Enabling the Temporal Data Warehouse

By Gregory Sannik,
Principal Consultant,
Teradata Corporation

Fred Daniels,
Senior Consultant,
Teradata Corporation

TERADATA®

Enabling the Temporal Data Warehouse

Table of Contents

Executive Overview

The temporal capabilities in Teradata® Database 13.10 can be considered game changers for both customers who have just begun their temporal journey and those who have been struggling for years with expensive, often inefficient homegrown solutions to managing slowly changing dimension and time series data. This white paper consists of three sections that offer practical advice, templates, and best practices to implement Teradata Database's temporal features for the first time or to upgrade from existing pre-temporal data structures, extract, transform and load (ETL) processes, and data access algorithms.

The first section presents a period data type attribute known as TRANSACTIONTIME, that reflects when data is added to or modified in the database. The second section looks at VALIDTIME, a second time dimension that records when facts are true in the real world as represented in the database. The final section discusses the nature of primary and foreign keys in bi-temporal tables, and the options in Teradata Database 13.10 for implementing entity and referential integrity constraints.

The topics covered include the period data type, temporal qualifiers, characteristics of temporal and bi-temporal tables, data maintenance, data navigation, and enforcement of data integrity. A case study approach will be used to frame best practices within a unified and coherent problem space. Practical examples and suggestions for using the native temporal support delivered in Teradata Database 13.10 will be provided.

Managing Time

Teradata Database 13.10, released in September of 2010, comes with a number of fully integrated, in-database data attributes, qualifiers, and predicates that are extremely useful for automating the management of time-varying data. This release also comes with a number of powerful functions to enable native time series analysis and comparison of periods of time. By optimizing temporal database capabilities for a massively parallel processing platform, this release allows organizations to gather, manage, and analyze "time varying" data more effectively.

Teradata Database 13.10 temporal capabilities rely on two period data type attributes, which are TRANSACTIONTIME and VALIDTIME. In a sense, these are the keys to data warehouse time travel. They allow the reproduction of a report that ran previously, such as six months ago, even though numerous changes have been made to the underlying data. This ability can be vital when responding to inquiries from regulators who want to know what information organizations had and when they had it.

Using TRANSACTIONTIME

A TRANSACTIONTIME column allows the data warehouse to capture changing attribute values in a sequence of versions of data rows. Begin and end timestamp values are maintained automatically and sourced directly from the Teradata Database using temporal qualifiers in data manipulation language (DML) statements.

Figure 1 shows a row instance diagram of a temporal table called Policy_TT that has a TRANSACTIONTIME column called policy_obs_pd. Three rows are currently in the table for two policies. However, only rows 2 and 3 are considered "open," because the second timestamp value of their policy_type_obs column is 9999-12-31 23:59:59.999999, indicating that they are open or correct until the end of time. Row 1 is how Policy 1 looked until 10:09:21.443214 on Aug. 3, 2007, when it was closed, as shown in Row 2.

policy_id	policy_holder_id	policy_type	policy_obs_pd
1	1	'HOME'	('2007-02-19 20:25:00.812367', '2007-08-03 10:09:21.443214')
1	1	'HOME+AUTO'	('2007-08-03 10:09:21.443214', '9999-12-31 23:59:59.999999')
2	2	'AUTO'	('2011-01-05 10:30:00.568900', '9999-12-31 23:59:59.999999')

Figure 1. Temporal table, row instance diagram.

Simplified Processes

Teradata Database 13.10 allows a SQL UPDATE to close existing and insert new open row versions. This lets extract, load and transform (ELT) developers process changed rows without executing both UPDATE and INSERT statements. Here is an example of a multi-statement Update/Insert request:

```
UPDATE Policy_TT
   from Policy_WRK WRK
  set
   policy_type           = WRK.policy_type
   where  Policy_TT.policy_id = WRK.policy_id
    and trans_type = 'Change'
;INSERT into Policy_TT
   (policy_id, policy_holder_id, policy_type )
  select
    policy_id,policy_holder_id, policy_type
   from
    Policy_WRK
   where trans_type in ('Add');
```

Alternatively, the multi-statement request could be re-coded as a single ANSI merge statement. In that case, a performance benefit may be realized by passing through the changed rows only once.

As in previous releases, the preferred method for applying mini-batch updates is through use of a work or staging table with a trans_type column that identifies new rows as "Add" and updates existing rows as "Change" requests. The latest version also offers these advantages:

> Implementation does not require specification of the TRANSACTIONTIME column's high watermark value to confine the scope of the update to open rows. Teradata Database 13.10 will only include open rows in any DML operation.

> The update statement is similar to a more traditional format that is typically used on a non-temporal table. The end value of the existing row's TRANSACTIONTIME is automatically set to the current time by Teradata Database 13.10 to indicate that this version has become obsolete. Succeeding versions of existing rows are inserted automatically as part of the update statement.

> "Insert only" is performed on new rows that need to be added, with the value of the TRANACTIONTIME period populated automatically.

> ELT performance on temporal tables is improved by reducing passes through staging data.

These options offer greater ETL process simplicity, rigor and consistency by centralizing the maintenance of the TRANSACTIONTIME column in the database management system engine rather than embedding it in each and every load script.

Data Navigation Advantages

Users will also find benefits relating to data navigation. For example, SQL SELECT syntax used to access current row instances in a temporal table need not be different from the syntax used to access a non-temporal table because current TRANSACTIONTIME and current VALIDTIME have been implemented as the default session temporal qualifiers in Teradata Database 13.10. This allows the deployment of the temporal capability to proceed with controlled impact on existing application code. Figure 2 shows examples of SQL selects that can be used to ask both current time and past time questions of a temporal table.

Other data navigation features include:

> An "as of" qualifier can be used to apply a single specific transaction time context to all tables participating in the query, or it can be specified on a table-by-table basis for mix-and-match time contexts.

> In general, SQL requests are more succinct and therefore more easily understood than in previous versions. This offers greater opportunity for simplifying semantic layer design by reducing the amount of SQL required for temporal navigation.

Question	SQL Select	Result Set		
Current Query Example	select policy_id, policy_type from Policy_TT; or *CURRENT TRANSACTIONTIME* select policy_id, policy_type from Policy_TT;	Policy_id 1 2	policy_type HOME+AUTO AUTO	
As of Query Example	TRANSACTIONTIME as of TIMESTAMP '2010-06-21 12:21:20.260000-08:00' select policy_id, policy_type from Policy_TT;	Policy_id 1	policy_type HOME+AUTO	
Mix and Match Join Query Example	select polholdr.policy_holder_name, pol.policy_id, pol.policy_type from Policy_TT TRANSACTIONTIME as of TIMESTAMP '2011-01-10 12:21:20.260000-08:00' pol inner join Policy_Holder_TT TRANSACTIONTIME as of TIMESTAMP '2011-01-06 23:59:59.999999-08:00' polholdr on polholdr.policy_holder_id = pol.policy_holder_id;	policy_holder_name Gregory Sannik Fred Daniels	policy_id 1 2	policy_type HOME+AUTO AUTO

Figure 2. Sample SQL select statements.

Maintaining Temporal Tables

Figure 3 shows the data definition language (DDL) for pre- and current Teradata Database 13.10 TRANSACTIONTIME tables called Policy and Policy_TT, which hold information about insurance policies. Structurally, the only difference between the two tables is that Policy requires two timestamp columns while Policy_TT requires a single period data type column called policy_obs_pd defined as TIMESTAMP(6) WITH TIME ZONE. Furthermore, policy_obs_pd has been further defined with 'AS TRANSACTIONTIME' keywords.

Figure 4 shows SQL that can be used to maintain the pre- and current Teradata Database 13.10 tables. Notice that both implementations make use of a work or staging table with a trans_type column that identifies new rows as "Add" requests and updates to existing rows as "Change" requests.

Pre Teradata Database 13.10 Implementation	Teradata Database 13.10 Implementation
CREATE MULTISET TABLE Policy(policy_id INTEGER, policy_holder_id INTEGER, policy_type CHAR(2) NOT NULL, policy_obs_beg_ts *TIMESTAMP(6) WITH TIME ZONE,* policy_obs_end_ts *TIMESTAMP(6) WITH TIME ZONE*) PRIMARY INDEX(policy_id);	CREATE MULTISET TABLE Policy_TT(policy_id INTEGER, policy_holder_id INTEGER, policy_type CHAR(2) NOT NULL, policy_obs_pd *PERIOD* *(TIMESTAMP(6) WITH TIME ZONE) NOT NULL AS* *TRANSACTIONTIME)* PRIMARY INDEX(policy_id);

Figure 3. Pre and current Teradata Database 13.10 DDL.

Pre Teradata Database 13.10 Implementation	Teradata Database 13.10 Implementation

UPDATE Policy
  from Policy_WRK WRK
  set
    policy_obs_end_ts = Current_TimeStamp
  where
    Policy.policy_id = WRK.policy_id
    and trans_type = 'Change'
    and Policy.policy_obs_end_ts = '9999-12-31
23:59:59.999999'
;INSERT into Policy
    (policy_id , policy_holder_id, policy_type,
policy_obs_beg_ts, policy_obs_end_ts)
  select
   policy_id, policy_holder_id, policy_type,   trans_ts, --
policy_obs_beg_ts
  '9999-12-31 23:59:59.999999' --policy_obs_end_ts
  from Policy_WRK
      where trans_type **in** ('Add', 'Change');
	UPDATE Policy_TT
	   from Policy_WRK WRK
	  set
	   policy_type = WRK.policy_type
	   where Policy_TT.policy_id = WRK.policy_id
	    and trans_type = 'Change'
	;INSERT into Policy_TT
	   (policy_id, policy_holder_id, policy_type)
	  select
	    policy_id,policy_holder_id, policy_type
	   from
	    Policy_WRK
	   where trans_type in ('Add');

Figure 4. SQL to maintain the Policy and Policy_TT tables.

Each uses a multi-statement update/insert to "close" the policy observation period of preceding versions and to insert succeeding versions of expired rows as well as brand new rows. Although functionally equivalent, there are some important differences:

> The Teradata Database 13.10 implementation does not require specification of the TRANSACTIONTIME column high water-mark value to confine the scope of the update to open rows.

> The update statement looks like a traditional update statement on a non-temporal table. The end value of the TRANSACTIONTIME column is automatically set to the current time by the database management system (DBMS) to indicate that this version has become obsolete.

> The Teradata Database 13.10 implementation performs the "insert only" on new rows that need to be added. The insert of new versions of existing rows is handled automatically as part of the update statement. This statement could also have been coded as an ANSI merge statement.

The Teradata Database 13.10 implementation that encapsulates the TRANSACTIONTIME column update offers greater rigor and consistency in data versioning. In addition, it provides a potential performance benefit by passing through the changed rows only once.

Querying Temporal Tables

As already stated, Teradata Database 13.10 also offers benefits to temporal data warehouse architecture from a data access perspective including the ability to encapsulate a table's temporal properties. For example, the SQL SELECT syntax to access current row instances of a temporal table does not need to differ from that used to access a non-temporal table. Because of this, deployment of the temporal capability can proceed with controlled impact on existing application code.

Figures 5 and 6 show row instance diagrams for the two TRANSACTIONTIME tables, Policy_TT and Policy_Holder_TT. The pre-Teradata Database 13.10 table definitions implement the policy and policy holder observation periods as period data types each holding begin and end timestamp(6) values.

policy_id	policy_holder_id	policy_type	policy_obs_pd
1	1	'HOME'	('2007-02-19 20:25:00.812367', '2007-08-03 10:09:21.443214')
1	1	'HOME+AUTO'	('2007-08-03 10:09:21.443214', '9999-12-31 23:59:59.999999')
2	2	'AUTO'	('2011-01-05 10:30:00.568900', '9999-12-31 23:59:59.999999')

Figure 5. Contents of Policy_TT.

policy_holder_id	policy_holder_name	policy_obs_pd
1	'Greg Sannik'	('2007-02-19 20:25:00.812367', '2007-08-03 10:09:21.443214')
1	'Greg Sannik'	('2007-08-03 10:09:21.443214', '9999-12-31 23:59:59.999999')
2	'Fred Daniels'	('2011-01-05 10:30:00.568900', '9999-12-31 23:59:59.999999')

Figure 6. Contents of Policy_Holder_TT.

Question	Pre-Teradata Database 13.10 Implementation	Teradata Database 13.10 Implementation	Result Set

Current Query Example	select policy_id, policy_type
	from Policy
	where
	policy_obs_end_ts = '9999-12-31 23:59:59.999999';
		select policy_id, policy_type
		from Policy_TT; or
		
		CURRENT TRANSACTIONTIME
		select policy_id, policy_type
		from Policy_TT;
			Policy_id policy_type
			
			1 HOME+AUTO
			2 AUTO

As of Query Example	select policy_id, policy_type from Policy where TIMESTAMP '2010-06-21 12:21:20.260000-08:00' >= policy_obs_beg_ts and TIMESTAMP '2010-06-21 12:21:20.260000-08:00' < policy_obs_end_ts;	TRANSACTIONTIME as of TIMESTAMP '2010-06-21 12:21:20.260000-08:00' select policy_id, policy_type from Policy_TT;	Policy_id policy_type 1 HOME+AUTO
Mix and Match Join Query Example	**select** polholdr.policy_holder_name, pol.policy_id, pol.policy_type **from** Policy pol **inner join** Policy_holder polholdr on polholdr.policy_holder_id = pol.policy_holder_id **where** '2011-01-10 12:21:20.260000-08:00' >= pol.policy_obs_beg_ts **and** '2010-01-10 12:21:20.260000-08:00' < pol.policy_obs_end_ts **and** '2011-01-06 23:59:59.999999-08:00' >= polholdr.Policy_holder_obs_beg_ts **and** '2011-01-06 23:59:59.999999-08:00' < polholdr.Policy_holder_obs_end_ts ;	select polholdr.policy_holder_name, pol.policy_id, pol.policy_type from Policy_TT TRANSACTIONTIME as of TIMESTAMP '2011-01-10 12:21:20.260000-08:00' pol inner join Policy_Holder_TT TRANSACTIONTIME as of TIMESTAMP '2011-01-06 23:59:59.999999-08:00' polholdr on polholdr.policy_holder_id = pol.policy_holder_id;	policy_holder_name policy_id policy_type Gregory Sannik 1 HOME+AUTO Fred Daniels 2 AUTO

Figure 7. Business questions with temporal focus.

Figure 7 shows SQL SELECT statements that support business questions having a "current" and "as of" temporal focus on the pre- and current Teradata Database 13.10 versions of the tables. There are several things worth noting:

> Queries that have a current temporal focus do not need a temporal statement qualifier.

> A Teradata Database 13.10 "AS OF" qualifier can be used to apply a single specific transaction time context to all tables participating in the query. Alternatively, "AS OF" qualifiers can be specified on a table-by-table basis in the from clause, thereby providing a means to mix and match temporal contexts.

> In general, the Teradata Database 13.10 SQL requests tend to be more succinct than requests in previous versions. In the past, "AS OF" queries, particularly those involving joins between multiple tables, were especially verbose because the temporal context needed to be restated for each temporal table participating in the query.

Adding the VALIDTIME Dimension

Let's now consider the scenario of a table that records information about insurance policies, and notes when attributes such as coverage amount and policy type were in effect. The values of these attributes could be different at different times, and knowing the values that were applicable at each point in time is critical in processing claims that may be filed much later.

In Teradata Database 13.10, a period data type is used to store this temporal information, and would be given a VALIDTIME attribute in the DDL. Unlike TRANSACTIONTIME periods that must be defined as TIMESTAMP(6) WITH TIMEZONE, VALIDTIME periods can be defined using DATEs or TIMESTAMPs, with or without TIMEZONE, and with different precision to the right of the decimal point. Also, unlike TRANSACTIONTIME columns which are maintained entirely by the database, the period of validity of each row must come from the business context. Some VALIDTIME tables may have rows whose attribute values only apply to a specific period. In other cases, the end of the period of validity may not be known in advance so it is considered good till the end of time. In these cases, a value of '9999-12-31' is used to reflect that.

The rows in the VALIDTIME table can be divided into current, history and future rows, depending on the relationship between the current date/time and the period of validity of each row. Below are row instance diagrams of two VALIDTIME tables called Policy_VT and Policy_Holder_VT.

Policy_VT has a VALIDTIME column called policy_buseff_pd. As of December 31, 2010, there are six rows in the Policy_VT table for three different policies (See Figure 8).

policy_id	policy_holder_id	policy_type	policy_buseff_pd
1	1	HOME	('2010-02-19', '2010-08-24')
1	1	HOME+AUTO	('2010-08-24', '9999-12-31')
2	2	AUTO	('2010-12-31', '9999-12-31')
2	2	AUTO	('2010-08-03', '2010-10-05')
2	2	AUTO+BOAT	('2010-10-05', '2010-12-31')
3	3	HOME+AUTO	('2011-01-15', '9999-12-31')

Figure 8. Contents of Policy_VT.

These rows would be categorized from a VALIDTIME perspective as follows on December 31, 2010:

> Rows 1 and 4 are history rows since the ends of the policy_buseff_pd periods are both less than December 31, 2010.

> Rows 2 and 3 are current rows because the beginning of the policy_buseff_pd periods is less than or equal to December 31, 2010, and end dates are '9999-12-31'.

> Row 5 is about to become a history row since its end date is '2010-12-31'

> Row 6 is a future row since the beginning of the policy_buseff_pd is greater than December 31, 2010.

Policy_Holder_VT also has a VALIDTIME column called policy_holder_buseff_pd. As of December 31, 2010, there are 4 rows in the Policy_Holder_VT table for three different policy holders.

> Row 2 is a history row since the end of the policy_holder_buseff_pd period is '2007-08-03 which is less than December 31, 2010.

> Rows 1 and 4 are considered current rows because the beginning of the policy_buseff_pd periods is less than December 31, 2010, and the end dates are '9999-12-31'.

> Row 3 is a future row since the beginning of the policy_holder_buseff_pd period is '2011-01-05' which is greater than December 31, 2010.

The Teradata Database 13.10 CREATE TABLE syntax to support our two VALIDTIME temporal tables is:

```
CREATE MULTISET TABLE Policy_VT(
policy_id INTEGER,
policy_holder_id INTEGER,
policy_type CHAR(8) NOT NULL,
policy_buseff_pd PERIOD (DATE) NOT NULL AS VALIDTIME
)
PRIMARY INDEX(policy_id);
CREATE MULTISET TABLE Policy_Holder_VT, NO FALLBACK,
   NO BEFORE JOURNAL,
   NO AFTER JOURNAL,
   CHECKSUM = DEFAULT,
   DEFAULT MERGEBLOCKRATIO
   (
   policy_holder_id INTEGER,
   policy_holder_name VARCHAR(30) CHARACTER SET
   LATIN NOT CASESPECIFIC NOT NULL,
   policy_holder_buseff_pd PERIOD(DATE) NOT NULL AS
   VALIDTIME
)
PRIMARY INDEX ( policy_holder_id );
```

(Notice that the VALIDTIME columns are defined at a date grain.)

policy_holder_id	policy_holder_name	policy_holder_buseff_pd
1	Greg Sannik	('2007-08-03', '9999-12-31')
1	Greg Sannik	('2007-02-19', '2007-08-03')
2	Fred Daniels	('2011-01-05', '9999-12-31')
3	Corky Sannik	('2010-08-03', '9999-12-31')

Figure 9. Contents Policy_Holder_VT.

Navigating a VALIDTIME Temporal Table

Just as a TRANSACTIONTIME column can be used to reflect a piece of information as it was recorded at an earlier point in time in the data warehouse, a VALIDTIME column can be used to reflect information applicable to different points in time in the business world including those in the future. Once again, the SQL SELECT syntax to access current row instances of a VALIDTIME table need not differ from those of a non-temporal table based on interrogation of a table's VALIDTIME column because its temporal qualifier is current.

Figure 10 shows examples of SQL SELECT statements used to query our two VALIDTIME temporal tables previously described.

The queries have been structured to answer business questions having either a current, as of, or a sequenced temporal focus. The resulting answer sets are shown for each question assuming that the selects are executed on December 31, 2010.

Question	SQL Statement	Result Set			
Current Query Example	select policy_id, policy_type from Policy_VT; or CURRENT VALIDTIME select policy_id, policy_type from Policy_VT; or	policy_id 1 2		policy_type HOME+AUTO AUTO	
As of Query Example	VALIDTIME as of DATE '2010-06-21' select policy_id, policy_type from Policy_VT;	Policy_id 1		policy_type HOME+AUTO	
Mix and Match Join Query Example	select polhold.policy_holder_name, pol.policy_id, pol.policy_type from Policy_VT VALIDTIME as of DATE '2011-01-15' pol inner join Policy_Holder_VT VALIDTIME as of DATE '2011-01-06' polhold on polhold.policy_holder_id = pol.policy_holder_id order by pol.policy_id;	policy_holder_name Gregory Sannik Fred Daniels Corky Sannik	policy_id 1 2 3	policy_type HOME+AUTO AUTO+BOAT HOME+AUTO	
Sequenced ValidTime Join Example	SEQUENCED VALIDTIME select polhold.policy_holder_name, pol.policy_id, pol.policy_type from Policy_VT pol inner join Policy_Holder_VT polhold on polhold.policy_holder_id = pol.policy_holder_id order by pol.policy_id;	policy_holder_name Gregory Sannik Gregory Sannik Fred Daniels Corky Sannik	policy_id 1 1 2 3	policy_type HOME HOME+AUTO AUTO HOME+AUTO	VALIDTIME ('2010-02-19', '2010-08-24') ('2010-08-24', '9999-12-31') ('2011-01-05', '9999-12-31') ('2011-01-15', '9999-12-31')

Figure 10. VALIDTIME temporal table data access.

There are a few things worth noting about the examples in Figure 10.

> There are strong similarities between data navigation of VALIDTIME and TRANSACTIONTIME tables as described in the prior section of this paper.

> Since the default session VALIDTIME qualifier is current, an explicit qualifier is not needed when the rows requested are current. As was the case with a TRANSACTIONTIME table, it is easy to insulate existing applications and users from the VALIDTIME temporal dimension of a table if they are not able or not interested in seeing the past or future.

> Different date or timestamp values can be used on each VALIDTIME table to mix and match temporal orientation of the query.

> A sequenced VALIDTIME query results in a VALIDTIME result set meaning the VALIDTIME column is exposed. The VALIDTIME period for each row is given the column heading 'VALIDTIME' and is the overlap of the Policy_VT row's valid-time and its associated Policy_Holder_VT row's valid-time. Notice also that rows for all policies are returned including a Policy 3's future row.

Data Maintenance

As in the case of a TRANSACTIONTIME update described in section one, updates to VALIDTIME tables are applied via a work table called Policy_VT_Wrk though a multi-statement request:

```
SEQUENCED VALIDTIME
DELETE Policy_VT
   from Policy_VT_WRK WRK
  where
    POLICY_VT.policy_id = WRK.policy_id
    and POLICY_VT.policy_holder_id =
WRK.policy_holder_id
    and WRK.trans_type = 'Change'
;NONSEQUENCED VALIDTIME INSERT into Policy_VT
   (policy_id, policy_holder_id, policy_buseff_pd, pol-
icy_type )
  select
    policy_id, policy_holder_id, policy_buseff_pd,
policy_type
   from
    Policy_VT_WRK
   Where trans_type in ('Add', 'Change');
```

However, unlike the TRANSACTIONTIME table example that used a simple, multi-statement UPDATE/INSERT, this request first performs a SEQUENCED VALIDTIME DELETE followed by a NONSEQUENCED VALIDTIME insert. The keywords SEQUENCED and NONSEQUENCED are temporal qualifiers. The SEQUENCED VALIDTIME qualifier is used in conjunction with the DELETE to qualify rows from the Policy table in the VALIDTIME dimension.

The NONSEQUENCED VALIDTIME qualifier is necessary for the INSERT statement because we are providing the values of the inserted rows from a SELECT statement that references the VALIDTIME column from the Policy_VT_WRK table. Had we provided the value for policy_buseff_pd via a variable or literal such period '(2010-10-05, 2010-12-31)', a SEQUENCED VALIDTIME qualifier would have worked.

Although it initially may appear unusual, a DELETE is required instead of the familiar UPDATE operator to expire the prior policy_buseff_pd and automatically create one or more revised periods of validity.

Impact of Updating a VALIDTIME Table

Figure 11 depicts the impact of updating the period of validity of a VALIDTIME temporal table for two fictitious insurance policies using a SEQUENCED DELETE/NONSEQUENCED INSERT statement. The intent of these temporal updates is to reflect the following results:

1. Policy 1's policy_type is changed from 'HOME' to 'HOME+ AUTO' to commence on August 3, 2010, valid till the end of time as indicated by use of the keyword UNTIL_CHANGED that translates into '9999-12-31' and
2. Policy 2's policy_type is changed from 'AUTO' to 'AUTO+ BOAT' from October 5, 2010, to December 30, 2010, and then is changed back to 'AUTO' on December 31, 2010, valid till the end of time perhaps resulting from the policy holder's discovery that a boat was more trouble than it was worth.

The DELETE/INSERT request is used in favor of an UPDATE/INSERT because in addition to updating a policy's type, we are attempting to end the period of validity of an existing row and create new row(s) with different periods of validity.

In the case of Policy 1, the SEQUENCED VALIDTIME DELETE request sets the original row's policy_buseff_pd end timestamp to '2010-08-24'. The NONSEQUENCED VALIDTIME INSERT is responsible for the creation of the second instance of Policy 1.

Contents of policy table before temporal update

policy_id	policy_holder_id	policy_type	policy_buseff_pd
1	1	HOME	('2010-02-19', '9999-12-31')
2	2	AUTO	('2010-08-03', '9999-12-31')

Execute temporal update (DELETE – INSERT)

policy_id	policy_holder_id	policy_type	policy_buseff_pd	trans_type
1	1	HOME+AUTO	('2010-08-03', 'UNTIL CHANGED')	Change
2	2	AUTO+BOAT	('2010-10-05', '2010-12-31')	Change

Contents of policy table after temporal update

policy_id	policy_holder_id	policy_type	policy_buseff_pd
1	1	HOME	('2010-02-19', '2010-08-24')
1	1	HOME+AUTO	('2010-08-24', '9999-12-31')
2	2	AUTO	('2010-08-03', '2010-10-05')
2	2	AUTO+BOAT	('2010-10-05', '2010-12-31')
2	2	AUTO	('2010-12-31', '9999-12-31')

Figure 11. Impact of updating the period of validity of a VALIDTIME temporal table for two fictitious insurance policies.

In the case of Policy 2, the DELETE did a couple of things. First, it set the end timestamp of the original row to '2010-10-05'. Second, it created another 'Auto' policy_type row with policy_buseff_pd starting '2010-12-31' valid through the end of time or '9999-12-31'.

The NONSEQUENCED VALIDTIME INSERT is responsible for another new row that reflected a policy type change from 'AUTO' to 'AUTO+BOAT' effective '2010-10-05' thru '2010-12-30'. Note that this row is effective from '2010-10-05' thru '2010-12-30' and not thru '2010-12-31' because period data type columns use an inclusive-exclusive representation to define time ranges.

These two policy update scenarios illustrate examples of what Dr. Richard T. Snodgrass, professor of Computer Science at the University of Arizona, calls 'overlap' and 'during' period operations. These are two of the possible operations for modifying the business periods held in a VALIDTIME column. Additional details of period operations can be found in his book *Developing Time-Oriented Database Applications in SQL*.

Possible Outcomes of a VALIDTIME Table Update

According to Snodgrass, when a change is made to a row in a VALIDTIME table, multiple events may have to occur, depending on the relationship between the period of validity (PV) of the old version of the row and the period of applicability (PA) of the new information. In the past each possible event had to be coded separately in multiple lines of complex, error-prone SQL statements. With the advent of Teradata Database 13.10, the application developer can rely on powerful sequenced DML statements that move the complex decision and processing sequence into the database engine.

Figure 12 shows before and after images that result from sequenced updates (left) and sequenced deletes (right). It depicts five possible relationships between the PA of the new information (red) and PV of the old information (blue): PA precedes PV, PA overlaps with the beginning of PV, PA is contained in PV, PA

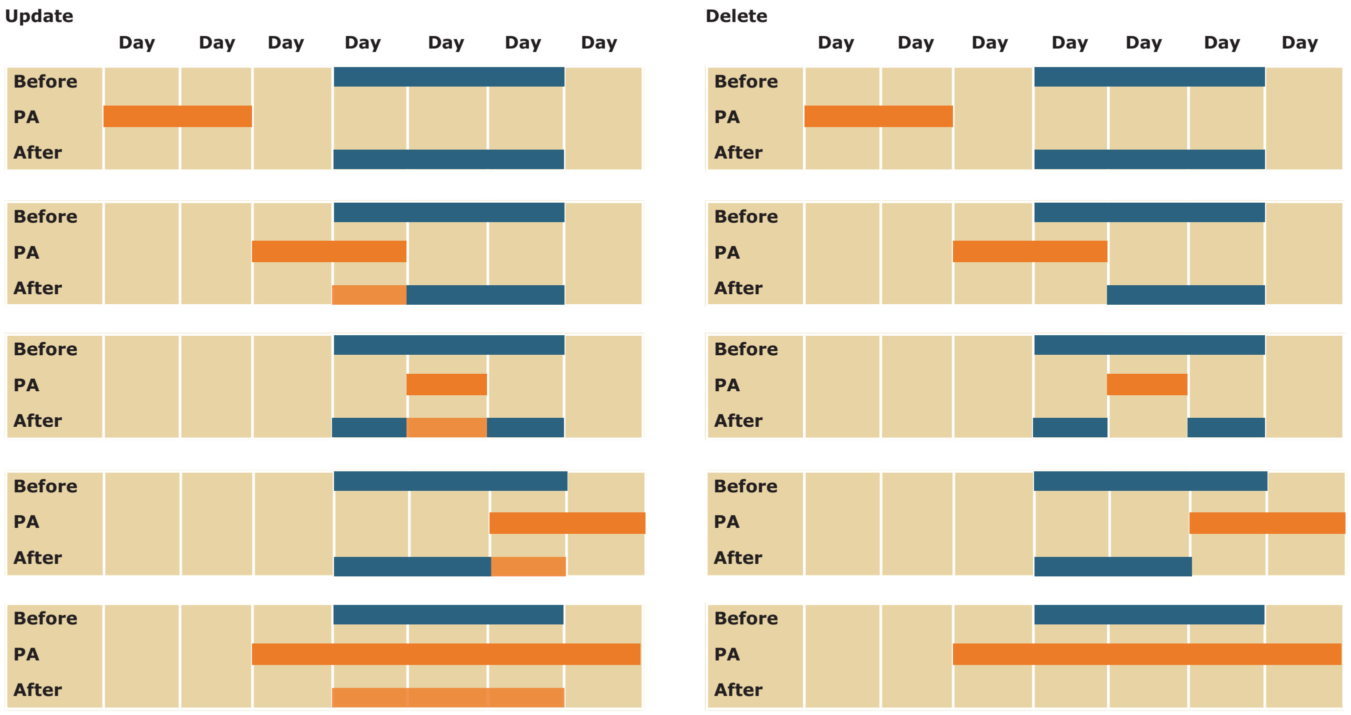

Figure 12. Sequenced update (left) and delete (right) with a specified period of applicability (PA).

overlaps with the end of PV, and PA contains PV. It should be noted that in each case any parts of the PA that do not overlap with the PV are ignored. An important concept to grasp when designing jobs to maintain VALIDTIME tables is that in contrast with non-temporal tables, an update is not necessarily identical to a delete followed by an insert.

For a non-temporal table the preferred practice is to use a single update rather than a delete followed by an insert. However, for a VALIDTIME table, the update may not produce the complete result. Therefore, a two-step delete/insert process is the recommended approach for maintaining a VALIDTIME table.

To summarize our consideration of VALIDTIME columns, they offer even greater capability in tracking the validity of information as it is known to the business at various points of time. They share some, but not all characteristics of TRANSACTIONTIME columns. Also, they require a somewhat different approach to applying temporal updates than do other technologies and pre-13.10 releases of Teradata Database. Although it may initially be unfamiliar, it makes sense after considering the properties of a VALIDTIME column that go beyond a simple non-key attribute of a table.

Characteristic	TRANSACTIONTIME Column	VALIDTIME Column
Use/meaning	When the data warehouse first became aware of a piece of information, and when it is considered to be expired	When the business considers a piece of information to have been valid.
Data Type	Period	Period
Number Allowed in a Table	One	One
Permitted Temporal Grains	(TIMESTAMP(6) WITH TIME ZONE)	Date, Timestamp(0) through Timestamp(6) with or without time zone
Nullability	Not Null	Both Null and Not Null permitted
Sample Syntax	TT_Column PERIOD (TIMESTAMP(6) WITH TIME ZONE) NOT NULL AS TRANSACTIONTIME	VT_Column PERIOD(DATE) NOT NULL AS VALIDTIME
How Values Sourced	Teradata DBMS Engine	Business World
How Maintained	Completely Automated	Part Manual/Part Automated
End Boundary Function Literal	UNTIL_CLOSED	UNTIL_CHANGED
Default Session Temporal Qualifier	CURRENT	CURRENT

Figure 13. Compare and contrast of VALIDTIME and TRANSACTIONTIME columns.

Defining Primary and Foreign Key Constraints in a Temporal Data Warehouse Architecture

Relational database systems use primary keys to uniquely identify rows in a table and enforce entity integrity. A row in a child table that refers to a row in a parent table will include the primary key column(s) of the parent together with a foreign key constraint in the DDL to enforce referential integrity. These concepts are familiar in the context of non-temporal tables, but they need to be generalized when applied to temporal tables.

Let's consider the simple case of a TRANSACTIONTIME table that may contain multiple versions of the same row. Only one version of a row can be open at a time, but there can be multiple closed versions of a row. A uniqueness constraint on a TRANSACTIONTIME table requires a current TRANSACTIONTIME temporal qualifier. This constraint will ignore all the logically deleted closed rows and only be enforced against the open rows. Thus TRANSACTIONTIME primary key (PK) and foreign key (FK) constraints are very similar to the corresponding constraints on non-temporal tables.

VALIDTIME tables are more complicated because there may be more than one row with the same PK values with different periods of validity for the non-key attributes. Foreign key relations between VALIDTIME tables must require that one or more parent rows exist whose period(s) of validity covers the period of validity of the child row.

Defining Temporal Primary Keys

There are many options in defining a primary key constraint on a temporal or bi-temporal table. For instance the primary key can include specification of the TRANSACTIONTIME and/or VALIDTIME column. Any of the three temporal qualifiers found in Figure 14 can be used to set the constraint's VALIDTIME temporal scope. However, the TRANSACTIONTIME qualifier must always be set to current.

Figure 15 shows the syntax that can be used to define a primary key on a TRANSACTIONTIME, a VALIDTIME and a bi-temporal table using modified Policy_Holder_TT, Policy_Holder_VT table definitions from the first two sections of the this paper as well as a new definition for a bi-temporal table called Policy_Holder_VT_TT.

For simplicity, CURRENT TRANSACTIONTIME and CURRENT VALIDTIME qualifiers have been used. As stated earlier, there were other temporal combinations possible, although this scenario might be considered one of the more common.

Primary Key Qualifier	Implication/Consideration
None	
CURRENT VALIDTIME	Current or future rows that have overlapping periods can not have the same key value
SEQUENCED VALIDTIME	Current, future or history rows that have overlapping periods can not have the same key value
NON-SEQUENCED VALIDTIME	No two rows, irrespective of temporal state, can have the same key value

Figure 14. Temporal table primary key qualifiers.

TRANSACTIONTIME Table	VALIDTIME Table	Bi-temporal Table
CREATE MULTISET TABLE Policy_Holder_TT (policy_holder_id INTEGER NOT NULL, policy_holder_name VARCHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL, policy_holder_obs_pd PERIOD (TIMESTAMP(6) WITH TIME ZONE) NOT NULL AS TRANSACTIONTIME, CONSTRAINT XPK_Policy_Holder_TT CURRENT TRANSACTIONTIME PRIMARY KEY (policy_holder_id)) PRIMARY INDEX(policy_holder_id);	CREATE MULTISET TABLE Policy_Holder_VT (policy_holder_id INTEGER NOT NULL, policy_holder_name VARCHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL, policy_holder_buseff_pd PERIOD (DATE) NOT NULL AS VALIDTIME , CONSTRAINT XPK_Policy_Holder_VT CURRENT VALIDTIME PRIMARY KEY (policy_holder_id)) PRIMARY INDEX(policy_holder_id);	CREATE MULTISET TABLE Policy_Holder_VT_TT (policy_holder_id INTEGER NOT NULL, policy_holder_name VARCHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL, policy_holder_buseff_pd PERIOD (DATE) NOT NULL AS VALIDTIME , policy_holder_obs_pd PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL AS TRANSACTION- TIME, CONSTRAINT XPK_Agreement CUR- RENT VALIDTIME AND CURRENT TRANSACTIONTIME PRIMARY KEY (policy_holder_id)) PRIMARY INDEX(policy_holder_id);

Figure 15. Examples of primary key constraint definitions.

Primary keys defined on temporal tables are implemented as system-generated join indexes. Here is an example of the resulting system-generated join index as a result of defining the primary key constraint on Policy_Holder_VT_TT.

```
 CREATE SYSTEM_DEFINED JOIN INDEX
Policy_Holder_VT_TT_TJI004
,NO FALLBACK,CHECKSUM = DEFAULT AS CURRENT
VALIDTIME AND CURRENT TRANSACTIONTIME
SELECT
   Policy_Holder_VT_TT.ROWID
   ,Policy_Holder_VT_TT.policy_holder_id
   ,Policy_Holder_VT_TT.policy_holder_buseff_pd
   ,Policy_Holder_VT_TT.policy_holder_obs_pd
FROM Policy_Holder_VT_TT
   PRIMARY INDEX ( policy_holder_id );
```

Notice the new keyword SYSTEM_DEFINED and the ROWID column that supports a join back to the rows of the base Policy_Holder_VT_TT table. Also, the join index has a non-unique as opposed to a unique primary index. The reason for this is because both the SEQUENCED and CURRENT temporal qualifiers do not prevent rows irrespective of temporal state from having the same key value. If a NONSEQUENCED temporal qualifier had been used, this would have been prevented and would have caused the system-generated join index to be defined with a unique primary index.

One final note, PK constraints can either be defined when the table is initially created or later via an ALTER TABLE statement. Since they are system generated, they are automatically dropped when either the constraint or the table is dropped.

Enabling the Temporal Data Warehouse

Defining Temporal Foreign Keys

Just as referential constraints can be defined between a non-temporal child and parent table, Teradata Database 13.10 offers the ability to define referential constraints between temporal tables or between a non-temporal child and a temporal parent.

As you might imagine, given the possible VALIDTIME and TRANSACTIONTIME temporal qualifier combinations for defining a primary key on a temporal table, there is an even greater number of possible combinations for defining a referential integrity (RI) constraint between two temporal tables.

Figure 16 shows options for defining foreign key constraints on VALIDTIME and TRANSACTIONTIME.

Currently, hard RI constraints are not supported on temporal tables. The implication is that the ETL developer and Data Quality Analyst are responsible for maintaining referential integrity between tables. Appendix D of the Temporal Support Manual provides SQL code that can be used to validate temporal referential integrity.

		Parent Table Type			
		NT	**VT**	**TT**	**VTTT**
Child Table Type	**NT**	RRI	TRC	RRI	TRC on open parent rows
	VT	NRI in VT dimension	CRI, SRI in VT dimension	Invalid	CRI, SRI in VT dimension
	TT	NRI in TT dimension	TRC, NRI in TT dimension	CRI, SRI in TT dimension	TRC, CRI/SRI in TT dimension
	VTTT	NRI in both the dimensions	CRI, SRI in VT dimension, NRI in TT dimension	NRI in VT dimension, CRI, SRI in TT dimension	CRI, SRI in both the dimensions

NT – Non temporal table
VT – ValidTime table
TT – TransactionTime table
TT – TransactionTime table
VTTT – Table with both dimensions
CRI – CURRENT RI
SRI – SEQUENCED RI
NRI – NONSEQUENCED RI
RRI – Regular RI (i.e., no temporal semantics applicable
TRC – Temporal Relationship Constraint

Figure 16. Options for defining RI constraints on temporal tables.

Here is an example of the syntax used to define a SEQUENCED VALIDTIME and CURRENT TRANSACTIONTIME foreign key from the Policy_VT_TT table to the Policy_Holder_VT_TT tables.

```
ALTER TABLE Policy_VT_TT
ADD  CONSTRAINT Policy_HOLDER_VT_TT_FK
CURRENT TRANSACTIONTIME AND SEQUENCED VALIDTIME
FOREIGN KEY (Policy_Holder_Id)
REFERENCES WITH NO CHECK OPTION
Policy_Holder_VT_TT (Policy_Holder_Id);
```

Other than the VALIDTIME and TRANSACTIONTIME clause, this resembles a constraint between two non-temporal tables. In this particular case, a SEQUENCED VALIDTIME and CURRENT TRANSACTIONTIME qualifier was used.

So far we have described how to define relationships between temporal tables. It is also possible to define a temporal referential constraint between a non-temporal child and a temporal parent like between a fact table and a slowly changing dimension. This type of constraint is called a Temporal Relationship Constraint (TRC).

A TRC is a special form of temporal constraint defined between a non-temporal table with a date or time column and a temporal table that can be either a VALIDTIME (VT) or a bi-temporal table. A common use case would be to support join elimination from a large event or transaction table to its lookup tables that translate text into code values for row filtering and code values into text for answer set display.

Soft-RI/Join Elimination

The Teradata Database feature known as Soft-RI/Join elimination has been available for many years. However, it may be one of the least understood aspects of the Teradata Database. First, it is the basis for eliminating table joins in views. These joins can be defined in support of traditional one-to-many parent-child table relationships, as well as a one-to-one sibling relationship among two or more vertical partitions of what otherwise would be a very wide table. Second, RI constraints can also be used to encourage greater optimizer aggressiveness with query plans and even help increase the chances of join index use.

Unfortunately prior to Teradata Database 13.10, RI constraints could not be defined on the 'begin' and 'end' columns of slowly changing dimensions (SCD). More specifically, the syntax neither supported the specification that a date or timestamp in a fact table was between the 'begin' and 'end' columns of the SCD (inclusive-inclusive constraint) nor date or timestamp column in a fact table was greater than or equal to the 'begin' and less than the 'end' column of the SCD as an inclusive-exclusive constraint. This all changes as a result of Teradata Database 13.10 temporal features. Using this release, temporal-based soft RI can now be defined, thereby enabling join elimination.

There are differences in the way join elimination takes place between a child table that can be either non-temporal, temporal, or bi-temporal and a temporal or bi-temporal parent or between a similar child table and a non-temporal parent. These differences are reflected in the explain plan when join elimination takes place. When the parent table is non-temporal, the absence of references to the parent table indicates that join elimination will occur. When the parent table is either temporal or bi-temporal, reference to the system-generated join index indicates that join elimination will occur.

This view executes two explain requests on two select statements. The first references the policy_holder_name column from policy_holder. The second does not reference any columns from the policy_holder table.

```
create view Policy_Policy_Holder as
(current VALIDTIME select
pol.policy_id,
pol.policy_holder_id,
pol.policy_type,
polhldr.policy_holder_name
from
Policy_VT_TT pol,
Policy_HOLDER_VT_TT polhldr
where polhldr.policy_holder_id = pol.policy_holder_id);
```

Scenario	Join Elimination of Policy_Holder_VT_TT	No Join Elimination of Policy_Holder
Select	select policy_type from Policy_Policy_Holder	Select policy_holder_name from Policy_Policy_Holder
Explain Statement	… 2) Next, we lock a distinct "pseudo table" for read on a RowHash to prevent global deadlock for POLICY_HOLDER_VT_TT_TJI004. 3) We lock pol in view Policy_Policy_Holder for read, and we lock POLICY_HOLDER_VT_TT_TJI004 in view Policy_Policy_Holder for read. … 5) We do an all-AMPs JOIN step from POLICY_HOLDER_VT_TT_TJI004 in view Policy_Policy_Holder by way of a RowHash match scan with a condition of ("((END(POLICY_HOLDER_VT_TT_TJI004 in view Policy_Policy_Holder.policy_holder_buseff_pd))>= DATE '2011-04-25') AND ((BEGIN(POLICY_HOLDER_VT_TT_TJI004 in view Policy_Policy_Holder.policy_holder_buseff_pd))<= DATE '2011-04-24')"), which is joined to Spool 3 (Last Use) by way of a RowHash match scan. POLICY_HOLDER_VT_TT_TJI004 and Spool 3 are joined using a merge join, with a join condition of ("POLICY_HOLDER_VT_TT_TJI004.policy_holder_id = policy_holder_id"). …	… 2) Next, we lock a distinct "pseudo table" for read on a RowHash to prevent global deadlock for polhldr. 3) We lock pol in view Policy_Policy_Holder for read, and we lock polhldr in view Policy_Policy_Holder for read. … 5) We do an all-AMPs JOIN step from polhldr in view Policy_Policy_Holder (with temporal qualifier as "CURRENT VALIDTIME AND CURRENT TRANSACTIONTIME") by way of a RowHash match scan with a condition of ("((BEGIN(polhldr in view Policy_Policy_Holder.policy_holder_buseff_pd))<= DATE '2011-04-24') AND (((END(polhldr in view Policy_Policy_Holder.policy_holder_obs_pd))= TIMESTAMP '9999-12-31 23:59:59.999999+00:00') AND ((END(polhldr in View Policy_Policy_Holder.policy_holder_buseff_pd))>= DATE '2011-04-25'))"), which is joined to Spool 3 (Last Use) by way of a RowHash match scan. polhldr and Spool 3 are joined using a merge join, with a join condition of ("polhldr.policy_holder_id = policy_holder_id"). …

Figure 17. Comparison of explains with and without temporal join elimination.

Figure 17 provides a side-by-side comparison of selected steps from the two explains. The intent is to focus on those steps that indicate whether join elimination is going to occur.

As you can see, the system-generated join index called POLICY_HOLDER_VT_TT_TJI004 plays a prominent role in the join plan on the left, replacing access to the POLICY_HOLDER_VT_TT table that occurs in the non-elimination scenario on the right side of Figure 17. This is also noteworthy since it is different from an explain where join elimination occurs between a child table and a non-temporal parent. In that case the only evidence that join elimination will occur is the absence of any references to the parent table in the explain text or absence of the parent table in the DBQLOBJTBL after execution.

The ability to define primary key constraints on temporal tables offers numerous opportunities for automatically ensuring row uniqueness without costly post-ETL primary key checks. In the current Teradata Database release, validating referential integrity remains the responsibility of the ETL developer/QA analyst. However the support for join elimination between temporal or bi-temporal tables will provide significant performance benefits when accessing complex, join views. These features will benefit customers who have not previously implemented temporal data warehouses, as well as those who have done so using non-temporal SQL.

Conclusion

The temporal capabilities in Teradata Database 13.10 offer several advantages to organizations that need to track history in their data warehouse:

> Improves consistency and efficiency in complying with external regulatory requirements.

> Provides stronger, more centralized control of reference data versioning.

> Protects against data loss and outages caused by accidental or malicious DELETE ALL FROM TABLE requests since a temporal delete expires rather than removes rows.

> Enables ANSI Merge operator for temporal updates, even if target table is partitioned on the end points of VALIDTIME and/or TRANSACTIONTIME columns.

> Optimizes query performance through soft-RI/join elimination and join indexing for bi-temporal tables.

> Simplifies semantic layer design as a result of the reduced amount of SQL required for temporal navigation.

> Ensures integrity of temporal data through proper, time aware primary key constraints.

Whether organizations are using temporal data for the first time or have been working with it for years, Teradata Database 13.10 can offer an effective means to gather, manage and analyze data that changes over time.

Ten Steps to Start Using Temporal Capabilities

To begin using Teradata Database 13.10 temporal capabilities:

1. Research temporal database functionality by downloading the Teradata Database 13.10 Temporal Table Support Manual on Teradata.com.
2. Order the Teradata Express evaluation and development software to experiment with period data types on Teradata.com, or visit the Teradata Developer Exchange at http://developer.teradata.com.
3. Determine your organization's temporal support needs from both regulatory and business intelligence (BI) perspectives.
4. Determine the volatility of reference data.
5. Understand the abilities, limitations and costs of the current architecture to support a temporal requirement.
6. Develop a design that supports reference data versioning (at least as an exercise).
7. Take inventory of the temporal mechanisms in the enterprise resource planning and online transaction processing environments.
8. Conduct a data profiling initiative to assess the business date availability and quality.
9. Base the initial design on expressed needs, and then pilot it (adjust it based on ongoing cost and benefit assessments).
10. Collect changes in the reference data now because the depth of the temporal tables is based on when collecting change begins.

About the Authors

Gregory J. Sannik is a principal consultant with Teradata Professional Services. He has worked in IT since 1978 and is a data warehousing expert in the financial services, insurance, and healthcare industries.

Fred Daniels is a senior consultant with Teradata Professional Services. He has worked in IT since 1991.

References

Developing Time-Oriented Database Applications in SQL, Snodgrass, Richard T., Morgan Kaufmann Publishers, 1997. http://www.cs.arizona.edu/people/rts/tdbbook.pdf

Temporal Table Support Release 13.10 B035-1182-109A September 2010