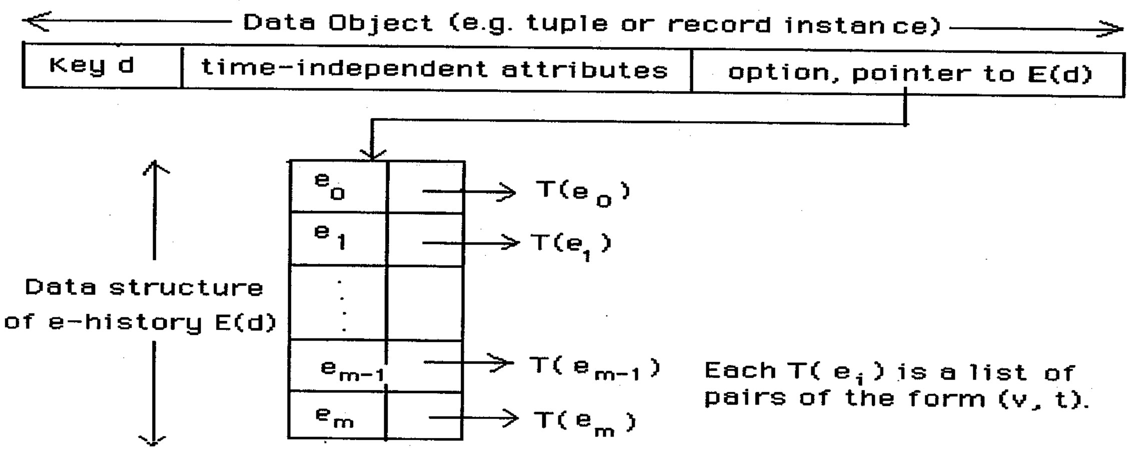

Data Object (e.g. tuple or record instance)
Key d
time-independent attributes
option, pointer to E(d)
Data structure of e-history E(d)
e_0
e_1
e_{m-1}
e_m
$T(e_0)$
$T(e_1)$
$T(e_{m-1})$
$T(e_m)$
Each $T(e_i)$ is a list of pairs of the form (v, t).

Fig. 1. The data storage model DM representing the history of a data object.

3. Three system-level algorithms for bitemporal databases

Based on the above data model, this section presents three system-level algorithms for modifying data objects and retrieving their valid values and intervals. Here, a modification is on the values of the same data object but at different e-time or t-time.

A. *An Algorithm for Modifying the History of a Data Object*

Suppose a modification M on the value v at e-time e is initiated for processing at t-time t. As explained below and in Figures 2 and 3, if v is neither empty nor null, M can be a change, insertion or update, depending on the value of e with respect to the e-time $e_0, \ldots e_i, e_{i+1}, \ldots e_m$ already recorded. M is a *deletion* if v is empty or a *nullification* if v is null.

Operation:	**Conditions:**	**Explanation:**
change:	$e = e_i < e_m$	value is changed to v in $[e_i, e_{i+1})$
insertion:	$e_i < e < e_{i+1}$	value remains unchanged in $[e_i, e)$ and changed to v in $[e, e_{i+1})$
update:	$e_m \le e$	value remains unchanged in $[e_m, e)$ and changed to v in $[e, \infty)$
deletion:	v = empty	same as the above three operations except that, after a deletion, the interval has no value
nullification:	v = null	same as the above three operations except that, after a nullification, the old value is resumed in the new interval

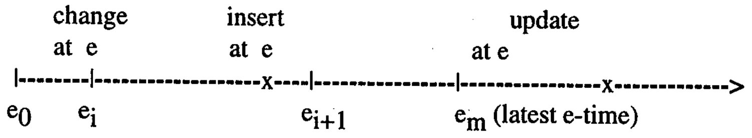


Fig. 2. Classification of operations.

Deletion and nullification need further explanation. They are both done by appending the pair (v, t_{h+1}) to T(e), where $t_h < t_{h+1} < t$. Figure 3 explains the value of d in $[t_{h+1}, t)$ before and after the operation. After both operations, the value in $[t_h, t_{h+1})$ is still v_h.

The value null has the backward effect that an ℵ can nullify another ℵ and thus that all v_i's in the specified e-time interval my be nullified. In the second case, the value in the previous e-time interval will be extended into this interval.

* value in $[t_{h+1}, t)$ is v_h before operation
* no value in $[t_{h+1}, t)$ after deletion
Value: v_{h-1} v_h * value in $[t_{h+1}, t)$ is v_{h-1} after nullification

---------|----------|-------------|--|--->

t-time: t_{h-1} t_h t_{h+1} (operation is processed at t_{h+1}) t

Fig. 3. Explanation of the deletion and nullification operations.

Note that the above classification of operations is purely logical. Operationally, as will be seen in Algorithm ADD, each is done by just adding a pair (v, now) to d's history.

***Algorithm ADD(d, v, e)*:** This algorithm adds the pair (v, now) at the end of the t-history T(d, e, now). As a consequence, v becomes the new value at e as of now.

***Input*:** Data object d (i.e., its key), value v and e-time e.

***Output*:** None.

Procedure (See Figure 1)**:** The given key d is first used for locating the appropriate tuple, where the pointer to the e-history E(d) can be found. E(d) is then searched for an entry indexed by e. If such an entry is not found, this operation is an insertion. Then, create the new entries T(e) = {(v, now)} and [e, ptr], and insert the latter into E(d), where ptr points to the location of T(e). If an entry is found, at e_i say, then append the pair (v, now) to $T(e_i)$.

B. *Two Algorithms for Retrieving Valid Values and Valid Intervals*

Algorithm INT(d, t) extracts the valid values and valid intervals for all the recorded e-time instants as of t. Algorithm VAL(d, e, t) extracts only those values which are valid at e as of t. A retrieval is *normal* if t = now and *rollback* if t < now.

***Algorithm INT(d, t)*:** For data object d, this algorithm retrieves all the valid (i.e., nonnull and non-empty) values and their valid intervals as of t-time t.

***Input*:** Data object d (i.e., its key) and t-time t.

***Output*:** A sequence SEQ = $\{(v_i, [e_i, e_{i+1}))\}_{i=1,\dots,n}$, which states that, as of t-time t, d has valid value v_i in e-time interval $[e_i, e_{i+1})$, for i =1, 2, ... n.

***Procedure*:**

```
begin */ Obtain the sequence S of non-null values at all
    the e-time instants as of t-time t. m is the subscript of
    the last e-time $e_m$ /*
  Locate e-history E(d) by d.
  S:= $\phi$; size := 0;  j := m                          (2)
  while j $\ge$ 1 do
  */  For each $e_j$, extract the current nonnull value from
      T(d, $e_j$, t). (see sequence (1)) /*
  begin  For $e_j$, obtain T(d, $e_j$, $\infty$) from E(d). Let
      T(d, $e_j$, t) = {($v_1$, $t_1$), ...($v_i$, $t_i$), ... ($v_h$, $t_h$)}, where
      $t_1 \le \dots \le t_j \dots t_{h-1} \le t_h \le t$, be the remainder after
      eliminating from T(d, $e_j$, $\infty$) all pairs whose t-time
      is bigger than t.
      k := h
      while $v_k$ = ℵ and k $\ge$ 2 do begin k := k - 2 end
      If k $\ge$ 1 then
         begin S := ($v_k$, $e_j$).S; size := size + 1 end   (3)
         j := j - 1
    end
    If size = 0 then stop                                (4)
*/ Suppose S = { ( $v_1$, $e_1$), ..., ( $v_i$, $e_i$),... ($v_{size}$, $e_{size}$) }
    after relabeling the subscripts, where $v_i$ is the non-null
    value of d at $e_i$ as of t. /*
*/ Create e-time intervals with non-empty values /*
    SEQ := $\phi$; i := 1
    while i < size do
      begin if $v_i \ne \phi$ then SEQ := SEQ. ($v_i$, [$e_i$, $e_{i+1}$])
            i := i + 1
      end
    coalesce (SEQ)
end
```

Note: Procedure coalesce(SEQ) combines consecutive intervals in SEQ which have identical value by repeating the following task until it is no longer possible: "Replace two consecutive pairs (v_i, [e_i, e_{i+1})) and (v_{i+1}, [e_{i+1}, e_{i+2})), where $v_i = v_{i+1}$, with the pair (v_i, [e_i, e_{i+2})). After the combination, rearrange the subscripts."

***Algorithm VAL(d, e, t)*:** This algorithm retrieves the nonnull but possibly empty value and its latest e-time interval ending at e as of t-time t.

***Input*:** Data object d (i.e., its key), e-time e and t-time t.

***Output*:** A pair S = (v_k, [e_j, e]), which states that, as of t-time t, v_k is the current, nonnull (but possibly empty) value of d in the e-time interval [e_j, e].

***Procedure*:** This algorithm contains those statements of Algorithm INT(.) up to Line (4) but with two modifications: a) /*Identify the latest e_j on or before e */ Insert the two statements '**while** $e_j > e$ and $j > 0$ **do begin** j := j - 1 **end**' and **If** j = 0 **then stop**" immediately after Line (2), where **stop** terminates the algorithm. b) Line (3) is replaced with the statement ' **If** $k \geq 1$ **then begin** S := (v_k, [e_j, e]); **stop end**'.

Example 1. ADD, INT and VAL had been applied to an entity-relationship database used in Statistics Canada [2], where A and D mean 'being-alive' and 'die', respectively. Starting from an empty database, the triplets listed in Column 1 of Table 1 have been ADDed at t-time t (Column 2). After these operations, as shown in Fig. 4, entity d has e-history E(d) = {e_1|T(e_1), e_2|T(e_2), e_3|T(e_3)}, where $e_1 < e_2 < e_3$ and $t_1 < t_2 < t_3 < t_4 < t_5$.

ADD(d,v,e)	**t-time**	**Explanation**
(d,empty,e_1)	t_1	data object d is instantiated at e_1 without an initial value
(d, A, e_2)	t_2	being-alive (i.e., born) at e-time e_2
(d, D, e_3)	t_3	died at e-time e_3
(d, ℵ, e_2)	t_4	birth at e_2 as processed at t_2 is erroneous and thus nullified
(d, ℵ, e_3)	t_4	death at e_3 as processed at t_2 is also erroneous and thus nullified
(d, ℵ, e_2)	t_5	the birth nullification processed at t_4 is erroneous and is nullified
(d, ℵ, e_3)	t_5	the death nullification processed at t_4 is erroneous and is nullified

Table 1. List of additions to entity d of an entity-relationship database

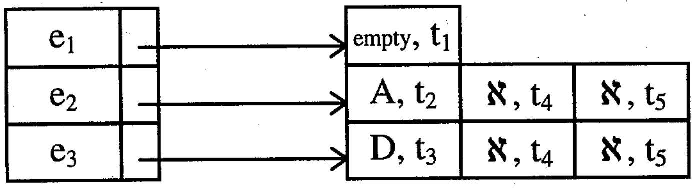


Fig. 4. The multi-list E(d) of entity d after performing the additions listed in Table 2.

Algorithm INT(d, t) obtains the valid values and intervals as of t-time t as follows: If $t_5 \leq t$, SEQ = {(A, [e_2, e_3)), (D, [e_3, ∞))}. This means that, on or after t-time t_5, d is recorded as alive in [e_2, e_3) and as dead on or after e_3. If $t_4 \leq t < t_5$, SEQ = {(empty, [e_1, ∞))}. This means that, between t_4 and t_5, d is recorded as having no valid value since e_1.

The following four cases show the results of applying Algorithm VAL(d, e, t).

	Condition on e and t of VAL(d, e, t)	**Extracted pair**
Case 1:	$e_1 \leq e < e_2$ and $t_4 \leq t < t_5$	(empty, [e_1, e])
Case 2:	$e_2 \leq e < e_3$ and $t_4 \leq t < t_5$	(empty, [e_2, e])
Case 3:	$e_2 \leq e < e_3$ and $t_5 \leq t$	(A, [e_2, e])
Case 4:	$e_3 \leq e$ and $t_5 \leq t$	(D, [e_3, e])

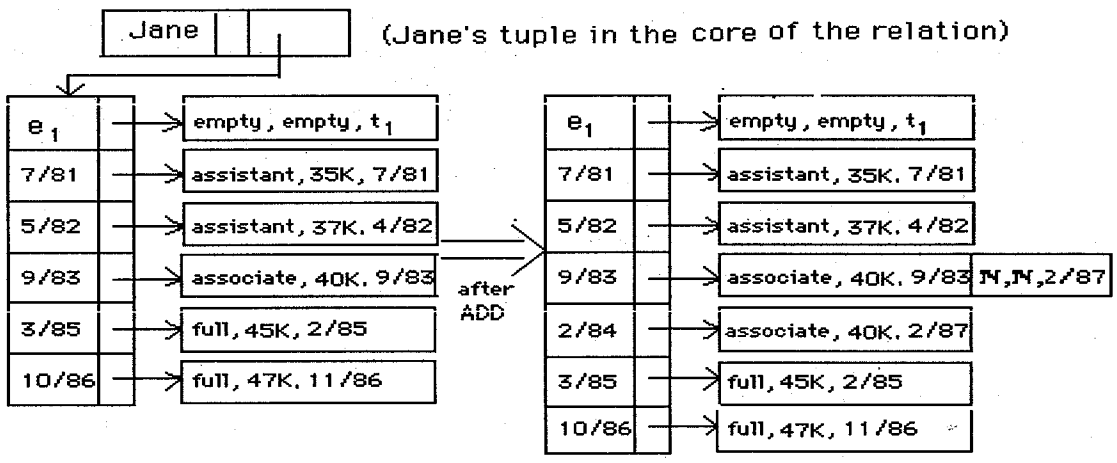
Example 2.

Table 2 shows a bitemporal relation used by Sarda [13], where [FROM,TO] is the e-time interval, [START, STOP] is the t-time interval, and infinity (∞) means that the value will last from FROM or START until it is changed. It contains only one object 'Jane'. Since Sarda's model uses tuple-timestamping, 'Jane' appears in all the combinations of e-time and t-tiem intervals. Figure 5.a shows the same database under our model. It contains only one tuple and an associated history E(Jane) = {T(e_1), T(7/81), T(5/82), T(9/83), T(3/85), T(10/86)}, where e_1 is the initial e-time and t_1 is the initial t-time.

Suppose the date on which Jane's salary became 40K should be 2/84 instead of 9/83. Correction is done on 2/87 by applying ADD(Jane, ℵ, ℵ, 9/83, 2/87) and ADD(Jane, associate, 40K, 2/84, 2/87). The first ADD nullifies the error occurring on 9/83 and the second inserts the updated value occurring on 2/84. Figure 5 shows the e-history for Jane as of 11/86 (just before correction is made) and as of 2/87 (just after correction is made).

NAME	RANK	SALARY	FROM	TO	START	STOP
Jane	Assistant	35K	7/81	5/82	7/81	∞
Jane	Assistant	37K	5/82	9/83	4/82	∞
Jane	Associate	40K	9/83	3/85	9/83	∞
Jane	Full	45K	3/85	10/86	2/85	∞
Jane	Full	47K	10/86	∞	11/86	∞

Table 2. Sarda's bitemporal relation [13].



(a) e-history of Jane as of 11/86 (b) e-history of Jane as of 2/87.

Fig. 5. Multi-list E(Jane).

Shown below are two calls of V(d,e,t) for finding Jane's salary immediately before 10/83 as recorded on 1/84 and 1/88. The two retrieved pairs are also shown.

VAL(d, e, t):	**Retrieved pair** (value, [e, e']):
VAL(Jane, 10/83, 1/84)	(associate, 40K, [9/83,10/83])
VAL(Jane, 10/83, 1/88	(assistant, 37K, [5/82, 10/83]).

4. Summary of comparison on design strategies and Performance

A detailed performance analysis of our algorithms ADD and INT and comparison with Sarda's algorithms CORRECTION and ROLLBACK [14] have been carried out in [1]. (Algorithm VAL is ignored because it is similar to Algorithm INT.) However, because of space limitation, only a brief summary of the results is reported here. We have chosen Sarda's algorithms for comparison for tworeasons: (1) Sarda's algorithms and ours are based on quite different approaches. A comparison will illustrate the pros and cons of the various strategies. (2) As far as we know, Sarda's algorithms are the only ones reported in the literature with detailed description of their design strategies and performance.

	Design Strategy:	Our Algorithms:	Sarda's Algorithms:
1.	*Database supported:*	Bitemporal	Historical
2.	*Representation of interval:*	Consecutive By a single time value	Non-consecutive By a pair of time values
3.	*Data structure:*	Multi-lists Attribute-value timestamping (each key appears only once in the entire database)	Flat files Tuple-timestamping (each key appears in as many tuples as there are time intervals)
4.	*Data storage method:*	Fully-materializing	Partially-materializing
5.	*Data retrieval method:*	Direct indexed searching over carefully-designed multi-list structures	Indexed searching and rollback over flat files
6.	*Operation handling:*	Time-driven	Name-driven

Table 3. Comparison on design strategies.

Selection of the above strategies has rendered the following advantages for our approach:

a. *Our algorithms are simpler.* In our approach, modifications are done simply by adding new data units to the database and retrievals by direct indexed searching; whereas Sarda's algorithms, besides indexed searching, involve storing and manipulating the operations' names and rolling back the tuples.

b. *Our approach is easier for implementation and integration into an existing database.* In our approach, the temporal information is all kept in a close unit (the history) which may be stored separately from the existing database. In each tuple, the fields for the time-related attributes are replaced by a pointer pointing to the relevant multi-list. Hence, modification to the existing database is minimal. Sarda's approach not only lengthens a tuple with some time attributes but also increases the number of tuples by as many as there are time-intervals for each key instant.

c. *Our modification algorithm has the capability in 'recovering' old values* of a data object by just updating it with a null value. (Section 3.)

B. Comparison on Performance of Modification Algorithms ADD and CORRECTION

Comparison is based on two estimates: efficiency and storage space requirement. They are obtained by calling ADD and CORRECTION each to perform a set of operations: update, insertion, deletion and change. The average number of blocks accessed (for efficiency estimation) and the total number of data units required (for storage space estimation) are then estimated based on certain formulas. For uniformity in comparison, we derive the estimates based on e-time intervals for both ADD and CORRECTION.

Difference in storage space requirement

Symbol:	Meaning or assumption:
a	total field size for all time-dependent data attributes
b	total field size for all time-independent data attributes
k	number of recorded e-time intervals overlapping with [e, e')
p	length of a time value or a pointer (assumed to be of same size)

A formula showing the difference between the space requirements of the two algorithms in terms of the parameters b, a, p and k has been obtained in [1]. As illustration, Table 4 shows the computational results from this formula for some randomly chosen values of b, a, p and k.

(S_s - space requirement of Sarda's Algorithm CORRECTION
S_a - space requirement of Algorithm ADD proposed in this paper)

b	a	p	k	$S_s - S_a$
32	8	4	1	301
			2	287
			3	273
40	4	4	1	323
			2	317
			3	311
40	8	4	1	349
			2	335
			3	321
40	16	4	1	401
			2	371
			3	341

b	a	p	k	$S_s - S_a$
40	8	8	1	403
			2	389
			3	375
48	8	8	1	451
			2	437
			3	423
48	16	8	1	503
			2	473
			3	443
64	16	8	1	599
			2	569
			3	539

Table 4. Difference in space requirements between CORRECTION and ADD.

REFERENCES

[1] T. Y. Cheung and X. Ye, "Modification and retrieval algorithms for bitemporal databases", Tech. Report TR-95-1, Dept. of Computer Science, City Univ. of Hong Kong.

[2] T. Y. Cheung, "Temporal databases -- their present and future", *Proc. 5th Intern. Hong Kong Computer Society Database Workshop*, Hong Kong, (1994), pp.29-46.

[3] H. Gunadhi and A. Segev, "Efficient indexing methods for temporal relations", *IEEE. Trans. on Knowledge and Data Eng.*, vol.5, no.3, (Jun. 1993), pp.496-509.

[4] C. S. Jensen, L. Mark, and N. Roussopoulos, "Incremental implementation model for relational databases with transaction time", *IEEE. Trans. on Knowledge and Data Eng.*, vol.3, no.4, (Dec. 1991), pp.461-473.

[5] C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass, "A glossary of temporal database concepts", *SIGMOD Rec.*, vol.21, no.3, (Sep. 1992), pp.35-43.

[6] C. S. Jensen, M. D. Soo, and R. T. Snodgrass, "Unification of temporal data models", *Proc. Intern. Conf. on Data Eng.*, Vienna, (Apr. 1993).

[7] R. Langerak, "View updates in relational databases with an independent scheme", *ACM Trans. on Database Systems,* vol.15, no.1, (Mar. 1990), pp.40-66.

[8] R. Maiocchi and B., Pernici, "Temporal data management systems: A comparative view", *IEEE Trans. on Knowledge and Data Eng.*, vol.3, no.4, (1991), pp.504-524.

[9] E. Mckenzie, and R. Snodgrass, "Supporting valid time in an historical relational algebra: Proofs and extensions", Tech. Rep. 91-15, Dept. of Comp. Science, Univ. of Arizona.

[10] S. B. Navathe and R., Ahmed, "A temporal relational model and a query language", *Information Sciences*, vol.49, (1989), pp.147-175.

[11] M. A. Roth, H. F. Korth, and A. Silberschztz, "Null values in nested relational databases", *Acta Informatica*, vol.26, (1989), pp.615-642.

[12] N. L. Sarda, "Extensions to SQL for historical databases", *IEEE Trans. on Knowledge and Data Eng.*, vol.2, no.2, (Jun. 1990), pp.220-230.

[13] N. L. Sarda, "Time-rollback using logs in historical databases", *Information and Software Technology,* vol.35, no.3, (1993), pp.171-180.

[14] R. Snodgrass, S. Gomez, and E. McKensie, "Aggregates in the temporal query language TQuel", *IEEE Trans. on Knowledge and Data Eng.*, vol.5, no.5, (Oct. 1993), pp.826-842.