Modeling Valid Time: An Efficient Representation

N.L. Sarda
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai, India
nls@cse.iitb.ernet.in

Abstract
The bitemporal conceptual data model proposed for handling time within the framework of relational data model can be effectively represented using valid-time and transaction-time intervals as timestamps on tuples. We observe that in many real-world applications, valid-time interval for a fact is not known completely while entering the fact in a temporal database. The model provides 'open' interval, from some valid time v to 'forever' for time-stamping such facts. However, updating temporal validity of such facts, which invariably will be required, is not handled efficiently by this representation. We propose an alternative representation which can reduce storage cost by up to 50% and which also results in less I/O during updates. We define the conventional and temporal algebra operations for this representation, and also add a few useful operators for processing open intervals. We propose extensions to temporal SQL to support open ended valid time intervals.

keywords Temporal database, Query processing

1 INTRODUCTION
The research in temporal databases has identified valid time and transaction time as the two main time dimensions along which the data is modeled. The valid time associated with a fact tells us when the fact was true in the modeled reality, whereas the transaction time associated with a fact tells us when the fact was current in the database. Thus, the two time dimensions are orthogonal to each other.

The bitemporal conceptual data model (BCDM) [Sno95] has now been widely accepted for modeling time. BCDM has been used as a basis for TSQL2 [Sno95], which is an extended SQL-2 providing support for storage, retrieval and update of temporal data.

time (x-axis) and valid time (y-axis)[Sno95]. A bitemporal chronon is a rectangle of unit size in this two dimensional space, and is given by pair \((t, v)\) of transaction and valid times.

Conceptually, a tuple is associated with a ‘temporal element’ that gives its full temporal history. The temporal element is a set of bitemporal chronons. For example, the following tuple

<table>
<thead>
<tr>
<th>Name</th>
<th>Role</th>
<th>Salary</th>
<th>(T_t)</th>
<th>(T_v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mohan</td>
<td>programmer</td>
<td>40K</td>
<td>(5,10), (5,11), (5,12)</td>
<td>(6,10), (6,11), (6,12)</td>
</tr>
</tbody>
</table>

specifies that the fact <Mohan, programmer, 40K> is true in valid time 10 to 12 and transaction time from 5 to 6. Graphically, it can be represented as shown in Figure 1 by a rectangle.

![Figure 1: Graphical representation of bitemporal data](image)

The figure also shows some facts about John which is still current. Unless it is changed, it remains current in the database at every system clock tick (which represents advance of transaction time). Conceptually, in chronon \(t = 16\), the tuple for John is represented as

<table>
<thead>
<tr>
<th>Name</th>
<th>Role</th>
<th>Salary</th>
<th>(T_t)</th>
<th>(T_v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>clerk</td>
<td>20K</td>
<td>(15,10), (15,11), ..., (15,15), (UC,10), (UC,11), ..., (UC,15)</td>
<td></td>
</tr>
</tbody>
</table>

where UC (abbreviation for Until Changed) symbolically represents advancing nature of transaction time. At every tick, the temporal DBMS generates a \((t, v)\) pair for every valid time associated with UC, and adds it to the temporal element.

Since a temporal element is a set, a BCR is not in the First Normal Form. Efficient interval-based representations have been suggested for a BCR, where a tuple is time-stamped with a valid time interval and a transaction time interval. A tuple in this representation will be given as \(<x, t_s, t_e, v_s, v_e, v_e, v_e>\), where \([t_s, t_e]\) gives start and end time for transaction time interval\(^1\) and \([v_s, v_e]\) gives same for valid time interval.

The two tuples of Figure 1 can be shown in this representation as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Role</th>
<th>Salary</th>
<th>(T_t)</th>
<th>(T_v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mohan</td>
<td>programmer</td>
<td>30K</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>Mohan</td>
<td>clerk</td>
<td>20K</td>
<td>15</td>
<td>10</td>
</tr>
</tbody>
</table>

The insert and delete operations can be defined on this representation as follows (for formal specifications, see [Sno95]):

- **insert**: if the tuple contains a new fact \(x\) over valid time \([v_1, v_2]\), it is added simply as \(<x, t_s, v_1, v_2>_s\), where \(t_s\) is current transaction time. If the fact already exists and is current, then the valid interval \([v_1, v_2]\) is adjusted to remove overlap (if any) with the existing tuple before adding the new tuple to the relation.

- **delete**: the stop time is changed from UC to the current transaction time \(t_c\).

- **update**: it is handled using delete followed by insert.

### 2.1 OPEN VALID INTERVALS

Let us consider real-world situations where the ending valid time for a fact is not known. This is a frequent situation in real-world. New facts to be added to a temporal relation will most likely have undefined ending time. [Sno95] have analyzed use of time variables in temporal databases. The ‘moving’ time variable NOW has been used by many researchers to indicate this situation. In TSQL2, the keyword ‘forever’ is suggested for this purpose. We prefer to denote it by OPEN, implying that the fact is valid now and will continue to be valid in future until the ending valid time is explicitly changed. Consider an open tuple (we do not show RANK attribute to conserve space):

- **Alice**
  - Salary: 100K
  - Valid time: from 30 onwards
  - Open: yes

  It states that Alice is employed on salary 100K from valid time 35 onwards, and that this fact is inserted in the database at system time 30.

After Alice puts in some service, some data about her will change. Let us consider an update to her salary from valid time 65 onwards carried out at transaction time 60. In the interval-representation, the effect of this update will be as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Trans. Time</th>
<th>Valid Time</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>100K</td>
<td>35</td>
<td>59</td>
<td>35</td>
</tr>
<tr>
<td>Alice</td>
<td>100K</td>
<td>60</td>
<td>UC</td>
<td>35</td>
</tr>
<tr>
<td>Alice</td>
<td>110K</td>
<td>60</td>
<td>UC</td>
<td>65</td>
</tr>
</tbody>
</table>

\(^1\)The notation for interval in square brackets indicates that the interval includes both end points.
The 1st tuple gives Alice's data as it was known to system from 30 to 59. This tuple is no more current from system time 60. The 2nd tuple gives modified data of Alice for valid period 35 to 64. Thus, at 60, system knows that Alice's salary was 100K from valid time 35 to 64 (but at 59 it was 100K from 35 to OPEN). The 3rd tuple is Alice's data after the salary raise she gets from valid time 65. (The 2nd tuple in (2) can be seen as error correction on tuple 1.)

It should be noted that salary revision for Alice consisted of one error-correction (where we revised valid-time of the earlier data) and one insert. The net effect is that one real-world update has resulted in (logical) deletion of an existing tuple and insertion of two new tuples.

2.2 OVERHEADS

A temporal database records every belief as it is known at every time instant in the real-world. Hence, update to an open-ended fact results in addition of two new tuples as seen above.

There is an obvious space and performance overhead in such a representation. The overheads become significant when most real-world activities (like employment of persons, processing of orders, etc.) would have open valid time when they begin and get recorded in the database. Only when the activities complete, we can supply end values for valid times. One can see that only for activities completed in the past, we have exact valid time interval to submit to the temporal DBMS.

We treat the resulting representation as creating overhead because tuples where $T_e \neq UC$ are really equivalent to 'deleted' tuples, which surface in the processing only when the transaction time is rolled back to a value before the $t_e$ time value. These tuples do not participate in processing when we view history as it is known presently, which would be the dominant use. An alternate representation, which preserves all the temporal history (along with the roll-back capability) but reduces storage overheads, will be a useful contribution towards efficient implementation of temporal databases.

In this paper, we present an alternate representation that can reduce storage space requirements by as much as 50% (assuming that most tuples undergo updates).

The overhead in processing in the earlier scheme should also be obvious: firstly, we are doing more work in creating 'deleted' tuples whenever updates to valid end time are performed. One such update consists of 1 read, 1 rewrite, and 2 inserts. Secondly, we will load our access paths unnecessarily with deleted tuples (e.g., index on employee name alone or concatenated with valid start time will encounter deleted tuples even when we are not performing roll-back). In the representation proposed here, we reduce these overheads. Each update to valid interval requires 1 less insert, and the burden on access paths is reduced by reducing number of tuples in the tables.

If a temporal DBMS has processed $n$ insertions (say, for newly hired employees), and $p$ updates (each of which also includes one valid-time correction as explained above), then the number of tuples in the DB will be

$$n + 2p$$

Of these, $p$ will have $T_e \neq UC$. Considering these tuples to be an unnecessary burden (which our alternate representation is able to avoid), the space overhead in the conventional interval-based representation is

$$\frac{p}{n + 2p}$$

Example 1: Consider employee database of an organization over a time frame of 10 years. It employs, on average, 500 persons. Every year, there is a turnover of 10% and 90% receive a salary increase or new rank or both. Thus,

| inserts (n) | 50 (new employees) |
| deletion | 50 (employees who resign) |
| updates (p) | 450 |
| Total | 550 (per year) |

% space wasted =

$$\frac{450}{50 + 900} \times 100$$

= 47.37%

It may be noted that the conventional representation will have two tuples for deleted employees also. The space wasted in that representation rises to 50% when we take this into consideration. Our alternate representation will provide space savings of 50%.

Example 2: Consider an order processing application, where, upon receiving a fresh order, an insertion is made with OPEN valid interval. When the order is fully processed, the tuple is corrected by (logically) deleting earlier tuple and adding a new tuple with proper ending valid time. There are two tuples per order completed. Thus, % of tuples with $T_e \neq UC$ will be 50% of the total.
table size. The representation suggested by us will provide space savings of 50%.

3 EXTENDED INTERVAL REPRESENTATION (EIR)

In this section, we propose an extension to the interval representation discussed above. We extend the definitions of insert, delete, update and rollback for this representation. We also demonstrate how the scheme avoids the overheads of the earlier representation.

A tuple in EIR consists of the following values:

- **x**: value of visible attributes
- **t_e**: transaction time when tuple entered (= UC if tuple is current in database)
- **t_o**: transaction time until which the valid interval of this tuple was having its end-time as OPEN (t_o could be = UC)
- **v_s**: start of valid time interval
- **v_e**: end of valid time interval (= OPEN if end is not yet defined)

We will denote the corresponding attributes, respectively, as X, T_e, T_o, V_s and V_e.

Let us first illustrate use of EIR for handling real-world situations with OPEN end time. We will consider other situations subsequently.

Consider the open-ended current tuple (1) as before. It will be represented as follows:

<table>
<thead>
<tr>
<th>Alice</th>
<th>100K</th>
<th>30</th>
<th>UC</th>
<th>UC</th>
<th>35</th>
<th>OPEN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

At transaction time 60, Alice’s salary is revised to 110K from valid time 65 onwards. The above tuple is then modified as follows:

<table>
<thead>
<tr>
<th>Alice</th>
<th>100K</th>
<th>30</th>
<th>UC</th>
<th>59</th>
<th>35</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>110K</td>
<td>60</td>
<td>UC</td>
<td>UC</td>
<td>65</td>
<td>OPEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that the two tuples in (4) are equivalent to three tuples in (2) earlier. The 1st tuple in (4) states that the fact <Alice,100K> was known to have validity [35, OPEN] in system time [30, 59], but its validity is [35, 64] in system time [60, UC]. Thus, this tuple contains two temporal facts for one real-world fact.

The T_o attribute, as in the standard interval-based model, represents time until which the fact was current; a fact is still current when T_o = UC.

If T_o = UC, the tuple simply represents one temporal data for x, that x is valid over (v_s,v_e) and current over (t_o,t_e). When V_e = OPEN, the T_o should be = UC (the reverse need not be true; if T_o = UC, V_e need not be OPEN; such a tuple represents a fact whose valid interval was completely known when it was inserted in the database). When T_o ≠ UC, the tuple represents two temporal facts for a tuple. Let us consider pair-wise values of T_e and T_o and what they signify:

<table>
<thead>
<tr>
<th>(T_e, T_o)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC, UC</td>
<td>tuple is current; T_o has no significance</td>
</tr>
<tr>
<td>t_e, UC</td>
<td>tuple current upto t_e; T_o has no significance</td>
</tr>
<tr>
<td>UC, t_o</td>
<td>represents two temporal facts; a tuple whose v_e has been modified from OPEN to some value v at time t_o.</td>
</tr>
<tr>
<td>t_e, t_o</td>
<td>tuple current upto t_e; it also represents two temporal data about a fact; t_o &lt; t_e</td>
</tr>
</tbody>
</table>

In general, the tuple < x, t_o, t_e, t_o, v_s, v_e > represents the following two validity and currency data for fact x when t_o ≠ UC:

- valid over (v_s,OPEN) and current from (t_o,t_o)
- valid over (v_s,v_e) and current from (t_o + 1,t_e)

The valid transitions between values of (t_e, t_o) are shown in Figure 2. Let us now consider situations

![Figure 2: valid transitions on T_o, T_e](image)

where open-ended intervals are not involved:

- John is employed from 20 to 40, and data entered at 20.

<table>
<thead>
<tr>
<th>John</th>
<th>30K</th>
<th>20</th>
<th>UC</th>
<th>UC</th>
<th>20</th>
<th>40</th>
</tr>
</thead>
</table>

- John’s employment data above is incorrect; it should be with salary 50K from 20 to 45 (correction at 33).
Thus, $T_o$ attribute comes into play only for valid intervals having $V_x = \text{OPEN}$ initially. It changes its value from UC to some $t$ only when OPEN is modified. $T_o$ attribute is a burden on storage if our data is not open-ended when entered.

4 UPDATE AND ALGEBRA OPERATIONS

In this section, we will define insert, delete and update operations, and also extend the temporal algebra operations for the extended interval representation scheme proposed in Section 3. We will denote a typical tuple in this representation as:

$$< x, t_x, t_e, t_o, v_x, v_e >$$

and the current transaction time (at which an operation is performed will be denoted by $t_c$. $R$ will denote a temporal relation, and $r$ a tuple in $R$. $r[A]$ projects $r$ on attribute $A$. $V$ and $T$ will refer to valid time and transaction time intervals in a tuple.

UPDATE & TEMPORAL OPERATIONS

1. Insert tuple $< x, t_x, v_x >$ in $R$ at system’s current time $t_c$:
   
   If the temporal relation $R$ already contains the fact $< x >$ and it is current, we must exclude from the tuple being added any overlap in valid time with the tuple already present. The algorithm for insert (on similar lines as in [Sno95]) can be stated as follows:

   $$P = \{[v_1, v_2]\}$$
   for each $r \in R$
   if $r[X] = x$ and ($T_e = UC$)
   for each $p \in P$
   if $r[V] \cap p$ /* overlap present */
   $P \leftarrow (P - p) \cup (p - r[V])$
   for each $p \in P$
   insert $< x, t_x, UC, UC, v_x, v_e >$ in $R$
   return $R$

   Here, $P$ is a set of time intervals, initialized to the valid interval of the tuple to be inserted. The first for-loop checks if $R$ contains a current tuple with $< x >$ for visible attributes. If yes, intervals in $P$ are broken down to remove the overlap. The second for-loop inserts a new tuple for each subinterval in $P$ (these are the ones not covered by the existing tuples).

   Note that $T_e$ will be either $UC$ or have a value less than $t_c$. Hence, we need to consider valid time interval in existing tuples to be as given by their $V_x$ and $V_e$ attributes.

2. Delete $< x >$ from $R$ at current time $t_c$:
   
   If relation $R$ contains the fact $< x >$, and it is current (ie., its $T_e = UC$), the delete operation terminates its currency by making $T_e = t_c$.

3. Delete validity of $< x >$ during valid time $[v_1, v_2]$ at $t_c$:
   
   This operation is useful when fact $< x >$ is present in $R$ but its validity needs to be modified. As an example, consider $< x, 20, UC, UC, 15, 75 >$ as an existing tuple, and we want to delete $< x >$ over $[41, 60]$ at $t_c = 30$. Effectively, the existing tuple needs to be replaced by the following three tuples:

   $$< x, 20, 29, UC, UC, 15, 75 >$$
   $$< x, 30, UC, UC, 15, 40 >$$
   $$< x, 30, UC, UC, 61, 75 >$$

   Note that the last two tuples record new validities of $< x >$ over intervals which are outside the deleted interval. This operation can be implemented using one delete and two inserts, but it requires users to compute the two subintervals not covered by the interval specified in delete. For this reason, we consider it beneficial to provide the delete-validity operation. It can be defined procedurally as follows:

   for each $r \in R$
   if $r[X] = x$ and $T_e = UC$
   $P \leftarrow \text{split}(r[V], [v_1, v_2])$
   $r[T_e] = t_c$ and rewrite $r$
   for each $p \in P$
   insert $< x, t_c, UC, UC, v_x, v_e >$ in $R$
   return $R$

   The function split returns in $P$ those subintervals of $r[V]$ which do not overlap with $[v_1, v_2]$.

4. Close-validity of $< x >$ to $v_e$ at $t_c$:
   
   This operation is provided for closing open interval of tuple(s) containing $x$. Its ending valid time will be changed from OPEN to $v_e$. This operation sets value for $T_e$.

   for each $r \in R$
   if $r[X] = x$ and $T_e = UC$ and $V_x = OPEN$
   $r[T_e] = t_c$
   $r[V_x] = v_e$
   rewrite $r$

   Note that this operation can not be simulated using insert and delete defined above. Also note that it can not be simulated using delete-validity of $< x >$ over interval $[v_c, OPEN]$. None of these operations use the semantics of $T_o$.  

367
5. Rollback, \( \rho_t(R) \), rolls back temporal relation \( R \) as it was known to the system at time \( t \). It gives real-world validity of data as it was known at \( t \). The result of rollback is a valid-time relation \( S(X, V_s, V_e) \) obtained as follows:

   for each \( r \in R \)
   if \( t \in \{ t_s, t_e \} \)
     if \( t_s > t \) and \( t_e \neq UC \)
       insert \( <r[X], r[V_s], \text{OPEN}> \)
         in \( S \)
     else
       insert \( <r[X], r[V]> \)
         in \( S \)
   return \( S \)

6. Timeslice, \( \tau_v(R) \), gives tuples which have/had validity at \( v \). It produces transaction-time relation \( Z(X, T_s, T_e) \) as follows:

   for each \( r \in R \)
   if \( t_o \neq UC \)
     if \( v \in \{ V_s, \text{OPEN} \} \)
       insert \( <r[X], r[T_s], T_o > \)
         in \( Z \)
     if \( v \in \{ [v_s, v_o] \} \)
       insert \( <r[X], T_o + 1, r[T_e] > \)
         in \( Z \)
     else
       if \( v \in \{ [v_s, v_o] \} \)
         insert \( <r[X], r[T_s, T_e] > \)
         in \( Z \)
   return \( Z \)

7. Unfold(R) : to convert \( R \) from EIR to interval-based representation.

   The relational algebra operations (as extended for temporal data model) along with new operators (like rollback and timeslice defined above) have been defined in the literature for interval-based representation (e.g., see [Sno95]). We can use the same definitions after converting the extended interval representation to the standard interval representation. We can define this conversion operation as follows.

   for each \( r \in R \)
   if \( t_o = UC \)
     if \( v \in \{ V_s, \text{OPEN} \} \)
       insert \( <r[X], r[T_s, T_e], r[V]> \)
         in \( S \)
   else
     insert \( <r[X], r[T_s], r[T_e], r[V_s], \text{OPEN}> \)
     and \( <r[X], r[T_o + 1], r[T_e], r[V]> \)
       in \( S \)
   return \( S \)

   It is also possible to define a ‘fold’ operation which will take a relation in interval format and convert it into the extended interval format. However, the operation requires a sort, and may not be required in most cases.

8. Projection : projection retains all the implicit temporal attributes; coalescing of value equivalent tuples will be carried out as in the conventional interval representation.

9. Selection : selection on explicit attributes is straightforward, and the result contains all explicit and implicit attributes. If the selection condition includes a predicate on valid or transaction time, the tuple in the extended representation needs to be unfolded first before applying time predicates:

   \[
   S = \sigma_p(R) = \sigma_p(\text{unfold}(R))
   \]

   Consider the selection predicate \( p \) to be \( pX \land pT \land pV \), a conjunction of predicates on explicit, transaction and valid time attributes. We can optimize the above expression to get result of selection in a single scan of \( R \). Let a tuple \( r \) in \( R \) be \( <x, t_s, t_e, v_o, v_s, v_e> \).

   \[
   S = \{ (r | r \in R \land pX(z) \land (pT(T_o, T_e) \land (pV(V_s, V_e) \lor (pT(T_o, T_e) \land (pV(V_s, OPEN) \lor \{pT(T_o, T_e) \land pV(V_s, V_e))))))) \}
   \]

   Here, the notation \( pT(T_o, T_e) \) means that predicate \( pT \) is true for the interval defined by \( (T_o, T_e) \).

10. Join : if any of the two operands of join is in extended interval format, it can be converted into the standard interval format using the unfold operation before applying the join operation. The join of \( R \) and \( S \) combines tuples \( r \) and \( s \) provided there is an overlap in both valid and transaction times of \( r \) and \( s \). Also, the result timestamp is the intersection of the corresponding time stamps in \( r \) and \( s \). As demonstrated above for the selection operation, we can optimize on the join specification so that no separate scans of \( R \) and \( S \) are necessary for unfolding. It may be noted that the result will be in the standard interval format, and that join of tuples \( r \) and \( s \) may produce up to 4 tuples in the result.

5 EXTENSIONS TO TSQL

TSQL is the consensus query language which extends SQL for providing temporal support. In this section, we suggest extensions to TSQL for efficient handling of open-ended valid intervals. These extensions maintain upward compatibility with TSQL.

5.1 Schema definition:

A valid-time state type table can be defined to support open-ended valid intervals by including the keyword OPEN-ENDED as follows:
CREATE ... 
AS OPEN-ENDED 
  VALID [STATE | EVENT ]... 
  [ AND TRANSACTION ] 
| AS TRANSACTION

When a tuple is inserted in such tables, by default, the ending valid time will be taken as OPEN.

5.2 CLOSE statement

We provide a new data manipulation statement for closing the ending valid time from OPEN to the specified valid time (which, by default, is NOW). CLOSE is applicable only to OPEN-ENDED valid time state tables.

```
CLOSE tablename 
  [ VALID v ] 
WHERE predicate
```

Default for v is NOW. The effect of this statement is to set $V_c$ to $v$ and $T_o$ to $t_c$ (current-time) for selected tuples.

Example: The order table defined below is a bitemporal state table, where we want to store open-ended tuples. These tuples will represent orders received but not yet completed.

```
CREATE TABLE order 
  (ordno INTEGER NOT NULL, 
   customer CHARACTER (30) NOT NULL, 
   item CHARACTER (10), 
   value DECIMAL (8,2)) 
AS OPEN-ENDED VALID STATE 
YEAR TO DATE 
AND TRANSACTION
```

A new order, received today (June 10, 1996) can be recorded as:

```
INSERT INTO Order 
(1234, 'SMITH', 'PC486', 4000)
```

When the order 1234 is fully processed, its status can be closed simply by executing (say, on June 30, 1996)

```
CLOSE ORDER 
WHERE ordno = 1234
```

The ORDER table will have the following tuple after CLOSE:

```
< 1234, 'SMITH', 'PC486', 4000, 6/10/96, UC, 
  6/30/96, 6/10/96, 6/30/96 >
```

(Note: the dates are shown above in mm/dd/yy format.)

If the Order table was the conventional bitemporal state table, we would require a DELETE followed by an INSERT statement. Moreover, for specifying valid time interval in the INSERT, we will need to look up the earlier tuple for the valid start time of the order.

For an open-ended valid time table (which does not include transaction time), the CLOSE statement is still useful. In the standard TSQL, we would require to use UPDATE with VALID PERIOD specification. To illustrate, consider the above order table to be a valid-time table. The INSERT statement given above for order number 1234 will also work fine with default values for valid start and end time (as [6/10/1996, OPEN]). When the order is completed, the CLOSE statement above will correctly set ending valid time value (giving valid interval as [6/10/1996, 6/30/1996]). In standard TSQL, we will need to give an UPDATE as follows:

```
UPDATE order 
VALID_PERIOD ' [10 June 96, 30 June 96]' 
WHERE ordno = 1234.
```

5.3 Upward Compatibility

The extension for open-ended valid intervals proposed here retains full compatibility with the standard TSQL. The statements of TSQL will work correctly for open-ended valid-time as well as bitemporal tables (they simply ignore the implicit $T_o$ time attribute). Hence, the existing application programs continue to work (for retrieval as well as update) correctly when existing tables are made into open-ended valid time tables. What could happen is that new applications (written using extended TSQL) may use $T_o$ attribute in some tuples that they add and process (while $T_o$ will be UC for all older tuples). The extended operators given here will correctly handle TSQL statements in existing and new application programs.

The open-ended tables can be used along with other tables just as in standard TSQL as their semantics is same as a bitemporal relation. The efficient internal representation (and extensions required in algebra) is transparent to TSQL users. Query optimization and processing can be as efficient as in TSQL. To achieve this in a simple manner, we can proceed as follows:

1. translate the (extended) TSQL query into an algebra query and apply optimizing transforms.
2. replace open-ended temporal relation $R$ by unfold($R$) which converts it into the standard interval representation.
3. move up the unfold operation (in the algebraic query tree) over unary operators (like projection, selection), since these unary operations
have been extended for the extended interval representation (with result also being in extended interval representation).

4. in pushing unfold upward, if we encounter a binary operation (like, join), we subsume it with the binary operator, where unfold is applied ('on the fly') to its operand before the binary operation is done. The result of binary operation will be in the standard interval representation.

6 CONCLUSIONS

Handling temporal data within database applications is a fairly common requirement. The research in temporal databases is directed towards providing facilities for modeling, storage, retrieval and maintenance of temporal data. In order to converge towards making specific recommendations that can be widely adopted, the temporal research community has jointly proposed the bitemporal conceptual model and has extended the industry-standard query language SQL.

The conceptual model is not in 1NF (as a temporal relation here implicitly includes a temporal element, which is a set of time values, in every tuple). An interval-based representation (suggested by [Sno87] for bitemporal data model, and by others, see for example [Nav89], [Sar90], for valid-time data models) is an efficient representation for the conceptual model.

However, the model does not handle efficiently the real-world data whose validity extends indefinitely in future. Some authors introduced time-variables such as NOW and 'forever' to represent such data. When such 'open-ended' facts reach end of their validity in the real-world, the conceptual model requires that the earlier tuple be deleted and a new tuple be added. This leads to a high storage cost.

In this paper, we have extended the interval representation in such a way that we do not need to insert a new tuple when we terminate an earlier open-ended fact. While suggesting the extension, we retain the same temporal semantics as the conceptual model in the sense that this representation is equivalent in its contents with the conceptual model. Our representation improves space utilization by up to 50%. There is also a corresponding reduction in disk I/O in update actions. Thus, the extension is highly useful when a temporal relation is expected to contain large number of 'open-ended' facts.

We also suggest a few simple extensions to temporal SQL for defining tables to handle open-ended facts. These extensions make it easier to process data whose valid end-time is not known.

References


[Sar90] Sarda, N., Extensions to SQL for historical databases, IEEE Transactions on Knowledge and Data Engineering 2, 2, pp. 220-230, June 1990.


