A simple approach to temporal data in relational DBMS

M. K. Crowe
University of Paisley, UK

Abstract

Recent articles and research issues suggest that temporal data remains a source of difficulty to database designers and implementers.

This paper analyses some of the issues and suggests some mechanisms that could help simplify the problems, notably a new derived column concept for NEXT, a new temporal join operator and temporal predicates CURRENT and AT. However, the suggestions avoid adding any new data types or table types and the language extensions are in the spirit of the SQL standard. The paper includes full details of the proposed specification of these functions in the style of the SQL standard. These specifications are implemented in version 1.2 of the Pyrrho DBMS, an (otherwise) SQL-2003 compliant relational DBMS.

Keywords: temporal data, DBMS, relational database, fold, interleave.

1 Introduction

Zimanyi [2] includes in his paper an excellent introduction to the history of the SQL/Temporal proposal, and how it did not find a place in as Part 7 of SQL2003 [4], and his article contains numerous examples of temporal data and excellent accounts of how temporal manipulations of such data can be effected using standard SQL.

He largely follows the original approach of Snodgrass [5] and the proposed SQL/Temporal language in defining a temporal database table as one in which there is one or two added pairs of columns FromDate and ToDate (when two pairs are added one is for "valid time" and the other for "transaction time"). This approach has been criticised as misconceived by Date et al [6]: certainly it seems an overly mechanistic approach to a series of subtle semantic problems (Gabbay and McBrien [7]).

Date et al first note that transaction time information is rather special and should be left to the database logs. For semantically-useful temporal data, they prefer to work with explicitly declared columns (so that tables may contain several temporal columns). Their temporal columns are of an interval type (different from SQL intervals) corresponding to the FromDate and ToDate pairs in Snodgrass's approach, and they build on the Allen operators to develop subtle relational operations for such temporal data.

The starting point of this paper is that ToDate columns contain *derived* data, and we explore instead an approach where such columns, if required, can be always calculated from other data in the tables. To motivate such a radical step, we observe that where ToDate columns are not treated as derived, their values must be heavily constrained and subtle arrangements made to update them every time that temporal data in the database is updated.

An extension (a derived column called NEXT) to the SQL standard, documented in this paper, provides a very simple way of computing ToDate, which then becomes a virtual column. There is no need for ToDate to be stored in the database, and, as we shall see, examples where a ToDate has been set explicitly fare better with a different data model. By eliminating the ToDate columns, a huge simplification occurs in the temporal database problem.

By this route we are led to define a very simple concept of "temporal table", which requires no special syntax or semantics, but with associated new concepts of CURRENT, AT and TEMPORAL JOIN. This paper gives the changes to the SQL standard that would result from the adoption of these ideas.

2 Database tables and historical records

Many DBMS's allow access to previous states of the data (called rollback DBMS in the temporal database literature). For such databases, every base table implies the existence of an underlying temporal table, based on transaction time. This underlying table in generally not stored explicitly in the database but its contents are reconstructed from logs when required. For example, the Pyrrho DBMS provides a construct ROWS(nn) which provides a synopsis of the log entries for a given base table, and allows previous states of the table and associated transaction times to be retrieved using ordinary query language.

But it might be as entirely legitimate to make a correction to one of the since or until dates in the People table as it would be to fix the BirthDate of an employee. Such a correction would be unusual, and would continue to be shown in history reports based on logs.

Using information from logs may look an attractive way of addressing the problems of temporal data in simple cases. In practice the presence of corrections to data can make such an approach unusable, and alternatives need to be considered.

2.1 Semantics of date data

To motivate the approach of this paper, consider a People table which records departmental affiliation. Rather than use logs for historical information, we create a new entry for Fred when he changes department.

Table 1: A People table with start and end dates.

Name	Dept	PostSince	PostUntil
Fred	D1	2005-10-02	2005-12-01
Fred	D2	2005-12-01	
Joe	D1	2006-03-25	
Mary	D1	2005-10-08	2006-06-21

The primary key for this table now consists of (Name,PostSince) date (since there is nothing to stop Fred moving back to his old department later on). Note that in this Figure, Mary's departure is recorded by updating her record by giving a PostUntil value.

The temporal database literature provides a number of constraints and rules to ensure that entries in the PosUntil field make sense. For example, if we add a new entry to record Fred's move to department D3 we require also to update the old one with the correct PosUntil field. We could require such a change to be handled by a stored procedure which automatically computes and enters the PostUntil field for his previous post, and trigger mechanisms could achieve this.

However, the viewpoint of this paper is that the two entries in the PostUntil column here have different semantics. We distinguish between the automatic calculation of PostUntil from a new entry, and the recording of the departure of an employee:

Table 2: The People table with start and end dates.

Name	Dept	PostSince	PostUntil	LeavingDate
Fred	D1	2005-10-02	2005-12-01	
Fred	D2	2005-12-01		
Joe	D1	2006-03-25		
Mary	D1	2005-10-08	2006-06-21	2006-06-21

In this table, the values of PostUntil can actually be calculated as COALESCE(NEXT, LeavingDate), using the new function NEXT presented later in this paper. This allows us to dispense with the PostUntil column: there is no need ever to store it in the database.

Table 3: The final version of the People table.

Name	Dept	PostSince	LeavingDate
Fred	D1	2005-10-02	
Fred	D2	2005-12-01	
Joe	D1	2006-03-25	
Mary	D1	2005-10-08	2006-06-21

In this final version, a personnel history is very simply found as SELECT * FROM PEOPLE (provided Mary's record has not been deleted yet), and the current list of employees is SELECT * FROM PEOPLE WHERE CURRENT(PostSince) AND LeavingDate IS NULL. If the PostUntil column is required it can be calculated using the formula given above. The database logs can still be used to examine expired records that have been deleted and to track what changes have been made to the data over time.

A naïve query about which department Fred works in will now give two answers, one of which is out of date. We can overcome this by renaming the tables so that the table in Table 3 is called PeopleHistory and using a view (stored query) so that SELECT * FROM PEOPLE table gives the expected information (one record for Fred, none for Mary) from the PeopleHistory table. A primitive predicate CURRENT, defined using another derived column LAST can ensure that this operation is very efficient. We give some examples of this process below.

2.2 Folding and interleaving

With "semitemporal" tables like Figure 3, it is likely that for the most part each time period will correspond to a new value (of salary, or PNumber say). In general it can happen that the result of query Q contains extra rows, as in Table 4.

Table 4: A table where folding would be useful.

Column1	Column2	FromDate
AAA	BBB	2005-06-18
AAA	BBB	2006-04-11
AAA	CCC	2006-07-05

Here all columns apart from the temporal columns match, and the intervals in the temporal columns are adjacent. We really need to combine the rows to obtain Table 5.

Table 5: A folded version of Table 4.

Column1	Column2	FromDate
AAA	BBB	2005-06-18
AAA	CCC	2006-07-05

Denote the result as

Q FOLD

The FOLD operation requires that its operand is a semitemporal table, i.e. has a primary key whose last component is temporal. The result of folding is a temporal table, and NEXT and LAST can be defined in meaningful ways.

The inverse of FOLD is INTERLEAVE, where additional temporal values are inserted into a temporal table. Table 4 can be obtained from Table 5 by the INTERLEAVE operation. If Q1 gives the result shown in Figure 5, then Figure 4 is obtained by

Q1 INTERLEAVE FromDate WITH VALUES (date '2006-04-11')

2.2.1 Joined tables

Let us suppose that we continue to adopt the approach of Figure 3. Suppose we start with two such tables, Salary and Affiliation.

Salary: (SSN, Amount, AmountSince)

Affiliation: (SSN, DNumber, DNumberSince)

The natural join of these two tables is not very useful.

If we want to omit the second row here, and to combine the date columns to create a new temporal table we should like to end up with Table 7.

Table 6: A natural join of temporal data.

SSN	**Amount**	**AmountSince**	**DNum**	**DNumSinc**
12354789	14500	2004-02-05	D5	2004-02-05
12354789	15000	2005-06-08	D5	2004-02-05
12354789	14500	2004-02-05	D1	2005-01-15
12354789	15000	2005-06-08	D1	2005-01-15
etc				

Table 7: A temporal join of two tables.

SSN	**Amount**	**DNum**	**DateSince**
12354789	14500	D5	2004-02-05
12354789	14500	D1	2005-01-15
12354789	1500	D1	2005-06-08
…			

In fact we define precisely this operation later is this paper as

SELECT * FROM Salary TEMPORAL JOIN Affiliation AS DateSince

2.2.2 SQL2003 proposals

The following paragraphs are suggested modifications to *SQL 2003: ISO/IEC 9075-2:2003 Information Technology – Database Languages – SQL – Part 2: Foundation (SQL/Foundation)* to implement the ideas of this paper. Unless otherwise stated the passages shown modify the corresponding passages in the standard. Section 4.14.10 and 8.20 are entirely new.

4.14.10 Temporal Tables

A table in which the last component of the primary key is of scalar type with a natural ordering, such as a date or timestamp, is called a *semitemporal base table*. The last component TK of the primary key PK is said to be the *default temporal column* of T.

A *semitemporal table* is a table that is either a semitemporal base table or the result of FOLD or INTERLEAVE. FOLD removes “unnecessary” rows in T, which differ from existing rows only in CT, while INTERLEAVE adds such unnecessary rows.

A *temporal table* is a semitemporal table T without unnecessary rows, i.e. such that T = T FOLD. In a temporal table T, the derived columns NEXT and LAST are defined. As with ordinary columns, NEXT and LAST may be prefixed by correlation or table names to indicate their table.

The TEMPORAL JOIN operation is available for performing a sort of natural join on two temporal tables whose default temporal columns have matching types, which are combined in the temporal join operation to provide a default temporal column in the result. For the purposes of this rule, date-time types shall not match if their specified or implied <interval qualifier>s do not match. The temporal join has a natural primary key consisting of the union of the non-temporal columns of the primary keys of the two temporal tables, together with the new temporal column.

The temporal predicate CURRENT(TK) is equivalent to TK=T.LAST, and TK AT V is equivalent to (TK<=V and TK=T.LAST or T.NEXT>V).

5.2 <token> and <separator>

2.2.3 Format

```
<non-reserved word> ::= … omit | LAST

<reserved word> ::= ..
| LAST | NEXT | FOLD | INTERLEAVE | TEMPORAL
```

6.1 <data type>

2.2.4 Format

```
<datetime type> ::=
  DATE [<interval qualifier>]
| TIME [<interval qualifier>]
      [ <left paren> <time precision> <right paren> ]
      [ <with or without time zone> ]
| TIMESTAMP [<interval qualifier>]
      [  <left  paren>  <timestamp  precision>  <right
paren> ]
      [ <with or without time zone> ]
```

2.2.5 Syntax rules

Change rule 32) to If DATE is specified but an <interval qualifier> is not specified, the implicit interval qualifier shall be YEAR TO DAY. If TIME is specified but an <interval qualifier> is not specified, the implicit interval qualifier shall be HOUR TO SECOND. If TIMESTAMP is specified but an <interval qualifier> is not specified, the implicit interval qualifier shall be YEAR TO SECOND.

2.2.6 General rules

Change rule 4) to For a <datetime type> the <primary datetime field>s contained shall be specified by the specified or implicit <interval qualifier>.

6.7 <column reference>

2.2.7 Format

```
<column reference> ::= ...
| <basic identifier chain> <period >
      <derived temporal column>
…
<derived temporal column > ::=
        NEXT
      | LAST
```

2.2.8 Syntax rules

Add

9) If NEXT or LAST is specified, then the <column reference shall be contained in a <query specification> *QS* whose <table expression> is a temporal table TR, *BIC* shall identify a temporal table T. If the <column reference> is contained in a <query specification> QS whose from clause contains only one table T, then *BIC* can be omitted, and the identification of *T* is implicit.

2.2.9 General rules

Add

2) If NEXT or LAST is specified,

a) Let the columns of *TR* be (CL_1,…,CL_n), reordering them if necessary so that the primary key of *TR* is (CL_1,..,CL_k). Then CR=CL_k .Let CL=(CL_1,..,CL_{k-1}).

b) Let LAST be the window function

```
        MAX(CR) OVER (PARTITION BY CL)
```

c) Let NEXT be the window function

```
        MAX(CR) OVER (PARTITION BY CL ORDER BY CR ROWS
        BETWEEN 1 FOLLOWING AND 1 FOLLOWING)
```

NOTE: (a) The use of MAX here is a no-operation since there is at most one row in the window frame. It is required syntactically in this expression.

(b) NEXT and LAST are not derived columns in the sense used elsewhere in the standard, since they are computed in the table they relate to, ignoring any WHERE clauses etc in the query specification in which they occur.

7.6 <table reference>

2.2.10 Format

```
<table primary> ::= …
| <table primary> FOLD
| <table primary> INTERLEAVE
     WITH <query primary>
```

2.2.11 Syntax rules

Add

26) If FOLD or INTERLEAVE is specified, let *PTR* be the <table primary> specified as the argument to the FOLD or INTERLEAVE operation, and let *TR* be the <table primary> specified by the FOLD or INTERLEAVE operation.

a) *PTR* shall be a semitemporal table. Let *CT* be its default temporal column. Let *DT* be the data type of *CT*.

b) If INTERLEAVE is specified, let *QP* be the <query primary>. The result *TT* of *QP* shall be a table with a single column TC of the type *DT*.

c) If FOLD or INTERLEAVE is let *CN* be the name of *CT*.

2.2.12 General rules

Add

6) If FOLD is specified, let the semitemporal table *PTR* have columns $S_1,\dots,S_n$ such that the primary key *CL* is $S_1,..,S_k$, with $S_k=CT$. Let *SL* be the list $SL_1,..,SL_{n-1}$.

Then let *PTR* be ordered by *CL*. Let $NCL=S_1,..,S_{k-1}$, the non-temporal components of *CL*. Partition *PTR* by *NCL*. Each window *TP* in the partition has identical values of *NCL*, and is ordered by distinct values of *CT*. Let $TP = \{TP_{ij} : i=1,..,m; j=1,..,n\}$ be such a window, and let $N = \{ i : TP_{ij} = TP_{i-1\,j}$ for all j such that $k<j<=n \}$.

Define $TS = \{ TP_{ij} : 1<=i<=m$ and $i \notin \mathrm{N}; j=1,..,m \}$.

Then the result *TR* is the union of all such *TS*. Its columns will be the columns of *PTR* with *CT* renamed as *CN*. It is a temporal table with the primary key consisting of *NCL* together with *CN*.

7) If INTERLEAVE is specified, let the semitemporal table *PTR* have columns $S_1,\dots,S_n$ such that $S_n=CT$. Let *SL* be the list $SL_1,..,SL_{n-1}$. Let *TT* be the result of *QP* and let *TC* be its single column. Then

```
TR = SELECT SL,CT FROM PTR
UNION DISTINCT
SELECT SL,TC AS CN FROM PTR TN, TT WHERE TN.CT AT TC
```

The result is a semitemporal table with the same primary key as *PTR*.

7.7 <joined table>

2.2.13 Format

```
<joined table> ::= ...
     | <temporal join>
```

```
<temporal join> ::=
<table reference> TEMPORAL [[AS] <identifier>] JOIN
<table factor>
```

2.2.14 Syntax rules

In the opening sentence of rules 6 and 7 replace NATURAL by NATURAL or TEMPORAL.

Add

13) If TEMPORAL is specified, TR_1 and TR_2 shall be temporal tables with primary keys whose non-temporal columns are NK_1 and NK_2, and default temporal columns CT_1 and CT_2 respectively. Let *NK* be $NK_1 \cup NK_2$. If the <identifier> is specified let it be *CN*, let *CN* be the name of CT_1. Let *TRL* be the column list of TR_2 with the name of CT_2 replaced by *CN*.

Then the <temporal join> shall be equivalent to

```
   (TR1 INTERLEAVE WITH (SELECT CT2 AS CN FROM TR2))
   NATURAL JOIN
   (TR2 TRL INTERLEAVE WITH (SELECT CT1 FROM TR1))
   FOLD
```

The primary key of the result shall be NK together with CN.

8.1 <predicate>

2.2.15 Format

```
<predicate> ::= …
| <temporal predicate>
```

2.2.16 General rules

Add at the end, or <temporal predicate>.

8.20 <temporal predicate>

2.2.17 Function

Specify a temporal condition that can be evaluated to give a boolean value.

2.2.18 Format

```
<temporal predicate> :: = CURRENT <column reference
list>
| <column reference> AT <value expression>
```

2.2.19 Syntax rules

1) The <column reference>s specified in the AT predicate, or in the <column reference list> of CURRENT, shall be default temporal columns in their respective tables.

2.2.20 General rules

1) Let CL_i be the default temporal column of tables T_i for each i. Then CURRENT(CL_1,..,CL_n) shall be equivalent to CL_1=T_1.LAST(CL_1) AND … CL_n=T_n.LAST(CL_n).

2) Let *CT* be the default temporal column of table *T*. Then *CT* AT *V* shall be equivalent to *V* >= *CT* AND (*CT*=*T*.LAST OR *V*<*T*.NEXT).

10.1 <interval qualifier>

2.2.21 Function

Change to Specify the precision of a date-time or interval data type.

2.2.22 Syntax rules

Change 2) to If TO is specified, then <start field> shall be more significant than <end field>.

2.2.23 Implementation considerations

Although the functions and views proposed in this paper are not very simple to write in SQL, they are efficient to implement in a DBMS.

As noted above the “Table 3” approach to modelling temporal data results in a change to the primary key of the base table, in which a date or timestamp column is added as a final element of the primary key. The DBMS therefore will have available an index which places the values in order; and the partitioning mentioned in this paper will frequently use the previous primary key. As a result, the constructs NEXT and LAST can be made very efficient. This enables CURRENT, AT, FOLD, INTERLEAVE and TEMPORAL JOIN can be made very efficient with very little effort on the part of the DBMS implementer.

With the “Table 3” design, the current values of temporal data can be accessed using views. With the implementation approach discussed in the last paragraph it can be seen that views of the kind described here can be quite efficient.

Above all, the maximal amount of restructuring for temporal data that should be contemplated is the use of “Figure 3” design for some temporal data. This contrasts sharply with the amount of underlying change required to implement either Date’s or Snodgrass’s general machinery, and the difficulty of the concepts involved.

Acknowledgements

It is a pleasure to acknowledge the value of discussions with Thomas Connolly, Junkang Feng and Xi Wang in developing these ideas.

References

[1] Date, C. J. Seminar at the e-Science institute, Edinburgh, November 2005. http://www.nesc.ac.uk/esi/events/601/

[2] Zimanyi, E: Temporal Aggregates and Temporal Universal Quantification in Standard SQL, *SIGMOD Record* **35** (2), June 2006 http://www.sigmod.org/sigmod/record/issues/0606/index.html

[3] Crowe, M.K. *The Pyrrho Database Management System* http://www.pyrrhodb.com, 2006

[4] SQL 2003: ISO/IEC 9075-2:2003 *Information Technology – Database Languages – SQL – Part 2: Foundation* (SQL/Foundation)

[5] Snodgrass, R: *Developing Time-Oriented Applications in SQL*, Morgan Kaufmann, 2000.

[6] Date, C. J., Darwen, H., Lorentzos, M: *Temporal Data and the Relational Data Model*, Morgan Kaufmann, San Francisco, ISBN 1-55860-855-9, 2003

[7] Gabbay, D., McBrien, P.: Temporal Logic & Historical Databases, *Proceedings of the 17th International Conference on Very Large Databases, Bercelona*, p. 423-430, Sept 1991, http://www.vldb.org/conf/1991/P423.PDF