

Teradata Vantage™ - Temporal Table Support

Release 17.10




July 2021

Copyright and Trademarks

Copyright © 2010 - 2021 by Teradata. All Rights Reserved.

All copyrights and trademarks used in Teradata documentation are the property of their respective owners. For more information, see [Trademark Information](#).

Product Safety

Safety type	Description
	Indicates a situation which, if not avoided, could result in damage to property, such as to equipment or data, but not related to personal injury.
	Indicates a hazardous situation which, if not avoided, could result in minor or moderate personal injury.
	Indicates a hazardous situation which, if not avoided, could result in death or serious personal injury.

Third-Party Materials

Non-Teradata (i.e., third-party) sites, documents or communications ("Third-party Materials") may be accessed or accessible (e.g., linked or posted) in or in connection with a Teradata site, document or communication. Such Third-party Materials are provided for your convenience only and do not imply any endorsement of any third party by Teradata or any endorsement of Teradata by such third party. Teradata is not responsible for the accuracy of any content contained within such Third-party Materials, which are provided on an "AS IS" basis by Teradata. Such third party is solely and directly responsible for its sites, documents and communications and any harm they may cause you or others.

Warranty Disclaimer

Except as may be provided in a separate written agreement with Teradata or required by applicable law, the information available from the Teradata Documentation website or contained in Teradata information products is provided on an "as-is" basis, without warranty of any kind, either express or implied, including the implied warranties of merchantability, fitness for a particular purpose, or noninfringement.

The information available from the Teradata Documentation website or contained in Teradata information products may contain references or cross-references to features, functions, products, or services that are not announced or available in your country. Such references do not imply that Teradata Corporation intends to announce such features, functions, products, or services in your country. Please consult your local Teradata Corporation representative for those features, functions, products, or services available in your country.

The information available from the Teradata Documentation website or contained in Teradata information products may be changed or updated by Teradata at any time without notice. Teradata may also make changes in the products or services described in this information at any time without notice.

Machine-Assisted Translation

Certain materials on this website have been translated using machine-assisted translation software/tools. Machine-assisted translations of any materials into languages other than English are intended solely as a convenience to the non-English-reading users and are not legally binding. Anybody relying on such information does so at his or her own risk. No automated translation is perfect nor is it intended to replace human translators. Teradata does not make any promises, assurances, or guarantees as to the accuracy of the machine-assisted translations provided. Teradata accepts no responsibility and shall not be liable for any damage or issues that may result from using such translations. Users are reminded to use the English contents.

Feedback

To maintain the quality of our products and services, e-mail your comments on the accuracy, clarity, organization, and value of this document to: docs@teradata.com.

Any comments or materials (collectively referred to as "Feedback") sent to Teradata Corporation will be deemed nonconfidential. Without any payment or other obligation of any kind and without any restriction of any kind, Teradata and its affiliates are hereby free to (1) reproduce, distribute, provide access to, publish, transmit, publicly display, publicly perform, and create derivative works of, the Feedback, (2) use any ideas, concepts, know-how, and techniques contained in such Feedback for any purpose whatsoever, including developing, manufacturing, and marketing products and services incorporating the Feedback, and (3) authorize others to do any or all of the above.

Contents

Chapter 1: Introduction to Temporal Table Support	5
Changes and Additions	5
Chapter 2: Getting Started	6
The Need to Represent Time	6
Introduction to Temporal Table Support	7
Temporal Data Types	7
Temporal Statements	8
Temporal Tables	9
Chapter 3: Basic Temporal Concepts	12
Temporal Database Management System	12
Temporal Database	12
Transaction Time and Valid Time	12
UNTIL_CHANGED and UNTIL_CLOSED	16
Temporal Row Types	16
Temporal Table Modifications	18
Nontemporal Operations	19
Temporal Table Queries	20
Session Temporal Qualifiers	21
Timestamping	22
Period Data Type Usage	23
Chapter 4: Creating Temporal Tables	25
Creating Valid-Time Tables	25
Creating Transaction-Time Tables	28
Row Partitioning Temporal Tables	31
Creating Join Indexes for Temporal Tables	33
Loading Data into Temporal Tables	34
Chapter 5: SQL Data Definition Language (Temporal Forms)	36
ALTER TABLE (Temporal Form)	36
CREATE JOIN INDEX (Temporal Form)	51
CREATE RECURSIVE VIEW/REPLACE RECURSIVE VIEW (Temporal Forms)	56
CREATE TABLE/CREATE TABLE AS (Temporal Forms)	59
CREATE TRIGGER/REPLACE TRIGGER (Temporal Form)	75
CREATE VIEW/REPLACE VIEW (Temporal Forms)	80
SET SESSION (Session Temporal Qualifiers)	84
SET SESSION TTGRANULARITY TO	88

Usage Notes for Temporal Tables	88
Chapter 6: SQL Data Manipulation Language (Temporal Forms)	106
ABORT (Temporal Form)	106
DELETE (Temporal Form)	110
INSERT/INSERT SELECT (Temporal Forms)	116
MERGE (Temporal Form)	123
ROLLBACK (Temporal Form)	130
SELECT/SELECT INTO (Temporal Forms)	133
FROM Clause (Temporal Form)	158
UPDATE (Temporal Form)	164
UPDATE (Temporal Upsert Form)	172
Cursors and Temporal Queries	176
Chapter 7: SQL Data Control Language (Temporal Forms)	178
GRANT (Temporal Form)	178
REVOKE (Temporal Form)	181
Chapter 8: Administration	185
System Clocks	185
Nontemporal Operations	185
Capacity Planning for Temporal Tables	186
Archiving Temporal Tables	188
Related Information	188
Appendix A: How to Read Syntax	190
Appendix B: Examples	192
Appendix C: Potential Concurrency Issues with Current Temporal DML	216
Appendix D: Enforcing and Validating Temporal Referential Constraints	222
Appendix E: ANSI Temporal Tables	240
Appendix F: Additional Information	245

Introduction to Temporal Table Support

Teradata Vantage™ is our flagship analytic platform offering, which evolved from our industry-leading Teradata® Database. Until references in content are updated to reflect this change, the term Teradata Database is synonymous with Teradata Vantage.

Teradata Vantage™ - Temporal Table Support describes the concepts fundamental to understanding Teradata support for temporal (time-aware) tables and data. This documentation includes SQL language reference material and examples specific to the practical creation and manipulation of temporal tables.

Teradata also supports ANSI/ISO compatible temporal tables and syntax, which are different in several respects from the temporal tables and syntax described here. For more information on ANSI-compatible temporal tables, see [ANSI Temporal Tables](#) and *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186.

Changes and Additions

Date	Description
July 2021	Minor edits.

Getting Started

This section provides an introduction to Teradata temporal table support.

The Need to Represent Time

Some applications need to design and build databases where information changes over time. Doing so without temporal table support is possible, although complex.

Consider an application for an insurance company that uses a Policy table where the definition looks like this:

```
CREATE TABLE Policy(  
    Policy_ID INTEGER,  
    Customer_ID INTEGER,  
    Policy_Type CHAR(2),  
    Policy_Details CHAR(40)  
)  
UNIQUE PRIMARY INDEX(Policy_ID);
```

Suppose the application needs to record when rows in the Policy table became valid. Without temporal table support, one approach that the application can take is to add a DATE column to the Policy table called Start_Date. Suppose the application also needs to know when rows in the table are no longer valid. Another DATE column called End_Date can accomplish this.

The new definition of the table looks like this:

```
CREATE TABLE Policy(  
    Policy_ID INTEGER,  
    Customer_ID INTEGER,  
    Policy_Type CHAR(2),  
    Policy_Details CHAR(40)  
    Start_Date DATE,  
    End_Date DATE  
)  
UNIQUE PRIMARY INDEX(Policy_ID);
```

Several complications are now evident. For example, if a customer makes a change to their policy during the life of the policy, a new row would need to be created to store the new policy conditions that are in effect from that time until the end of the policy. But the policy conditions prior to the change are also likely to be important to retain for historical reasons. The original row represents the conditions that were in effect for the

beginning portion of the policy, but the `END_DATE` needs to be updated to reflect when the policy conditions were changed.

Additionally, because of these types of changes, it becomes likely that more than one row now has the same value for `Policy_ID`, so the primary index for the table also needs to change. All modifications to the table must now consider changing the `Start_Date` and `End_Date` columns. Queries will be more complicated.

The mere presence of a `DATE` column in a table does not make the table a temporal table, nor make the database a temporal database. A temporal database must record the time-varying nature of the information managed by the enterprise.

Rather than using approaches such as adding `DATE` columns to traditional tables, Vantage provides built-in support to more effectively create, query, and modify time-varying tables.

Introduction to Temporal Table Support

Teradata provides the built-in capabilities that are required in a temporal database management system. Temporal data types and temporal statements facilitate creating applications that need to represent time and the information that changes over time.

Feature	Description
Temporal data types	The period data type represents an anchored duration of time.
Temporal column attributes	In addition to user-defined time, which can be represented by using <code>DateTime</code> data types such as <code>DATE</code> and <code>TIMESTAMP</code> , Teradata temporal table support adds the capability to add valid-time and transaction-time dimensions to tables, by means of temporal column attributes.
Temporal statements	Temporal variations of existing statements let you create and alter temporal tables, and query and modify data that changes over time. Queries and modifications can include temporal qualifiers that reference a time dimension and act as criteria or selectors on the data. They affect only the data that meets the time criterion. Temporal DML statements can be generally qualified as: <ul style="list-style-type: none"> • <code>CURRENT</code>, affecting only data that is currently in effect • <code>SEQUENCED</code>, affecting only data that is in effect for a specified time period • <code>AS OF</code>, affecting only data that is in effect at a specified point in time • <code>NONSEQUENCED</code>, where the time dimension is ignored, the table is treated as a nontemporal table, and the DML statement is applied to all data in the table

Temporal Data Types

Teradata provides temporal table support at the data type level with period data types. A period is an anchored duration that represents a set of contiguous time granules within the duration. It has a beginning bound (defined by the value of a beginning element) and an ending bound (defined by the value of an ending element). Beginning and ending elements can be `DATE`, `TIME`, or `TIMESTAMP` types, but both must be the same type.

The duration that a period represents starts from the beginning bound and extends up to, but does not include, the ending bound.

If the element type is DATE or TIMESTAMP, the ending bound can have a special value of UNTIL_CHANGED, where Vantage interprets the ending bound of the period as forever, or without end.

As a first step toward adding temporal table support to the Policy table, the application for the insurance company can create the Policy table with a PERIOD(DATE) column to record when rows are valid.

```
CREATE TABLE Policy(
  Policy_ID INTEGER,
  Customer_ID INTEGER,
  Policy_Type CHAR(2) NOT NULL,
  Policy_Details CHAR(40),
  Validity PERIOD(DATE)
)
PRIMARY INDEX(Policy_ID);
```

Although the Policy table is a nontemporal table, the application can use the built-in support that Vantage provides for period types, including period constructors, literals, operators, functions, and predicates.

For example, to add a row to the table, the application could use the period constructor to specify a value for the Validity column.

```
INSERT INTO Policy
  (Policy_ID, Customer_ID, Policy_Type, Policy_Details, Validity)
VALUES (541008, 246824626, 'AU', 'STD-CH-345-NXY-00',
  PERIOD(DATE '2009-10-01', UNTIL_CHANGED));
```

To retrieve rows from the Policy table that became valid on a specific date, the application could use the BEGIN function like this:

```
SELECT * FROM Policy WHERE BEGIN(Validity) = DATE '2010-01-01';
```

Related Information

For more information on...	See...
period data types	<ul style="list-style-type: none"> Period Data Type Usage <i>Teradata Vantage™ - Data Types and Literals</i>, B035-1143 <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i>, B035-1145

Temporal Statements

Using the period data type and the temporal column attributes in a CREATE TABLE statement, the application for the insurance company can create the Policy table as a temporal table with a valid-time column to record when rows are valid.

```
CREATE MULTISET TABLE Policy(
  Policy_ID INTEGER,
  Customer_ID INTEGER,
  Policy_Type CHAR(2) NOT NULL,
  Policy_Details CHAR(40),
  Validity PERIOD(DATE) NOT NULL AS VALIDTIME
)
PRIMARY INDEX(Policy_ID);
```

The application can easily add a row to the table.

```
INSERT INTO Policy
  (Policy_ID, Customer_ID, Policy_Type, Policy_Details, Validity)
VALUES (541008, 246824626, 'AU', 'STD-CH-345-NXY-00',
  PERIOD(DATE '2009-10-01', UNTIL_CHANGED));
```

Similarly, the application can easily query the table. The result of the following query is the rows that are valid at the current time (where the value of the Validity column overlaps with the current time):

```
CURRENT VALIDTIME SELECT * FROM Policy;
```

Related Information

For more information on...	See...
temporal table concepts	Basic Temporal Concepts
creating temporal tables	<ul style="list-style-type: none"> • Creating Temporal Tables • SQL Data Definition Language (Temporal Forms)
manipulating temporal tables	<ul style="list-style-type: none"> • SQL Data Definition Language (Temporal Forms) • Modifying Temporal Tables
querying temporal tables	<ul style="list-style-type: none"> • SQL Data Definition Language (Temporal Forms) • Querying Temporal Tables

Temporal Tables

Temporal tables store and maintain information with respect to time. Using temporal tables, Vantage can process statements and queries that include time-based reasoning. Temporal tables include one or two special columns, which store time information:

- A transaction-time column records and maintains the time period for which Vantage was aware of the information in the row. Vantage automatically enters and maintains the transaction-time column data, and consequently automatically tracks the history of such information.
- A valid-time column models the real world, and stores information such as the time an insurance policy or product warranty is valid, the length of employment of an employee, or other information that is important to track and manipulate in a time-aware fashion. When you add a new row to this type of table, you use the valid-time column to specify the time period for which the row information is valid. This is the period of validity (PV) of the information in the row.

As rows are changed in temporal tables, the database automatically creates new rows as necessary to maintain the time dimensions. For example, if a row in a table with transaction time is modified, the row is automatically split into two rows:

- A new row is created to represent the changed information that exists following the modification. The beginning bound of its transaction time period value is set to the time of the modification, and its ending bound is left open (set to UNTIL_CLOSED). As far as the database is concerned, the information in the new row is true until it changes again or until the row is deleted.
- The original row represents the row as it existed before the modification. The ending bound of the transaction time period value of the row is set to the time of the modification. This row is “closed” and becomes a history row, because the information it contains from before the modification is no longer true. However, the row is not physically deleted from the database. It remains as a historical record of how the row existed before the modification was made.

Modifications to rows in table with a valid-time column are more flexible. When a row is modified in a table with a valid-time column, you can specify the time period for which the modification applies. This is the period of applicability (PA) of the modification. Depending on the relationship between the PV of the row and the PA of the modification, Vantage may split the modified row into multiple rows. For example, if the modification is applicable only to a brief period that lies within the PV of the row, three rows will result from a modification:

- One row has the original information, and a valid-time period that covers the time from the beginning of the original PV of the row until the modification happened.
- The second row has the modified information, and a valid-time period that matches the PA of the modification statement.
- The third row has the original information, like the first row, but has a valid-time period that covers the time starting from after the modification is no longer valid through the end time of the PV of the original row.

If the PA of the modification overlaps, but does not lie within the PV of the row, the modification will split the row into only two rows, similar to the example for a transaction-time table.

Transaction time and valid time are considered independent time dimensions, and their columns serve different purposes, so a table can have both a valid-time column and a transaction-time column. Such a dual-purpose temporal table is called a bitemporal table.

Related Information

For more information on...	See...
temporal table concepts	Basic Temporal Concepts

Basic Temporal Concepts

This section defines concepts related to Teradata temporal table support.

Temporal Database Management System

A temporal database management system (DBMS) is a DBMS that provides built-in support for the time dimension, including special facilities for storing, querying, and updating data with respect to time. A temporal DBMS can distinguish between historical data, current data, and data that will be in effect in the future.

The intent of a temporal database management system is to reason with time.

A temporal DBMS provides a temporal version of SQL, including enhancements to the data definition language (DDL), constraint specifications and their enforcements, data types, data manipulation language (DML), and query language for temporal tables.

Temporal Database

A temporal database stores data that relates to time periods and time instances. It provides temporal data types and stores information relating to the past, present, and future. For example, it stores the history of a stock or the movement of employees within an organization. The difference between a temporal database and a conventional database is that a temporal database maintains data with respect to time and allows time-based reasoning, whereas a conventional database captures only a current snapshot of reality.

For example, a conventional database cannot directly support historical queries about past status and cannot represent inherently retroactive or proactive changes. Without built-in temporal table support from the DBMS, applications are forced to use complex and often manual methods to manage and maintain temporal information.

Transaction Time and Valid Time

Static, time-related columns can be added to tables by adding columns defined to have DateTime data types, such as DATE or TIMESTAMP. Teradata also supports two built-in time dimensions that can be used to create temporal tables: transaction time and valid time.

Each of these time dimensions is represented by a column with a period data type. The column stores a pair of DATE or TIMESTAMP values that define the beginning and end of the transaction- or valid-time period for a row. Transaction time and valid time are independent time dimensions. A table can have either type of column, both, or neither:

- A table with a transaction-time column is called a transaction-time table.
- A table with a valid-time column is called a valid-time table.
- A table with both a transaction-time and a valid-time column is called a bitemporal table.

- A table with neither a transaction-time nor a valid-time column is a nontemporal table.

Transaction Time

Transaction time is the time period during which a fact, represented by all the information in a row, is or was known to be in effect in the database. It models the database reality, recording when rows have been added, modified, and changed in the database. Transaction-time periods are stored in a transaction-time column:

- The beginning of the transaction-time period is the time when the database became aware of a row, when the row was first recorded in the database. This is when the row was added to a table.
- The end of a transaction time period reflects when the fact was superseded by an update to the row, or when the row was deleted from the database. Rows containing information that is currently in effect have transaction-time periods with indefinite ending bounds, represented as `UNTIL_CLOSED`.

Transaction-time columns are defined by specifying `AS TRANSACTIONTIME` in the column definition, and have a period data type with an element type of `TIMESTAMP(6) WITH TIME ZONE`. You cannot normally set or modify the value of a transaction-time column. Vantage maintains these values automatically. (However, for database maintenance and troubleshooting, closed rows can be modified or deleted by administrators who have been granted the `NONTEMPORAL` privilege.)

Every change to a table that has a transaction-time column is tracked by the database. In a sense, physical rows are never deleted or modified in tables that have a transaction-time column:

- When a row is “deleted” from the table, the row is not physically deleted from the table. Instead, the transaction-time column is automatically modified to have an ending bound that specifies the time of the deletion, which marks the row as “closed,” and no longer available.
- When a row is “modified” in the table, the original row with the original values is marked as closed, and a copy of the row having the modified values is automatically inserted into the table.

The resulting snapshots of deleted and modified rows, which are retained in the table, provide a complete internal history of the table. Any prior state of a table having a transaction-time column can be reproduced. However, closed rows are unavailable to most DML modifications or deletions.

Add transaction-time columns to tables for which historical changes should be automatically tracked and maintained in the database. For example, transaction-time tables can be used for information that must retain a history of all changes, such as for tables used for regulatory compliance reporting.

Valid Time

Valid time models the real world, and denotes the time period during which a fact, represented by all the information in a row, is in effect or true. Valid-time periods are stored in a valid-time column.

Valid-time columns store information such as the time an insurance policy or contract is valid, the length of employment of an employee, or other information that is important to track and manipulate in a time-aware fashion. The valid-time period is also known as the period of validity (PV) of the row.

Valid-time columns are defined by specifying `AS VALIDTIME` in the column definition, and have a period data type with an element type of `DATE` or `TIMESTAMP(n)` (optionally including `WITH TIME ZONE`). You specify the value of the valid-time column when a new row is inserted into the table.

Vantage automatically maintains the valid-time column for rows that are changed or deleted, according to how the time period specified for the change or deletion relates to the original PV of the row.

For example, assume a row in a valid-time table represents the terms of a contract that is valid for two years. If the terms (row) must be modified during the contract period:

- A copy of the row is automatically created and modified to show the new terms. The PV of the row begins at the time of the change, to show when the new terms started. The PV of the row retains the original ending bound for the valid-time column, to retain the original contract end date.
- The original row, storing the original terms of the contract is marked as a history row. The PV is set to end at the time of the modification, because that is when the old terms ceased to be valid.

Such a modification changes the row information starting at the current time of the modification, and the change is valid throughout the remaining PV of the row.

Modifications to tables that have valid-time columns can also apply to specified time periods, even periods that do not overlap the current time, such as times that have passed or that are in the future. The changes will affect only those rows with PVs that overlap the specified time period, and only for the period during which the change is applicable. Other kinds of modifications to these tables can affect rows for their entire PVs, much like changes to nontemporal tables.

For example, if the terms of the contract in the example above were changed for only six weeks during the middle of the two-year contract period, the change would automatically yield three rows in the table:

- A copy of the row is automatically created and modified to show the new terms. The PV of the row reflects the six weeks for which the new terms are in effect.
- The original row, storing the original terms of the contract, is marked as a history row. The PV is set to end at the time the new terms begin.
- A new row is inserted to reflect the conditions after the six-week change in terms has ended, when the contract reverts to the original terms. The PV for the new row begins when the new terms expire, and ends at the original end time for the original row.

In this way, valid-time tables also keep an automatic history of all changes. Unlike transaction-time, however, history rows in tables with valid-time remain accessible to temporal SQL queries and DML. Because they model the real world, valid-time tables can have rows with a PV in the future, because things like contracts and policies may not begin or end until a future date.

Add valid-time columns to tables for which the information in a row is delimited by time, and for which row information should be maintained, tracked, and manipulated in a time-aware fashion. A valid time column is most appropriate when changes to rows occur relatively infrequently. To represent attributes that change very frequently, such as a point of sale table, an event table is preferable to a valid-time table. Temporal semantics do not apply to event tables.

Bitemporal Tables

Transaction time and valid time are independent time dimensions that are used for different purposes. Bitemporal tables have both a transaction-time column and a valid-time column. Changes to bitemporal tables that happen automatically as a result of row modifications are independent for the transaction-time and valid-time dimensions. These dimensions must be considered separately when determining what will happen to a row as a result of a modification.

For example, if a row in a bitemporal table is deleted, the ending bound of the transaction-time period is automatically changed to reflect the time of the deletion, and the row is closed to further modifications. The database reality, reflected by the modified ending bound of the transaction-time period, is that the row has been deleted.

The valid-time period of the row remains unchanged. Because the deletion does not affect the ending bound of the valid-time period, the row information retains its character in the valid-time dimension as historical, current, or future information. However, because the row was deleted, the row does not participate in further DML operations for the table, even though it remains in the table as a closed row in transaction time.

Because of the transaction-time column, all modifications to rows in bitemporal tables automatically create closed rows in the transaction time dimension, just as they do for transaction-time tables. This is in addition to rows that might be created to account for changes in the valid-time dimension.

For example, assume the terms of a contract are stored in a row of a bitemporal table. If the terms are changed during the period when the contract is valid, the row must be modified, as with an UPDATE statement. Because this is a temporal table, Vantage automatically inserts a copy of the row to store the new terms. The period of validity of the new row is automatically set to begin at the time of the change, and end at the original end date of the contract. The beginning bound of the transaction-time period of the new row reflects when the new row was created.

The original row is automatically modified to have the end of the period of validity reflect the time of the change, when the old terms become no longer valid. This row becomes a history row in the valid-time dimension. Note that both rows remain open rows in the transaction time dimension, and as such, are still available to all types of DML queries and modifications. These changes are purely a result of the valid-time dimension of the table.

Because the table also includes a transaction-time dimension, another copy is made of the original row, reflecting the original period of validity, but the row is closed in the transaction time dimension at the time the terms changed. No further changes can be made to this row, because it is closed in transaction time. It provides a permanent “before” snapshot of the row as it existed in the database before it was changed.

Note that the actions which are performed automatically by Vantage on the row include independent actions that result from the table having both a valid-time column and a transaction-time column.

Related Information

For more information on...	See...
temporal timestamping	Timestamping
UNTIL_CHANGED and UNTIL_CLOSED	UNTIL_CHANGED and UNTIL_CLOSED
creating temporal tables	Creating Temporal Tables
history, current, future, open, and closed rows	Temporal Row Types
NONTEMPORAL temporal qualifier	Nontemporal Operations
CREATE_TABLE (temporal form)	CREATE TABLE/CREATE TABLE AS (Temporal Forms)

UNTIL_CHANGED and UNTIL_CLOSED

UNTIL_CHANGED and UNTIL_CLOSED are special values that represent the ending bound of periods for which the duration is indefinite, forever, or for which the end is an unspecified and unknown time in the future when the row will be changed. They are used in some circumstances when a new row is inserted into a temporal table:

- All new rows that have transaction-time columns are assigned transaction-time periods with end bounds of UNTIL_CLOSED. UNTIL_CLOSED is only associated with transaction-time columns.
- New rows that have valid-time columns can be assigned valid-time periods with end bounds of UNTIL_CHANGED to represent that the information in the row is valid indefinitely.

UNTIL_CLOSED has a data type of `TIMESTAMP(6) WITH TIME ZONE` and a value of `TIMESTAMP '9999-12-31 23:59:59.999999+00:00'`.

The value of UNTIL_CHANGED depends on the data type and precision of the valid-time column. If the type is `PERIOD(DATE)`, UNTIL_CHANGED is the value `DATE '9999-12-31'`. If the type is `PERIOD(TIMESTAMP)`, UNTIL_CHANGED is the value of `TIMESTAMP '9999-12-31 23:59:59.999999+00:00'`, with precision and time zone matching that specified for the valid-time data type.

For more information on UNTIL_CHANGED and UNTIL_CLOSED, see the discussion of Period data types in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Temporal Row Types

A row in a valid-time table can be a current row, future row, or history row.

A row in a transaction-time table can be an open or closed row.

A row in a bitemporal table can be a current row, future row, or history row in the valid-time dimension and an open row or closed row in the transaction-time dimension.

Current Row

In a valid-time table, a current row is a row with a valid-time period that overlaps the current time. A row with a valid-time value of NULL is not considered a current row.

In a bitemporal table, a current row has a valid-time period that overlaps the current time, and a transaction-time period that is open.

Future Row

In a valid-time table, a future row is a row with a valid-time period that begins after the current time. A row with a valid-time value of NULL is not considered a future row.

Transaction-time tables never have future rows. The transaction-time period of a row in a transaction-time table begins at the time the row is created in the transaction-time table.

In a bitemporal table, a future row is a row with a valid-time period that begins after the current time, and a transaction-time period that is open.

History Row

In a valid-time table, a history row is a row that is no longer valid: a row with a valid-time period that ends before the current time. A row with a valid-time value of NULL is not considered a history row.

In a transaction-time table, a row that is closed is also considered to be a history row, regardless of the valid-time period. See [Closed Row](#).

In a bitemporal table, a history row is a row with a valid-time period that ends before the current time, or a row that is closed in transaction time, or a row that has both of these conditions. A row with a valid-time value of NULL is not considered a history row.

Open Row

An open row is a row that is currently known to the database, a row that is currently in effect in the database. It is a row that has not been (logically) deleted from the database or superseded by a row modification.

In a transaction-time table, an open row is a row with a transaction-time period that has an ending bound of UNTIL_CLOSED. When a new row is added to a table with transaction time, the ending bound of the transaction-time column period value is set to UNTIL_CLOSED, and the original row is considered to be open until the row is modified or deleted.

UNTIL_CLOSED has a value of TIMESTAMP '9999-12-31 23:59:59.999999+00:00'.

In a valid-time table, all rows are considered to be open. When a row is deleted from a valid-time table, the row is physically deleted from the database, rather than closed. See [Valid Row](#).

In a bitemporal table, an open row is a row with a transaction-time period that has an ending bound of UNTIL_CLOSED. Open rows that are no longer valid in the valid-time dimension are considered history rows, even though they remain open in the transaction-time dimension. See [History Row](#).

Closed Row

A closed row is a row that is no longer in effect in the database. It is a row that has either been (logically) deleted from the database, or otherwise superseded by a row modification, which closes the original row, and opens a new row with the changed information.

In a transaction-time or bitemporal table, a closed row is a row with a transaction-time period that has an ending bound different from UNTIL_CLOSED (9999-12-31 23:59:59.999999+00:00). Such a row is said to be closed in transaction time.

The concept of a closed row does not apply to valid-time tables. See [No Longer Valid Row](#).

Valid Row

A valid row is a current or future row. It is a row in a valid-time or bitemporal table with a valid-time period that overlaps current time, or that begins in the future.

For bitemporal tables, only rows that are open in the transaction-time dimension can be valid.

The concept of a valid row does not apply to a transaction-time table.

No Longer Valid Row

A row that is no longer valid is a row in a valid-time or bitemporal table with a valid-time period that ends before the current time. It is considered to be a history row in the valid-time dimension.

For bitemporal tables, only rows that are open in the transaction-time dimension can be considered no longer valid. Rows that are closed in the transaction-time dimension of a bitemporal table are considered history rows, regardless of their valid-time period, and are not considered either valid or no longer valid.

Temporal Table Modifications

Modifications to temporal tables can be current, sequenced, or nonsequenced. These operations apply to the valid-time dimension. The system automatically determines which rows are to be modified based on the interaction between the period of validity of each row and the period of applicability of the modification SQL.

With respect to the transaction-time dimension of transaction-time and bitemporal tables, a row is considered either open or closed. Open rows participate in database operations. Closed rows are historical snapshots of rows that have been deleted or modified subsequent to a prior state. After a row has been closed, it no longer participates in normal SQL operations, but can be viewed using temporal SQL.

Rows that have been closed in the transaction-time dimension remain as a permanent log of database operations on these rows. They are not available to be deleted or modified. However, these rows can be deleted and modified by users having the special NONTEMPORAL privilege, provided that capability is enabled in the database.

Period of Applicability

The period of applicability (PA) is the period specified implicitly or explicitly in a temporal query or DML statement. It is the period in valid time for which the query or modification applies. Vantage determines how to handle the SQL request based on the relationship between the PA of the statement and the PV in the valid-time column of the rows in the temporal table. PV is described in [Valid Time](#).

Current Temporal Modification

Current temporal modifications are modifications to the current rows of a temporal table.

The period of applicability of a current modification implicitly begins from the current time and extends indefinitely, represented by an ending period bound of UNTIL_CHANGED. Current modifications need not explicitly specify a period in the modification SQL. The CURRENT keyword causes the system to automatically apply the modification to those rows whose period of validity overlaps the current time.

Note:

Some proposed implementations of temporal tables suggest that a current modification also apply to future rows. In the Teradata implementation, a current modification applies only to current rows.

A current modification to any column creates history rows in the temporal table.

Sequenced Temporal Modification

Sequenced temporal modifications are modifications that apply for an explicitly specified time period, the PA.

Rows qualify for the modification if their valid-time period, the PV, overlaps the PA specified for the modification. Sequenced modifications can be made to rows with periods of validity in the past, present, or future.

Nonsequenced Temporal Modification

Nonsequenced temporal modifications are modifications to a temporal table that treat the temporal columns as any other column, and impose no special temporal semantics. Nonsequenced modifications may explicitly mention the valid-time column, but do not automatically create history rows.

Nontemporal Operations

A nontemporal operation is an operation where the NONTEMPORAL prefix is used with an ALTER TABLE, CREATE TABLE AS (Copy Table Syntax), DELETE, INSERT, or UPDATE statement. Nontemporal operations allow modifications to be made to closed rows in transaction-time and bitemporal tables which are normally not allowed on tables with transaction time. These operations can circumvent the automatic history that is normally kept for these kinds of temporal tables. They can specify a transaction

time when modifying and inserting rows, and can physically delete closed rows. These operations are normally not allowed on temporal tables that have transaction time. Use of the `NONTEMPORAL` keyword is discouraged in most situations, and requires the special `NONTEMPORAL` privilege. For more information see [Usage Notes](#).

Temporal Table Queries

Queries involving a temporal table with valid time can be current, sequenced, or nonsequenced. On a table with transaction time, temporal queries can be current or nonsequenced.

Current Temporal Query

A current query is a `SELECT` statement that extracts and operates on the current rows of a table:

- For a transaction-time table, current queries operate only on open rows.
- For a valid-time table, current queries operate only on rows with valid-time periods that overlap the current time.
- For bitemporal tables, current queries operate only on rows that are both open in the transaction-time dimension and current in the valid-time dimension.

A current query produces a nontemporal table as a result set.

Sequenced Temporal Query

A sequenced query is a `SELECT` statement that extracts and operates on rows in a valid-time or bitemporal table with valid-time periods that overlap a time period specified in the query (the PA of the query). If no time period is explicitly specified in the query, the default PA is all time, and the query applies to all open rows in the table. Such queries can return rows that are history rows, current rows, future rows, or combinations of the three.

A sequenced query produces a temporal table as a result set. The valid time of the result rows is the overlap of the query PA with the original row PV.

As Of Query

An as of query is a `SELECT` statement that extracts and operates on rows in temporal tables with valid-time and transaction-time periods that overlap an AS OF date or time specified in the query. As of queries operate on the data as a snapshot at any point in time. Typically, as of queries are used for querying historical data.

The AS OF clause can be applied to the valid-time and transaction-time dimensions together or independently. When applied to the valid-time dimension, it retrieves rows where the PV overlaps the specified AS OF time. When applied to the transaction-time dimension, it retrieves rows with transaction-time periods that overlap the specified AS OF time.

An as of query is similar in semantics to a current query; an AS OF extracts the information based on the specified time and a current query extracts the information as of the current time or date of the query being

executed. However, a current query operates on only open rows in the transaction-time dimension. An AS OF query can read rows as of a particular point in time in the transaction-time dimension regardless of whether the rows are closed or open.

An as of query produces a nontemporal table as a result set.

Nonsequenced Temporal Query

A nonsequenced query is a SELECT statement that treats temporal columns of temporal tables as if they were nontemporal columns. It does not place any special semantics on temporal columns. It considers all states simultaneously.

A NONSEQUENCED VALIDTIME query can optionally include a PA. In this case, the query produces a valid-time temporal table, where the valid time of the result set rows is the PA specified in the query. If a NONSEQUENCED VALIDTIME query does not include a PA, the query produces a nontemporal table as a result set.

A NONSEQUENCED TRANSACTIONTIME query cannot include a PA, and always produces a nontemporal table as a result set.

Related Information

For more information on...	See...
open, closed, current, future, and history rows	Temporal Row Types .
transaction time and valid time	Transaction Time and Valid Time
current, sequenced, as of, and nonsequenced queries	<ul style="list-style-type: none"> • SELECT/SELECT INTO (Temporal Forms) • Querying Temporal Tables

Session Temporal Qualifiers

The SET SESSION statement lets you set session temporal qualifiers in the valid-time dimension, transaction-time dimension, or both dimensions. When a DML or SELECT statement refers to a temporal table but omits a temporal qualifier, the system uses the value of the session temporal qualifier.

A session temporal qualifier of CURRENT VALIDTIME AND CURRENT TRANSACTIONTIME allows applications written prior to the temporal capability to continue functioning without modifications in a non-temporal way on tables that may have been converted to temporal tables.

You can check the session temporal qualifier setting by looking at the Temporal Qualifier field of the output of the HELP SESSION statement. For more information on session temporal qualifiers, see [SET SESSION \(Session Temporal Qualifiers\)](#). For more information on HELP SESSION, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Note:

Best practice is to always use explicit temporal qualifiers on SELECT and DML statements that operate on temporal tables, rather than relying on the default temporal session qualifiers.

Timestamping

An awareness of time is the defining feature of a temporal database. Rows with transaction-time columns are automatically tracked in time by the system, starting from the time the row is first inserted in a table. Rows with valid-time columns specify the period of time for which the information in the row is considered to be in effect. Whenever a row in any type of temporal table is modified or deleted, the system automatically timestamps the row and any new rows that are created as a result of the modification. These timestamps note the time of the change, and are used to close rows with a transaction-time column, and modify the PV as appropriate for rows with a valid-time column.

For example, if a row has a valid-time period that extends from last week to next week, and during that period a change is made to the row, the original information in the original row ceases to be effective at the time of the change. Consequently, the database timestamps the end bound of the valid-time column with the time of the change, because that marks the time the original information in the row is no longer valid. The database automatically adds a copy of the row to the table, having the changed values, and timestamps the valid-time period to begin at the time of the change. The new row retains the ending bound of the valid-time period from the original row.

Because transaction time and valid time are fundamentally different time dimensions, with different purposes in temporal tables, timestamps are calculated differently for transaction-time columns than they are for valid-time columns.

Transaction-Time Timestamping

When a row is added to, or modified in a temporal table with transaction time, the system automatically timestamps the transaction-time column to indicate when the system became aware of the new or modified information in the row.

By default, the timestamp used for transaction-time columns is the value read from the system clock by each AMP at the instant the row is inserted or modified. This value is referred to as `TT_TIMESTAMP` throughout this documentation:

- The beginning bound of the transaction-time period is automatically set to `TT_TIMESTAMP` for rows inserted into tables with transaction time.
- The ending bound of the transaction-time period is automatically set to `TT_TIMESTAMP` for rows that are modified in tables with transaction time. This maintains a history of when the change to the row occurred.

This automatic timestamping process produces different timestamps for each row within the same load job, and for each row within the same transaction. That means that all modifications, even those within a single transaction, are individually tracked by the database for tables that have a transaction-time column.

For example, a transaction consisting of two statements, where one statement inserts a row and the other statement deletes the previously inserted row leaves a track in the database in the form of a stored history row that is closed in transaction time, and unavailable to most SQL.

If a single modification to a row results in multiple rows being automatically added to the database, the system uses the same TT_TIMESTAMP value to timestamp all affected rows. For example, an update to a row of a table with a transaction-time column could result in an update to the original row, plus the insertion of one or two new rows. In this case, TT_TIMESTAMP would be the same time for all rows. For examples of how a modification to one row can result in one or two additional rows being added to the temporal table, see [Sequenced Updates](#).

Note:

The granularity of transaction-time timestamping can be changed by means of the SET SESSION TTGRANULARITY statement. For more information, see [SET SESSION TTGRANULARITY TO](#).

Valid-Time Timestamping

A valid-time column specifies the period of time for which the information in a row is effective. Because this information models the real world, such as the period for which a contract is valid:

- The valid-time period should always be explicitly specified when adding rows to tables that have a valid-time column.
- An explicit time period should always be specified when making DML modifications to tables that have a valid-time column. This specifies the period of applicability (PA) of the change, which may not exactly match the PV of any row. If the PA of a modification does not match the PV of a row, Vantage determines how to make the change by the relationship between the PA and PV. If the PA and PV overlaps, the modification involves adding new rows to the table to account for the period before and after the change.

Although the PA and PV can be explicitly specified for operations on tables with valid-time columns, Vantage will use default values if these periods are not specified. For valid-time tables, the current time at the time of a row insertion or modification is timestamped as the value of the built-in function TEMPORAL_TIMESTAMP, or TEMPORAL_DATE if the valid-time column has a DATE type.

The value of TEMPORAL_TIMESTAMP or TEMPORAL_DATE for a transaction is the time or date when the first non-locking reference is made to a temporal table, or when the built-in function is first accessed during the transaction.

For more information on TEMPORAL_TIMESTAMP built-in function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Period Data Type Usage

Both temporal and nontemporal tables support Period data types and literals.

With respect to temporal tables:

- transaction-time columns must have a data type of `PERIOD(TIMESTAMP(6) WITH TIME ZONE)` or be a derived period column where the component begin and end time columns are both `PERIOD(TIMESTAMP(6) WITH TIME ZONE)`.
- Valid-time columns can have a Period or derived period data type of `PERIOD(DATE)`, `PERIOD(TIMESTAMP[(n)])`, or `PERIOD(TIMESTAMP[(n)] WITH TIME ZONE)`.

For more information on Period and DateTime data types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Creating Temporal Tables

This section describes tasks related to creating temporal tables.

Temporal tables are distinguished from nontemporal tables by having temporal columns: a valid-time column, a transaction-time column, or both. A temporal column can be derived from two individual DateTime columns representing the beginning and end of a duration, or it can be a true Period data type column that represents the duration as a single Teradata data type. Teradata recommends using derived temporal columns, because Vantage cannot collect statistics on temporal data in Period data type columns.

Creating Valid-Time Tables

There are several ways to create a valid-time table:

- Create a new table as a valid-time table
- Add a valid-time column to a nontemporal or transaction-time table
- Create a valid-time table from a copy of a nontemporal table

Creating a New Valid-Time Table

To create a valid-time table, use a normal CREATE TABLE statement, and define one column as a valid-time column. The valid-time column can either be derived from two DATE, TIMESTAMP, or TIMESTAMP[(n)] WITH TIME ZONE columns, or can be a single Period data type column: PERIOD(DATE), PERIOD(TIMESTAMP[(n)]), or PERIOD(TIMESTAMP[(n)] WITH TIME ZONE). Use the VALIDTIME or AS VALIDTIME column attribute to assign the column to be the valid-time column.

Example: Creating a Derived Period Valid-Time Column

```
CREATE MULTISET TABLE Policy(
  Policy_ID INTEGER,
  Customer_ID INTEGER,
  Policy_Type CHAR(2) NOT NULL,
  Policy_Details CHAR(40),
  Policy_Start DATE NOT NULL,
  Policy_End DATE NOT NULL,
  PERIOD FOR Validity(Policy_Start,Policy_End) AS VALIDTIME
)
PRIMARY INDEX(Policy_ID);
```

Example: Creating a Period Data Type Valid-Time Column

```
CREATE MULTISET TABLE Policy(
  Policy_ID INTEGER,
  Customer_ID INTEGER,
  Policy_Type CHAR(2) NOT NULL,
  Policy_Details CHAR(40),
  Validity PERIOD(DATE) NOT NULL AS VALIDTIME
)
PRIMARY INDEX(Policy_ID);
```

Adding a Valid-Time Column to a Table

To add a valid-time column to a nontemporal or transaction-time table, use the ALTER TABLE statement.

Temporal tables cannot have unique primary indexes. If the original nontemporal table has a unique primary index, use ALTER TABLE to modify the primary index to nonunique prior to adding the temporal column. Uniqueness can be applied to a temporal table using other constraints. For more information, see [Using Constraints with Temporal Tables](#).

Example: Adding a Period Data Type Valid-Time Column

Consider the following nontemporal table definition that lacks duration data:

```
CREATE MULTISET TABLE Customer (
  Customer_Name VARCHAR(40),
  Customer_ID INTEGER,
  Customer_Address VARCHAR(80),
  Customer_Phone VARCHAR(12)
)
PRIMARY INDEX (Customer_ID);
```

The following statement adds a valid-time column to the Customer table:

```
ALTER TABLE Customer
ADD Customer_Validity PERIOD(DATE) AS VALIDTIME;
```

Converting a Period Column to a Valid-Time Column

If you have a table that defines a Period column, take the following steps to convert the existing Period column to a valid-time column:

1. Note all the constraint information on the table.
2. Drop all the constraints.
3. ALTER TABLE to add a new valid-time column.

4. Submit NONSEQUENCED VALIDTIME update to set the new valid-time column with the existing period column value.
5. ALTER TABLE to drop the existing Period column from the table.
6. ALTER TABLE to rename the valid-time column with the name of the dropped column.
7. Create all of the previously dropped constraints with the desired valid-time qualifier.

Creating a Valid-Time Table as a Copy of a Nontemporal Table

To create a valid-time table as a copy of an existing nontemporal table, use CREATE TABLE AS (the copy table form of CREATE TABLE). Use the AS clause to specify a temporal query that returns a table with valid time.

Example: Creating a Valid-Time Table as a Copy of a Nontemporal Table

Consider the following nontemporal table:

```
CREATE TABLE Policy_NT (
  Policy_ID INTEGER,
  Customer_ID INTEGER,
  Policy_Type CHAR(2) NOT NULL,
  Policy_Details CHAR(40)
)
PRIMARY INDEX(Policy_ID);
```

To create a copy of the Policy_NT table as a valid-time table, use a nonsequenced query in the AS clause of CREATE TABLE to specify a valid time period of applicability qualifier on the SELECT. The result is a valid-time table where the period of validity for every row is set to the period of applicability that was used in the query.

```
CREATE MULTISET TABLE Policy(
  Policy_ID,
  Customer_ID,
  Policy_Type,
  Policy_Details,
  Validity
) AS (
  NONSEQUENCED VALIDTIME PERIOD '(2009-01-01, UNTIL_CHANGED)'
  SELECT *
  FROM Policy_NT)
WITH DATA
PRIMARY INDEX(Policy_ID);
```

The resulting Policy table has a valid-time column named Validity:

```

SHOW TABLE Policy;

CREATE MULTISET TABLE Policy ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT,
  DEFAULT MERGEBLOCKRATIO
  (
    Policy_ID INTEGER,
    Customer_ID INTEGER,
    Policy_Type CHAR(2) CHARACTER SET LATIN NOT CASESPECIFIC,
    Policy_Details CHAR(40) CHARACTER SET LATIN NOT CASESPECIFIC,
    Validity PERIOD(DATE) AS VALIDTIME)
  PRIMARY INDEX ( Policy_ID );

```

Related Information

For more information on...	See...
valid-time periods	Valid Time
CREATE TABLE (temporal form)	CREATE TABLE/CREATE TABLE AS (Temporal Forms)
derived period columns	CREATE TABLE/CREATE TABLE AS (Temporal Forms) , and <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144
ALTER TABLE (temporal form)	ALTER TABLE (Temporal Form)
UPDATE TABLE (temporal form)	UPDATE (Temporal Form)

Creating Transaction-Time Tables

There are three ways to create a transaction-time table:

- Create a new table as a transaction-time table
- Add a transaction-time column to a nontemporal or valid-time table
- Add a derived period column for transaction time to a table that uses two columns to represent the beginning and end of a duration that represents the transaction time for the row information.

Creating a New Transaction-Time Table

To create a transaction-time table, use a normal CREATE TABLE statement, and define one column of the table as a transaction-time column. The data type of the column can be derived from two TIMESTAMP(6) WITH TIME ZONE columns, or can be a single PERIOD(TIMESTAMP(6) WITH TIME ZONE) column,

and must use the TRANSACTIONTIME or AS TRANSACTIONTIME column attribute. Transaction-time columns must also specify the NOT NULL column attribute.

Example: Creating a Derived Period Transaction-Time Table

```
CREATE MULTISET TABLE Policy_Types (
  Policy_Name VARCHAR(20),
  Policy_Type CHAR(2) NOT NULL PRIMARY KEY,
  Policy_Start TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  Policy_End TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  PERIOD FOR Policy_Duration(Policy_Start,Policy_End)
  AS TRANSACTIONTIME
)
PRIMARY INDEX (Policy_Name);
```

Example: Creating a Period Data Type Transaction-Time Table

```
CREATE MULTISET TABLE Policy_Types (
  Policy_Name VARCHAR(20),
  Policy_Type CHAR(2) NOT NULL PRIMARY KEY,
  Policy_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
  AS TRANSACTIONTIME
)
PRIMARY INDEX (Policy_Name);
```

Adding a Transaction-Time Column to a Table

To add a transaction-time column to a nontemporal or valid-time table, use the ALTER TABLE statement.

Temporal tables cannot have unique primary indexes. If the original nontemporal table has a unique primary index, use ALTER TABLE to modify the primary index to nonunique prior to adding the temporal column. Uniqueness can be applied to a temporal table using other constraints. For more information, see [Using Constraints with Temporal Tables](#).

Example: Creating a Transaction-Time Column Based on Two Existing Timestamp Columns

If a nontemporal table has two pre-existing TIMESTAMP(6) WITH TIME ZONE columns that represent the beginning and end bounds of a period of time, you can add a derived period column to the table based on these columns. The derived period column can serve as a transaction-time column, converting the table to a temporal table.

Note:

The begin and end columns of a derived period column cannot be included in a primary index if the derived period column serves as a valid-time or transaction-time column.

Given a table that was originally created with the following DDL statement:

```
CREATE MULTISET TABLE Policy(
  Policy_ID INTEGER,
  Customer_ID INTEGER,
  Policy_Type CHAR(2) NOT NULL,
  Policy_Details CHAR(40),
  Policy_Tx_Begin TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  Policy_Tx_End TIMESTAMP(6) WITH TIME ZONE NOT NULL
)
PRIMARY INDEX(Policy_ID);
```

Prior to converting the table to a temporal table, existing constraints should be noted, then dropped. The table could be converted to a transaction-time table using the following statement:

```
ALTER TABLE Policy
  ADD PERIOD FOR Policy_Tx_Duration (Policy_Tx_Begin,Policy_Tx_End)
  AS TRANSACTIONTIME;
```

After creating the temporal table, the constraints that were dropped can be reapplied.

Example: Adding a Period Data Type Transaction-Time Column

Consider the following nontemporal table definition that lacks duration data:

```
CREATE MULTISET TABLE Customer (
  Customer_Name VARCHAR(40),
  Customer_ID INTEGER,
  Customer_Address VARCHAR(80),
  Customer_Phone VARCHAR(12)
)
PRIMARY INDEX (Customer_ID);
```

The following statement adds a transaction-time column to the Customer table:

```
ALTER TABLE Customer
  ADD Customer_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
  AS TRANSACTIONTIME;
```

Converting a Period Column to a Transaction-Time Column

If you have a table that includes a column of type PERIOD(TIMESTAMP(6) WITH ZONE), take the following steps to convert the existing Period column to a transaction-time column:

1. Note all the constraint information on the original table.

2. Drop all the constraints on the original table.
3. Grant NONTEMPORAL privilege to the user on the table.
4. ALTER TABLE to add a transaction-time column.
5. Submit NONTEMPORAL UPDATE to set the new transaction-time column with the existing column value being converted.
6. ALTER TABLE to drop the existing Period column.
7. ALTER TABLE to rename the transaction-time column with the name of the dropped column.
8. Create all the previously dropped constraints with the desired transaction-time qualifier.

Related Information

For more information on...	See...
transaction-time periods	Transaction Time
CREATE TABLE (temporal form)	CREATE TABLE/CREATE TABLE AS (Temporal Forms)
ALTER TABLE (temporal form)	ALTER TABLE (Temporal Form)
UPDATE TABLE (temporal form)	UPDATE (Temporal Form)

Row Partitioning Temporal Tables

To improve the performance of current queries on a temporal table, the table should be row partitioned. The table is logically divided into a set of current rows, and history rows (open rows and closed rows in the transaction-time dimension). Current queries are directed automatically to the partition containing current, open rows.

Note:

Column partitioning can also be applied to temporal tables, however the row partitioning described here should always constitute one of the partitioning types used for a temporal table.

Row Partitioning a Valid-Time Table

To row partition a valid-time table, use the following PARTITION BY clause.

Example: Row Partitioning a Valid-Time Table

```
CREATE MULTISET TABLE Policy(
  Policy_ID INTEGER,
  Customer_ID INTEGER,
  Policy_Type CHAR(2) NOT NULL,
  Policy_Details CHAR(40),
  Validity PERIOD(DATE) NOT NULL AS VALIDTIME
```

```

    )
PRIMARY INDEX(Policy_ID)
PARTITION BY
CASE_N(
    END(Validity)>= CURRENT_DATE AT INTERVAL - 12:59' HOUR TO MINUTE,
    NO CASE);

```

Row Partitioning a Transaction-Time Table

To row partition a transaction-time table, use the following PARTITION BY clause.

Example: Row partitioning a transaction-time table

```

CREATE MULTISET TABLE Policy_Types (
    Policy_Name VARCHAR(20),
    Policy_Type CHAR(2) NOT NULL PRIMARY KEY,
    Policy_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
        AS TRANSACTIONTIME
)
PRIMARY INDEX (Policy_Name)
PARTITION BY
CASE_N (END(Policy_Duration) >= CURRENT_TIMESTAMP, NO CASE);

```

Row Partitioning a Bitemporal Table

To row partition a bitemporal table, use the following PARTITION BY clause.

Example: Row partitioning a bitemporal table

```

CREATE MULTISET TABLE Policy_Bitemp (
    Policy_ID INTEGER,
    Customer_ID INTEGER,
    Policy_Type CHAR(2) NOT NULL,
    Policy_Details CHAR(40),
    Validity PERIOD(DATE) NOT NULL AS VALIDTIME,
    Policy_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
        AS TRANSACTIONTIME
)
PRIMARY INDEX (Policy_ID)
PARTITION BY CASE_N(
    (END(Validity) IS NULL OR
    END(Validity) >= CURRENT_DATE AT
        INTERVAL - '12:59' HOUR TO MINUTE) AND
    END(Policy_Duration) >= CURRENT_TIMESTAMP,

```

```

END(Validity) < CURRENT_DATE AT
    INTERVAL -'12:59' HOUR TO MINUTE AND
END(Policy_Duration) >= CURRENT_TIMESTAMP,
END(Policy_Duration) < CURRENT_TIMESTAMP);

```

Maintaining a Current Partition

As time passes, and current rows become history rows, you should periodically use the ALTER TABLE TO CURRENT statement to transition history rows out of the current partition into the history partition. ALTER TABLE TO CURRENT resolves the partitioning expressions again, transitioning rows to their appropriate partitions per the updated partitioning expressions. For example:

```
ALTER TABLE temporal_table_name TO CURRENT;
```

This statement also updates any system-defined join indexes that were automatically created for primary key and unique constraints defined on the table.

Related Information

For more information on...	See...
row partitioning temporal tables	Partitioning Expressions for Temporal Tables
CREATE TABLE (temporal form)	CREATE TABLE/CREATE TABLE AS (Temporal Forms)
ALTER TABLE TO CURRENT (regular form)	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144

Creating Join Indexes for Temporal Tables

Join indexes aid performance by providing a smaller data set and shorter data access path for common queries that would otherwise require full table scans. Join indexes can also help the Optimizer better optimize queries. Join indexes can be created for temporal tables.

Creating a Join Index on a Table with Valid Time

To create a join index on a table with valid time, precede the SELECT statement in the join index definition with a CURRENT VALIDTIME, NONSEQUENCED VALIDTIME, or SEQUENCED VALIDTIME qualifier. For example:

```

CREATE JOIN INDEX Policy_JI AS
CURRENT VALIDTIME SELECT Policy_ID, Policy_Type, Validity
FROM Policy;

```

A join index that is current or sequenced in the valid-time dimension must project the valid-time column in the SELECT statement to ensure that the join index is used appropriately. Here, the Validity column is the valid-time column.

Creating a Join Index on a Table with Transaction Time

To create a join index on a table with transaction time, precede the SELECT statement in the join index definition with a CURRENT TRANSACTIONTIME or NONSEQUENCED TRANSACTIONTIME qualifier. For example:

```
CREATE JOIN INDEX Policy_Types_JI AS
CURRENT TRANSACTIONTIME SELECT Policy_Type, Policy_Duration
FROM Policy_Types;
```

A join index that is current in the transaction-time dimension must project the transaction-time column in the SELECT statement. Here, the Policy_Duration column is the transaction-time column.

Maintaining Current Join Indexes

As time passes, and current rows become history rows, you should periodically use the ALTER TABLE TO CURRENT statement to ensure that the rows in the index continue to reflect only rows that are valid. For example:

```
ALTER TABLE Policy_JI TO CURRENT;
```

Related Information

For more information on...	See...
CREATE JOIN INDEX (temporal form)	CREATE JOIN INDEX (Temporal Form)
ALTER TABLE TO CURRENT (regular form)	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144

Loading Data into Temporal Tables

Teradata supports the following methods of bulk loading data into temporal tables:

- FastLoad (and applications that support the FastLoad protocol), can perform bulk loads directly into empty temporal tables, if the tables are not column partitioned.

Do not use CURRENT INSERT (valid time or transaction time) if the FastLoad script includes a CHECKPOINT specification. Restarts during loading can change the valid time and transaction time values for rows that are inserted after the restart.

In this case, use `NONTEMPORAL INSERT` instead, to ensure an exact one-to-one correspondence between the number of rows inserted and the number of rows that results in the target table.

- MultiLoad can be used to load data into nontemporal staging tables, followed by the use of `INSERT SELECT` statements to load the data from the staging tables into temporal tables.
- Teradata Parallel Transporter (TPT) includes the Update Operator, which can load temporal data from valid-time NoPI staging tables to valid-time or bitemporal tables.

SQL Data Definition Language (Temporal Forms)

This section describes the SQL DDL statements related to temporal tables.

This material covers the syntax, rules, and other details that are specific to temporal table support.

All existing rules that apply to the corresponding non-temporal DDL statements also apply to the statements here. For more information about these rules, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

ALTER TABLE (Temporal Form)

Performs any of the following:

- Adds a valid-time column or a transaction-time column or both to an existing table.
- Adds, modifies, or drops columns from temporal tables.
- Drops a valid-time column or a transaction-time column or both.
- Adds, drops, or modifies constraints.

ALTER TABLE Syntax (Temporal Form)

Note:

Teradata Vantage™ - Temporal Table Support describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
[NONTEMPORAL] ALTER TABLE
  [ database_name. | user_name. ] table_name
  [, alter_table_option ] [, ...]
  [ other_alter_table_option ] [, ...] [;]
```

For descriptions of standard data types, column attributes, and modifications relating to indexes, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

other_alter_table_option

```
{ column_change |
  table_level_constraint_change |
```

```

ADD NORMALIZE [ ALL BUT ( ignored_column_list ) ]
    ON normalize_column [ ON OVERLAPS [ OR MEETS ] ] |

DROP NORMALIZE

}

```

column_change

```

{ ADD column_name {
    data_type [ column_attributes ] |
    column_attributes |
    column_constraint_attribute |
    temporal_column
} |

DROP column_name [IDENTITY] [ WITHOUT DELETE ] |

RENAME old_column_name [AS] TO new_column_name |

derived_period_column

}

```

table_level_constraint_change

```

{ table_level_reference_definition |

ADD [ CONSTRAINT name ] [ time_option ] CHECK ( boolean_condition ) |

DROP [ CONSTRAINT name ] CHECK |

MODIFY [ CONSTRAINT name ] [ time_option ] CHECK ( boolean_condition
) |

table_level_unique_definition

}

```

column_constraint_attribute

```

[ CONSTRAINT name ] {

```

```

    [ time_option ] { CHECK ( boolean_condition ) | UNIQUE | PRIMARY
KEY } |

    [ RI_time_option ] REFERENCES WITH NO CHECK OPTION
    table_name [ ( column_name ) ]
}

```

temporal_column

```

{ { PERIOD (DATE) | PERIOD ( [ ( precision ) ] [ WITH TIME ZONE ] ) }
  [ NOT NULL ] [AS] VALIDTIME |

  PERIOD (TIMESTAMP(6) WITH TIME ZONE) NOT NULL [AS] TRANSACTIONTIME
}

```

derived_period_column

```

{ ADD PERIOD FOR derived_column ( begin_column, end_column )
  [ [AS] { VALIDTIME | TRANSACTIONTIME } ] |

  DROP [ PERIOD FOR ] derived_column
}

```

table_level_reference_definition

```

{ { ADD [ CONSTRAINT name ] | DROP }
  [ RI_time_option ]
  FOREIGN KEY ( column_name [,...] )
  REFERENCES WITH NO CHECK OPTION table_name [ ( column_name
[,...] ) ] |

  DROP INCONSISTENT REFERENCES
}

```

time_option

```

CURRENT TRANSACTIONTIME
[ AND [ CURRENT | SEQUENCED | NONSEQUENCED ] VALIDTIME ]

```

table_level_unique_definition

```
{ DROP CONSTRAINT name |

  ADD [ CONSTRAINT name ] time_option
    { UNIQUE | PRIMARY KEY } ( column_name [,...] )
}
```

RI_time_option

```
{ CURRENT | SEQUENCED | NONSEQUENCED } TRANSACTIONTIME
[ AND [ CURRENT | SEQUENCED | NONSEQUENCED ] VALIDTIME ]
]
```

ALTER TABLE Syntax Elements (Temporal Form)**NONTEMPORAL**

Specifies that the table to be altered has transaction time and that the ALTER TABLE operation will modify data in the table, rather than save historical snapshots of existing rows before making changes.

database_name.table_name

Specifies the name of the table to alter and the optional name of the database or user in which it is contained if different from the current database.

user_name.table_name***table_name******alter_table_option***

Specifies an option from the Alter Table Options list for a conventional ALTER TABLE statement.

Column Changes Clause***ADD column_name***

Used to add or change the specified column and its specified attributes.

ADD and DROP cannot both be specified on the same column in the same ALTER TABLE statement.

The ADD keyword either changes the definition of an existing column or adds a new column to the table. If the named column already exists, ADD indicates that its attributes are to be changed.

Adding or changing a column on a table with transaction time requires that the statement include the **NONTEMPORAL** prefix.

Data Type Column Attributes

Specifies one or more data definition phrases that define data for the column.

You must always specify a data type for a newly added column. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for more information on data types and column attributes.

For information on Period data types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

DROP *column_name* [IDENTITY] [WITHOUT DELETE]

Specifies that the named column is to be removed from the table.

Dropping a column on a table with transaction time requires that the statement include the **NONTEMPORAL** prefix.

When the transaction-time column is dropped from a table, all closed rows are deleted from the table.

When the valid-time column is dropped from a table, all the rows that are no longer valid are deleted from the table.

The **IDENTITY** option removes the **IDENTITY** attribute from a column without dropping the column itself. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

The **WITHOUT DELETE** option allows the **TRANSACTIONTIME** column to be removed from the table without deleting the rows that have been closed in transaction time.

Column Constraint Attributes

For more information on constraints, see [Using Constraints with Temporal Tables](#).

CONSTRAINT *name*

Specifies the optional name for a constraint.

CURRENT TRANSACTIONTIME

Specifies that only rows that are open in transaction time are to be checked for constraint violations.

SEQUENCED TRANSACTIONTIME

Specifies that any value for the constrained column in the child table must exist in the parent table during the time period for which it exists in the child.

Note:

The sequenced transactiontime constraint qualifier is valid only for the REFERENCES constraint

NONSEQUENCED TRANSACTIONTIME

Specifies that any value for the constrained column in the child table must exist in the parent table. A nonsequenced constraint is like a nontemporal relational constraint, and ignores the transaction-time column in the child table.

Note:

The parent table cannot have a transaction-time column. The nonsequenced transactiontime constraint qualifier is valid only for the REFERENCES constraint.

CURRENT VALIDTIME

Specifies that only rows that are current and future in valid time are to be checked for constraint violations. History rows are not checked.

[SEQUENCED] VALIDTIME

Specifies that rows that are history, current, and future in valid time are to be checked for constraint violations.

NONSEQUENCED VALIDTIME

Specifies that rows that are history, current, and future in valid time are to be checked for constraint violations. Nonsequenced constraints treat the valid-time column as a non-temporal column, so are similar to constraints on non-temporal tables. They apply to all open rows of the table.

CHECK (*boolean_condition*)

Specifies a boolean conditional expression that must be true, or else the row violates the check constraint.

Check constraints cannot be placed on valid-time or transaction-time columns.

UNIQUE**PRIMARY KEY**

Specifies that during the qualified time, any given values in the constrained columns will not exist in more than one row at any instant in time:

- For a current constraint this means any current or future rows that have overlapping time periods cannot have the same value in the columns.

- For a sequenced constraint this means any history, current, or future rows that have overlapping time periods cannot have the same value in the columns.
- For a nonsequenced constraint this means that the value of the column is unique in every row in the table, irrespective of whether row time periods overlap. This is similar to a unique or primary key constraint in a non-temporal table.

In all cases, if the table has a transaction-time column, the constraint is applied only to rows that are open in transaction time.

For a current or sequenced PRIMARY KEY or UNIQUE constraint defined on a valid-time table, the valid-time column must be defined as NOT NULL.

The PK or UNIQUE columns cannot include the valid-time or transaction-time columns.

Because PK and unique constraints on temporal tables are implemented as system-defined join indexes or unique secondary indexes, the constraint is not allowed if it would cause the maximum number of secondary indexes to be exceeded for the table.

REFERENCES *table_name* [(*column_name*)]

Specifies a primary key-foreign key (or unique) referential integrity constraint where *table_name* is the parent table.

The column cannot be a valid-time or transaction-time column.

WITH NO CHECK OPTION

Specifies that referential integrity is not to be enforced for the specified primary key-foreign key relationship.

Temporal Column Definitions

[AS] VALIDTIME

Specifies that the column to be added is a valid-time column.

A valid-time column can be added only if its data type is PERIOD(DATE), PERIOD(TIMESTAMP[(n)]), or PERIOD(TIMESTAMP[(n)] WITH TIME ZONE).

There can be no more than one valid-time column in a temporal table.

The existing table cannot have a UPI and cannot have any unique, primary key, or referential integrity constraints if a valid-time column is added.

[AS] TRANSACTIONTIME

Specifies that the column to be added is a transaction-time column.

A transaction-time column must have a data type of PERIOD(TIMESTAMP(6) WITH TIME ZONE).

There can be no more than one transaction-time column in a temporal table.

A DEFAULT clause cannot be specified for a transaction-time column.

A transaction-time column must be defined as NOT NULL.

The existing table cannot have a UPI and cannot have any unique, primary key, or referential integrity constraints if a transaction-time column is added.

Derived Period Column Changes

derived_column

Specifies the name of the derived period column.

(begin_column,end_column)

Specifies the names of the DateTime columns that will serve as the beginning and ending bounds of the derived period column. The data types of the columns must be identical, and both must be defined as NOT NULL.

The begin and end bound columns used for a derived period column that will function as a valid-time column must be of data type DATE or TIMESTAMP(*n*) WITH TIME ZONE, where *n* is the precision of the timestamp, and WITH TIME ZONE is optional.

The begin and end bound columns used for a derived period column that will function as a transaction-time column must be of data type TIMESTAMP(6) WITH TIME ZONE.

[AS] VALIDTIME [AS] TRANSACTIONTIME

Specifies that the derived period column will serve as the valid-time or transaction-time column of a temporal table.

Note:

The begin and end columns of a derived period column cannot be included in a primary index if the derived period column serves as a valid-time or transaction-time column.

Constraint Change Options

For more information on constraints, see [Using Constraints with Temporal Tables](#).

ADD DROP ADD/DROP CONSTRAINT *name*

Used to add or drop a constraint.

When a named primary key or unique constraint is dropped, the associated system-defined join index or USI is automatically dropped.

You cannot use ALTER TABLE to drop an unnamed primary key or unique constraint. Instead, use DROP JOIN INDEX to drop a system-defined join index, or DROP INDEX to drop a system-defined unique secondary index.

Table Level REFERENCES Definition

For more information on constraints, see [Using Constraints with Temporal Tables](#).

CONSTRAINT *name*

Specifies the optional name for the constraint.

CURRENT TRANSACTIONTIME

Specifies that every value for the constrained column in the child table must exist in the open rows of the parent table.

SEQUENCED TRANSACTIONTIME

Specifies that any value for the constrained column in the child table must exist in the parent table during the time period for which it exists in the child.

NONSEQUENCED TRANSACTIONTIME

Specifies that any value for the constrained column in the child table must exist in the parent table. A nonsequenced constraint is like a nontemporal relational constraint, and ignores the transaction-time column in the child table.

Note:

The parent table cannot have a transaction-time column.

CURRENT VALIDTIME

Specifies that any value for the constrained column in the child table must exist in the open current or future rows of the parent table. The valid-time period of the child row must be contained within the combined valid-time periods of current and future rows in the parent table that have a value that matches the child table. History rows in the child table are not checked, and history rows in the parent table are not considered.

[SEQUENCED] VALIDTIME

Specifies that any value for the constrained column in the child table must exist in the open rows of the parent table. The valid-time period of the child row must be contained within the combined valid-time periods of open rows in the parent table that have a value that matches the child table.

NONSEQUENCED VALIDTIME

Specifies that any value for the constrained column in the child table must exist in the parent table. A nonsequenced constraint is like a nontemporal relational constraint, and ignores the valid-time column in the child table.

Note:

The parent table cannot have a valid-time column.

FOREIGN KEY

Specifies a foreign key for the table.

column_name

Specifies a name for a column defined as part of the foreign key.

REFERENCES WITH NO CHECK OPTION

Specifies an integrity reference to the parent table named in *table_name*.

Referential integrity is not enforced for the specified primary key-foreign key (or unique) relationship

table_name

Specifies the name of the referenced parent table used in the referential integrity constraint definition.

column_name

Specifies a column in the column set that makes up the parent table PRIMARY KEY or UNIQUE candidate key columns. The column cannot be a valid-time or transaction-time column.

DROP INCONSISTENT REFERENCES

Specifies to delete all inconsistent references defined on the table.

Table Level CHECK Definition

For more information on constraints, see [Using Constraints with Temporal Tables](#).

CONSTRAINT *name*

Specifies the optional name for the constraint.

Note:

Transactiontime and validtime options, if used together, can be specified in any order, separated by AND.

CURRENT TRANSACTIONTIME

Specifies that only rows that are open in transaction time are to be checked for constraint violations.

CURRENT VALIDTIME

Specifies that only rows that are current and future in valid time are to be checked for constraint violations. History rows are not checked.

[SEQUENCED] VALIDTIME

Specifies that rows that are history, current, and future in valid time are to be checked for constraint violations.

NONSEQUENCED VALIDTIME

Specifies that rows that are history, current, and future in valid time are to be checked for constraint violations. Nonsequenced constraints treat the valid-time column as a non-temporal column, so are similar to constraints on nontemporal tables. They apply to all open rows of the table.

CHECK (*boolean_condition*)

Specifies a boolean conditional expression that must be true, or else the row violates the check constraint.

Check constraints cannot be placed on valid-time or transaction-time columns.

Table Level UNIQUE Definition

For more information on constraints, see [Using Constraints with Temporal Tables](#).

CONSTRAINT *name*

Specifies the optional name for the constraint.

Note:

Transactiontime and validtime options, if used together, can be specified in any order, separated by AND.

CURRENT TRANSACTIONTIME

Specifies that only rows that are open in transaction time are to be checked for constraint violations.

CURRENT VALIDTIME

Specifies that only rows that are current and future in valid time are to be checked for constraint violations. History rows are not checked.

[SEQUENCED] VALIDTIME

Specifies that rows that are history, current, and future in valid time are to be checked for constraint violations.

NONSEQUENCED VALIDTIME

Specifies that rows that are history, current, and future in valid time are to be checked for constraint violations. Nonsequenced constraints treat the valid-time column as a non-temporal column, so are similar to constraints on nontemporal tables. They apply to all open rows of the table.

**UNIQUE
PRIMARY KEY**

Specifies that during the qualified time, any given values in the constrained columns will not exist in more than one row at any instant in time:

- For a current constraint this means any current or future rows that have overlapping time periods cannot have the same value in the columns.
- For a sequenced constraint this means any history, current, or future rows that have overlapping time periods cannot have the same value in the columns.
- For a nonsequenced constraint this means that the value of the column is unique in every row in the table, irrespective of whether row time periods overlap. This is similar to a unique or primary key constraint in a non-temporal table.

In all cases, if the table has a transaction-time column, the constraint is applied only to rows that are open in transaction time.

For a current or sequenced PRIMARY KEY or UNIQUE constraint defined on a valid-time table, the valid-time column must be defined as NOT NULL.

The PK or UNIQUE columns cannot include the valid-time or transaction-time columns.

Because PK and unique constraints on temporal tables are implemented as system-defined join indexes or unique secondary indexes, the constraint is not allowed if it would cause the maximum number of secondary indexes to be exceeded for the table.

column_name

Specifies a column in the column set to be used as the primary key or as unique. The column cannot be a valid-time or transaction-time column.

Table Level NORMALIZE Definition

For more information on the NORMALIZE option, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

NORMALIZE

Specifies the rows are to be combined based on a specified period column. The values for all other columns must be identical for rows to be combined. The period column values for combined rows must overlap or meet, and are coalesced to result in a single time period that is the union of the time periods for the combined rows.

The period column for normalization can be a derived period column.

ALL BUT (*ignored_column_list*)

Specifies a column name or comma-separated list of column names that are to be ignored for purposes of the normalization. When rows are inserted or updated in the table, the value of the ignored columns in the result is non-deterministic. These values will reflect values from rows that have been recently added to the table, but precise values cannot be predicted nor necessarily reproduced.

Note:

Although temporal tables with transaction time can be normalized, the values in the transaction-time column are likely to be unique. Therefore most table rows will not qualify for normalization unless the transaction-time column is included in the *ignored_column_list*.

ON *normalize_column*

Specifies the period or derived period column to be normalized. This cannot be a transaction-time column, but can be a valid-time or other period column.

Note:

When normalization is performed on tables having transaction time, only open rows qualify for normalization.

ON OVERLAPS

Specifies the normalize condition is that the period values for rows that will be coalesced must overlap. ON OVERLAPS OR MEETS is the default for NORMALIZE.

OR MEETS

Specifies the normalize conditions is that the period values for rows that will be coalesced must meet. ON OVERLAPS OR MEETS is the default for NORMALIZE.

Usage Notes

Adding a Valid-Time Column

If a default value is specified for a new valid-time column, the column is populated with the specified value. Otherwise, the new column is populated as follows:

IF the valid-time column has a ...	THEN the column is populated with a value of ...
PERIOD(DATE) data type	PERIOD(TEMPORAL_DATE, UNTIL_CHANGED).
PERIOD(TIMESTAMP) data type	PERIOD(TEMPORAL_TIMESTAMP, UNTIL_CHANGED). The precision and time zone values are set depending on the data type of the new column.

In addition to the rules for specifying a valid-time column specified in [CREATE TABLE/CREATE TABLE AS \(Temporal Forms\)](#), the following rules apply when using ALTER TABLE to add a valid-time column to an existing table:

- If the table has a transaction-time column, the ALTER TABLE statement must specify the NONTEMPORAL prefix. This requires the NONTEMPORAL privilege on the table.
- The table cannot have a UPI.
- If it does, first use ALTER TABLE *table_name* MODIFY NOT UNIQUE, which is described in *SQL Data Definition Language*, to modify the PI to be non-unique.
- The table may have a NUPI, or it can have no primary index.
- Existing CHECK constraints become current constraints in the valid-time dimension. An error is reported if there are any other types of constraints. See [Using Constraints with Temporal Tables](#).
- Any join indexes defined on the table must be dropped before the table can be made a valid-time table.
- The table cannot be the subject table of an existing trigger.
- Existing views, macros, or triggered action statements that reference the table but do not specify a valid-time qualifier in the statement referencing the table must be modified to add a valid-time qualifier.

If an executing stored procedure includes an SQL statement that references the table being altered, and no explicit qualifier is specified in the SQL, the compile time qualifier is applied to the SQL.

The partitioning for the table cannot be altered to be partitioned on the added valid-time column unless the table is empty.

Adding a Transaction-Time Column

The added transaction-time column is populated with the system default value of `PERIOD(TEMPORAL_TIMESTAMP, UNTIL_CLOSED)`.

In addition to the rules for specifying a transaction-time column specified in [CREATE TABLE/CREATE TABLE AS \(Temporal Forms\)](#), the following rules apply when using `ALTER TABLE` to add a transaction-time column to an existing table:

- The table cannot have a UPI.
- If it does, first use `ALTER TABLE table_name MODIFY NOT UNIQUE`, which is described in *SQL Data Definition Language*, to modify the PI to be non-unique.
- The table may have a NUPI, or it can have no primary index.
- Existing CHECK constraints become current constraints in the transaction-time dimension. An error is reported if there are any other types of constraints. See [Using Constraints with Temporal Tables](#).
- Any join indexes defined on the table must be dropped before the table can be made a transaction-time table.
- The table cannot be the subject table of an existing trigger.
- Existing views, macros, or triggered action statements that reference the table but do not specify a transaction-time qualifier in the statement referencing the table must be modified to add a transaction-time qualifier.

If an executing stored procedure includes an SQL statement that references the table being altered, and no explicit qualifier is specified in the SQL, the compile time qualifier is applied to the SQL.

The row partitioning for the table cannot be altered to be partitioned on the added transaction-time column unless the table is empty.

Dropping Temporal Columns

Dropping any type of column from a transaction-time or bitemporal table requires the `NONTEMPORAL` privilege on the table, and the `NONTEMPORAL` qualifier to `ALTER TABLE` must be used.

Temporal columns cannot be dropped if any constraints or a join index or a trigger are defined on the table.

Temporal columns cannot be dropped if the table is partitioned on the temporal columns.

When a transaction-time column is dropped, all closed rows (all history rows in the transaction-time dimension) are deleted from the table.

When a valid-time column is dropped, all the rows that are no longer valid (all history rows in the valid-time dimension) are deleted from the table.

Related Information

For more information on...	See...
ALTER TABLE (regular form)	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
adding temporal columns to nontemporal tables	<ul style="list-style-type: none"> • Adding a Valid-Time Column to a Table. • Adding a Transaction-Time Column to a Table.
CREATE TABLE (temporal form)	CREATE TABLE/CREATE TABLE AS (Temporal Forms).
derived period columns	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
dropping temporal columns from tables	<ul style="list-style-type: none"> • Usage Notes. • Example: Dropping a Valid-Time Column. • Example: Dropping a Transaction-Time Column.
temporal table constraints	Using Constraints with Temporal Tables.

CREATE JOIN INDEX (Temporal Form)

Creates a join index on temporal tables.

For more information on sequenced aggregate join indexes, see [Aggregate Functions in Sequenced Queries](#).

CREATE JOIN INDEX Syntax (Temporal Form)

Note:

Teradata Vantage™ - Temporal Table Support describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
CREATE JOIN INDEX [ database_name. | user_name. ] join_index_name
  standard options AS [ temporal_qualifier ] select_statement
```

temporal_qualifier

```
{ valid_time_qualifier [ AND transaction_time_qualifier ] |
  transaction_time_qualifier [ AND valid_time_qualifier ]
}
```

valid_time_qualifier

```
[ CURRENT | SEQUENCED | NONSEQUENCED ] VALIDTIME
```

transaction_time_qualifier

```
{ CURRENT | NONSEQUENCED } TRANSACTIONTIME
```

CREATE JOIN INDEX Syntax Elements (Temporal Form)

database_name

Specifies an optional database name or user name specified if the join index is to be contained in a database or user other than the current database or user.

user_name**join_index_name**

Specifies the name given to the join index created by this statement.

standard options

Specifies standard table and join index options that are the same for temporal and nontemporal tables. For information about these options, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*.

CURRENT VALIDTIME

Specifies that the join index is current in valid time.

The *select_statement* must project the valid-time column in a join index definition that is current in valid time. If the column is a derived period column, the component columns of the derived period column must be projected.

SEQUENCED VALIDTIME

Specifies that the join index is sequenced in valid time.

The *select_statement* must project the valid-time column in a join index definition that is sequenced in valid time. If the column is a derived period column, the component columns of the derived period column must be projected.

NONSEQUENCED VALIDTIME

Specifies that the join index is nonsequenced in valid time.

AND

Specifies a keyword for specifying both a valid-time qualifier and a transaction-time qualifier. The valid-time and transaction-time qualifiers, if used together, can be specified in any order.

CURRENT TRANSACTIONTIME

Specifies that the join index is current in transaction time.

The *select_statement* must project the transaction-time column in a join index definition that is current in transaction time. If the column is a derived period column, the component columns of the derived period column must be projected.

NONSEQUENCED TRANSACTIONTIME

Specifies that the join index is nonsequenced in transaction time.

select_statement

Specifies conventional SELECT statement syntax for creating a join index.

Usage Notes

Join Indexes on Tables with Transaction Time

The following table shows the types of join index that can be created on transaction-time tables, and whether the transaction-time column (or component columns of a derived period transaction-time column) must be projected in the index.

Qualifier	Single Table JI	Multitable JI	Transaction-time Column Required in JI and SEQUENCED VT AJI
CURRENT TRANSACTIONTIME	Allowed	Allowed	Yes / No
TRANSACTIONTIME AS OF	Disallowed	Disallowed	Not applicable
SEQUENCED TRANSACTIONTIME	Disallowed	Disallowed	Not applicable

Qualifier	Single Table JI	Multitable JI	Transaction-time Column Required in JI and SEQUENCED VT AJI
NONSEQUENCED TRANSACTIONTIME	Allowed	Allowed	No / No

If no explicit transaction-time qualifier is specified in the statement, the system uses the session transaction-time qualifier.

When a current join index is created, the following condition is added to the join definition: `END(< TransactionTimeColumn >) IS UNTIL_CLOSED`.

If a current join index on a transaction-time table has an outer join and results in a derived table, the system returns an error.

Vantage maintains any current join indexes in the transaction-time dimension with every DML statement on the base table.

Although projecting the transaction-time column is not required for nonsequenced join indexes, doing so can increase the usefulness of the index.

Join Indexes on Tables with Valid Time

The following table shows the types of join index that can be created on valid-time tables, and whether the valid-time column (or component columns of a derived period valid-time column) must be projected in the index.

Qualifier	Single Table JI	Multitable JI	Valid-time Column Required in JI / Aggregate JI
CURRENT VALIDTIME	Allowed	Disallowed	Yes / No
VALIDTIME AS OF	Disallowed	Disallowed	Not applicable
SEQUENCED VALIDTIME	Allowed	Allowed	Yes / No
NONSEQUENCED VALIDTIME	Allowed	Allowed	No / No

If no explicit valid-time qualifier is specified in the statement, the system uses the session valid-time qualifier.

Vantage maintains any current and sequenced join indexes in the valid-time dimension with every current or sequenced DML statement on the base table, regardless of whether the column being modified is included in the join index.

For sequenced join indexes, the system does not append a VALIDTIME column that is normally added to the results of a SEQUENCED SELECT statement.

Although projecting the valid-time column is not required for nonsequenced join indexes, doing so can increase the usefulness of the index.

To avoid the high current join index and sequenced join index maintenance cost for a table with valid time, modify the columns that are not referenced in the current join index using nonsequenced DML (nontemporal DML if table is bitemporal). Such columns must be time-invariant columns whose history is not required.

If the join index involves only time-invariant columns, the best practice is to create a nonsequenced join index. This avoids the reference of a valid-time column and, thus, avoids join index maintenance steps when columns that are not part of the join index are modified.

Current multitable JIs are not supported for valid-time tables, however a sequenced valid-time index can be created to include only current valid-time queries issued at any point in time from current time to future time, as shown in the following example of a multitable join index created from two bitemporal tables. Note that the temporal columns must be projected in the sequenced index.

TEMPORAL_DATE and TEMPORAL_TIMESTAMP in Join Indexes

When TEMPORAL_DATE and TEMPORAL_TIMESTAMP are used in the WHERE clause of a CREATE JOIN INDEX statement, they resolve to static values. They do not confer any temporal behavior on the join index.

Maintaining Current Join Indexes

As time passes, and current rows become history rows, you should periodically use the ALTER TABLE TO CURRENT statement to ensure that the rows in the index continue to reflect only rows that are valid. For example:

```
ALTER TABLE Policy_JI TO CURRENT;
```

Example: Join Index on Table with Valid Time

```
CREATE JOIN INDEX AS
SEQUENCED VALIDTIME and CURRENT TRANSACTIONTIME
SELECT X1, Y1, VT1, TT1, X2, Y2, VT2, TT2
FROM   t1, t2
WHERE  END(t1.VT1) >= TEMPORAL_DATE
AND    END(t2.VT2) >= TEMPORAL_DATE;
```

Related Information

For more information on...	See...
ALTER TABLE TO CURRENT	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144

For more information on...	See...
CREATE JOIN INDEX	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144
join indexes for temporal tables	Creating Join Indexes for Temporal Tables

CREATE RECURSIVE VIEW/REPLACE RECURSIVE VIEW (Temporal Forms)

Creates or replaces a recursive view definition involving temporal tables.

CREATE RECURSIVE VIEW/REPLACE RECURSIVE VIEW Syntax (Temporal Forms)

Note:

Teradata Vantage™ - Temporal Table Support describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
{ CREATE | REPLACE } RECURSIVE VIEW
  [ database_name. | user_name. ] view_name ( column_name [,...] )
  AS { as_spec | ( as_spec ) } [;]
```

as_spec

```
[ temporal_qualifier ] seed_statement
    UNION ALL [ union_all_spec [ UNION ALL ... ] ]
[ temporal_qualifier ]
recursive_statement
```

temporal_qualifier

```
{ valid_time_qualifier [ AND transaction_time_qualifier ] |
  transaction_time_qualifier [ AND valid_time_qualifier ] |
  AS OF date_timestamp_expression
}
```

union_all_spec

```
[ temporal_qualifier ] { seed_statement | recursive_statement }
```

valid_time_qualifier

```
[ CURRENT | SEQUENCED | NONSEQUENCED ] VALIDTIME
```

transaction_time_qualifier

```
{ CURRENT TRANSACTIONTIME |  
  TRANSACTIONTIME AS OF date_timestamp_expression |  
  NONSEQUENCED TRANSACTIONTIME  
}
```

CREATE RECURSIVE VIEW/REPLACE RECURSIVE VIEW Syntax Elements (Temporal Forms)

view_name

Specifies the name of the recursive view.

database_name

Specifies the name of the database or user to contain *view_name* if something other than the current database or user.

user_name***column_name***

Specifies the name of a view column. If more than one column is specified, list their names in the order in which each column is to be displayed for the view.

CURRENT VALIDTIME

Specifies that *seed_statement* is a current query in the valid-time dimension. The result set is a nontemporal table.

VALIDTIME AS OF *date_timestamp_expression*

Specifies that *seed_statement* retrieves rows where the period of validity overlaps the specified AS OF expression. The result set is a nontemporal table.

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

NONSEQUENCED VALIDTIME

Specifies that *seed_statement* is a nonsequenced query in the valid-time dimension. If *period_expression* is specified, the nonsequenced query produces a table with valid time; otherwise, the result set is a nontemporal table.

The result set for a valid-time table includes an extra column for the overlapped valid-time period. If the list of columns for the view does not provide a name for the extra column, the default name is VALIDTIME.

period_expression

Specifies the period of applicability for the nonsequenced query.

CURRENT TRANSACTIONTIME

Specifies that *seed_statement* or *recursive_statement* is a current query in the transaction-time dimension. The result set is a nontemporal table.

TRANSACTIONTIME AS OF *date_timestamp_expression*

Specifies that *seed_statement* or *recursive_statement* retrieves rows whose transaction-time period in the row overlaps the specified AS OF expression. The result set is a nontemporal table.

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

NONSEQUENCED TRANSACTIONTIME

Specifies that *seed_statement* or *recursive_statement* is a nonsequenced query in the transaction-time dimension. A nonsequenced query produces a nontemporal table as a result set.

AS OF *date_timestamp_expression*

Specifies that *seed_statement* or *recursive_statement* retrieves rows whose valid-time and transaction-time periods overlap the specified AS OF expression.

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

seed_statement

Specifies existing seed SELECT statement syntax for CREATE RECURSIVE VIEW or REPLACE RECURSIVE VIEW.

recursive_statement

Specifies existing recursive SELECT statement syntax for CREATE RECURSIVE VIEW or REPLACE RECURSIVE VIEW.

Usage Notes

Seed Statement and Recursive Statement

All rules that apply to the temporal form of the SELECT statement are applicable for queries specified in recursive views.

If no temporal qualifier is specified for the view and the view references any temporal tables, the temporal qualifier defaults to the applicable session temporal qualifier.

The seed statement and the recursive statement must have compatible valid-time qualifiers.

Selecting from a Recursive View

A query that selects from a recursive view can specify a temporal qualifier that is different from the temporal qualifier of the SELECT statements in the view definition.

Related Information

For more information on...	See...
CREATE RECURSIVE VIEW (regular form)	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144
CREATE VIEW (temporal form)	CREATE RECURSIVE VIEW/REPLACE RECURSIVE VIEW (Temporal Forms)
temporal table views	Views on Temporal Tables

CREATE TABLE/CREATE TABLE AS (Temporal Forms)

Required Privileges

The privileges required are the same as those required for a conventional CREATE TABLE statement.

For the Copy Table syntax, if the CREATE TABLE statement specifies the NONTEMPORAL qualifier, the NONTEMPORAL privilege is also required on the target temporal table.

CREATE TABLE/CREATE TABLE AS Syntax (Temporal Form)

Note:

Teradata Vantage™ - Temporal Table Support describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
[NONTEMPORAL] CREATE MULTiset [ GLOBAL TEMPORARY | VOLATILE ] TABLE
  [ database_name. | user_name. ] table_name
  [ , create_table_option [...] ]
  ( other_create_table_option [,...] )
  standard_table_options [;]
```

Note:

For descriptions of standard data types, column attributes, index and partition options, and other standard CREATE TABLE syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

other_create_table_option

```
{ column_definition |
  derived_period_column |
  table_level_definition |
  NORMALIZE [ ALL BUT ( ignored_column_list ) ]
    ON normalize_column [ ON OVERLAPS [ OR MEETS ] ]
}
```

column_definition

```
column_name {
    data_type [ column_attribute | column_constraint_attributes ] [...] |
    temporal_column
}
```

derived_period_column

```
PERIOD FOR derived_column ( begin_column, end_column )
    [ [AS] { VALIDTIME | TRANSACTIONTIME } ]
```

table_level_definition

```
{ unique_definition | reference_definition | check_definition }
```

column_constraint_attributes

```
[ CONSTRAINT name ] {
    [ time_option ] { CHECK ( boolean_condition ) | UNIQUE | PRIMARY
    KEY } |
    [ RI_time_option ] REFERENCES WITH NO CHECK OPTION
    table_name [ ( column_name ) ]
}
```

temporal_column

```
{ { PERIOD (DATE) | PERIOD ( [ ( precision ) ] [ WITH TIME ZONE ] ) }
  [ NOT NULL ] [AS] VALIDTIME |
  PERIOD ( TIMESTAMP(6) WITH TIME ZONE ) NOT NULL [AS] TRANSACTIONTIME
}
```

unique_definition

```
[ CONSTRAINT name ]
  [ time_option ]
  { UNIQUE | PRIMARY KEY }
  ( column_name [,...] )
```

reference_definition

```
[ CONSTRAINT name ]
  [ RI_time_option ]
  FOREIGN KEY
  ( column_name [,...] )
  REFERENCES WITH NO CHECK OPTION table_name [ (
column_name [,...] ) ]
```

check_definition

```
[ CONSTRAINT name ]
  [ time_option ]
  CHECK ( boolean_condition )
```

time_option

```
CURRENT TRANSACTIONTIME
  [ AND [ CURRENT | SEQUENCED | NONSEQUENCED ] VALIDTIME ]
```

RI_time_option

```
{ CURRENT | SEQUENCED | NONSEQUENCED } TRANSACTIONTIME
  [ AND [ CURRENT | SEQUENCED | NONSEQUENCED ] VALIDTIME ]
]
```

CREATE TABLE/CREATE TABLE AS Syntax Elements (Temporal Form)

NONTEMPORAL

Specifies (for Copy Table Syntax, CREATE TABLE AS only) that all the transaction-time values are to be copied from the source table to a target table that has transaction time.

NONTEMPORAL

If the target table is not a table with transaction time, the NONTEMPORAL prefix is ignored.

MULTISET

Specifies that duplicate rows are allowed.

Note:

Temporal tables must be multiset tables.

GLOBAL TEMPORARY

Specifies that a temporary table definition be created and stored in the Data Dictionary for future materialization.

VOLATILE

Specifies that a volatile table be created, with its definition retained in memory only for the course of the session in which it is defined.

database_name.table_name

user_name.table_name

table_name

Specifies the name of the new table and the optional name of the database or user in which it is to be contained if different from the current database.

create_table_option

Specifies an option from the Create Table Options list for a conventional CREATE TABLE statement.

Column Definition Clause

column_name

Specifies the name of one or more columns, in the order in which they and their attributes are to be defined for the table.

data_type column_attributes

Specifies one or more data definition phrases that define the type and attributes of data that will be stored in the column.

See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for more information on data types and column attributes.

For information on Period data types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Column Constraint Attributes

For more information on constraints, see [Using Constraints with Temporal Tables](#).

CONSTRAINT *name*

Specifies the optional name for a constraint.

Note:

Transactiontime and validtime options, if used together, can be specified in any order, separated by AND.

CURRENT TRANSACTIONTIME

Specifies that only rows that are open in transaction time are to be checked for constraint violations.

SEQUENCED TRANSACTIONTIME

Specifies that any value for the constrained column in the child table must exist in the parent table during the time period for which it exists in the child.

Note:

The sequenced transactiontime constraint qualifier is valid only for the REFERENCES constraint.

NONSEQUENCED TRANSACTIONTIME

Specifies that any value for the constrained column in the child table must exist in the parent table. A nonsequenced constraint is like a nontemporal relational constraint, and ignores the transaction-time column in the child table.

Note:

The parent table cannot have a transaction-time column. The nonsequenced transactiontime constraint qualifier is valid only for the REFERENCES constraint.

CURRENT VALIDTIME

Specifies that only rows that are current and future in valid time are to be checked for constraint violations. History rows are not checked.

[SEQUENCED] VALIDTIME

Specifies that rows that are history, current, and future in valid time are to be checked for constraint violations.

NONSEQUENCED VALIDTIME

Specifies that rows that are history, current, and future in valid time are to be checked for constraint violations. Nonsequenced constraints treat the valid-time column as a nontemporal column, so are similar to constraints on nontemporal tables. They apply to all open rows of the table.

CHECK (*boolean_condition*)

Specifies a boolean conditional expression that must be true, or else the row violates the check constraint.

Check constraints cannot be placed on valid-time or transaction-time columns.

UNIQUE**PRIMARY KEY**

Specifies that during the qualified time, any given values in the constrained columns will not exist in more than one row at any instant in time:

- For a current constraint this means any current or future rows that have overlapping time periods cannot have the same value in the columns.
- For a sequenced constraint this means any history, current, or future rows that have overlapping time periods cannot have the same value in the columns.
- For a nonsequenced constraint this means that the value of the column is unique in every row in the table, irrespective of whether row time periods overlap. This is similar to a unique or primary key constraint in a non-temporal table.

In all cases, if the table has a transaction-time column, the constraint is applied only to rows that are open in transaction time.

For a current or sequenced PRIMARY KEY or UNIQUE constraint defined on a valid-time table, the valid-time column must be defined as NOT NULL.

The PK or UNIQUE columns cannot include the valid-time or transaction-time columns.

Because PK and unique constraints on temporal tables are implemented as system-defined join indexes or unique secondary indexes, the constraint is not allowed if it would cause the maximum number of secondary indexes to be exceeded for the table.

REFERENCES *table_name* [(*column_name*)]

Specifies a primary key-foreign key (or unique) referential integrity constraint where *table_name* is the parent table.

The column cannot be a valid-time or transaction-time column.

The implications of temporal qualifiers used with referential constraints are more complex than those for other constraints. For more information, see [Temporal Referential Constraints](#).

WITH NO CHECK OPTION

Specifies that referential integrity is not to be enforced for the specified primary key-foreign key (or unique) relationship.

Temporal Column Definitions

[AS] VALIDTIME

Specifies that the column to be added is a valid-time column.

A valid-time column can be added only if its data type is PERIOD(DATE), PERIOD(TIMESTAMP[(n)]), or PERIOD(TIMESTAMP[(n)] WITH TIME ZONE).

There can be no more than one valid-time column in a temporal table.

The existing table cannot have a UPI and cannot have any unique, primary key, or referential integrity constraints if a valid-time column is added.

[AS] TRANSACTIONTIME

Specifies that the column to be added is a transaction-time column.

A transaction-time column must have a data type of PERIOD(TIMESTAMP(6) WITH TIME ZONE).

There can be no more than one transaction-time column in a temporal table.

A DEFAULT clause cannot be specified for a transaction-time column.

A transaction-time column must be defined as NOT NULL.

The existing table cannot have a UPI and cannot have any unique, primary key, or referential integrity constraints if a transaction-time column is added.

Derived Period Column Definitions

derived_column

Specifies the name of the derived period column.

(begin_column,end_column)

Specifies the names of the DateTime columns that will serve as the beginning and ending bounds of the derived period column. The data types of the columns must be identical, and both must be defined as NOT NULL.

The begin and end bound columns used for a derived period column that will function as a valid-time column must be of data type DATE or TIMESTAMP(*n*) WITH TIME ZONE, where *n* is the precision of the timestamp, and WITH TIME ZONE is optional.

The begin and end bound columns used for a derived period column that will function as a transaction-time column must be of data type TIMESTAMP(6) WITH TIME ZONE.

[AS] VALIDTIME**[AS] TRANSACTIONTIME**

Specifies that the derived period column will serve as the valid-time or transaction-time column of a temporal table.

Note:

The begin and end columns of a derived period column cannot be included in a primary index if the derived period column serves as a valid-time or transaction-time column.

Table Level UNIQUE Definition

For more information on constraints, see [Using Constraints with Temporal Tables](#).

CONSTRAINT *name*

Specifies the optional name for the constraint.

Note:

Transactiontime and validtime options, if used together, can be specified in any order, separated by AND.

CURRENT TRANSACTIONTIME

Specifies that only rows that are open in transaction time are to be checked for constraint violations.

CURRENT VALIDTIME

Specifies that only rows that are current and future in valid time are to be checked for constraint violations. History rows are not checked.

[SEQUENCED] VALIDTIME

Specifies that rows that are history, current, and future in valid time are to be checked for constraint violations.

NONSEQUENCED VALIDTIME

Specifies that rows that are history, current, and future in valid time are to be checked for constraint violations. Nonsequenced constraints treat the valid-time column as a nontemporal column, so are similar to constraints on nontemporal tables. They apply to all open rows of the table.

UNIQUE**PRIMARY KEY**

Specifies that during the qualified time, any given values in the constrained columns will not exist in more than one row at any instant in time:

- For a current constraint this means any current or future rows that have overlapping time periods cannot have the same value in the columns.
- For a sequenced constraint this means any history, current, or future rows that have overlapping time periods cannot have the same value in the columns.
- For a nonsequenced constraint this means that the value of the column is unique in every row in the table, irrespective of whether row time periods overlap. This is similar to a unique or primary key constraint in a non-temporal table.

In all cases, if the table has a transaction-time column, the constraint is applied only to rows that are open in transaction time.

For a current or sequenced PRIMARY KEY or UNIQUE constraint defined on a valid-time table, the valid-time column must be defined as NOT NULL.

The PK or UNIQUE columns cannot include the valid-time or transaction-time columns.

Because PK and unique constraints on temporal tables are implemented as system-defined join indexes or unique secondary indexes, the constraint is not allowed if it would cause the maximum number of secondary indexes to be exceeded for the table.

column_name

Specifies a column in the column set to be used as the primary key or as unique. The column cannot be a valid-time or transaction-time column.

Table Level REFERENCES Definition**CURRENT TRANSACTIONTIME**

Specifies that every value for the constrained column in the child table must exist in the open rows of the parent table.

SEQUENCED TRANSACTIONTIME

Specifies that any value for the constrained column in the child table must exist in the parent table during the time period for which it exists in the child.

NONSEQUENCED TRANSACTIONTIME

Specifies that any value for the constrained column in the child table must exist in the parent table. A nonsequenced constraint is like a nontemporal relational constraint, and ignores the transaction-time column in the child table.

Note:

The parent table cannot have a transaction-time column.

CURRENT VALIDTIME

Specifies that any value for the constrained column in the child table must exist in the open current or future rows of the parent table. The valid-time period of the child row must be contained within the combined valid-time periods of current and future rows in the parent table that have a value that matches the child table. History rows in the child table are not checked, and history rows in the parent table are not considered.

[SEQUENCED] VALIDTIME

Specifies that any value for the constrained column in the child table must exist in the open rows of the parent table. The valid-time period of the child row must be contained within the combined valid-time periods of open rows in the parent table that have a value that matches the child table.

NONSEQUENCED VALIDTIME

Specifies that any value for the constrained column in the child table must exist in the parent table. A nonsequenced constraint is like a nontemporal relational constraint, and ignores the valid-time column in the child table.

Note:

The parent table cannot have a valid-time column.

FOREIGN KEY

Specifies a foreign key for the table.

column_name

Specifies a name for a column defined as part of the foreign key.

REFERENCES WITH NO CHECK OPTION

Specifies an integrity reference to the parent table named in *table_name*.

Referential integrity is not enforced for the specified primary key-foreign key (or unique) relationship

table_name

Specifies the name of the referenced parent table used in the referential integrity constraint definition.

column_name

Specifies a column in the column set that makes up the parent table PRIMARY KEY or UNIQUE candidate key columns. The column cannot be a valid-time or transaction-time column.

Table Level CHECK Definition

For more information on constraints, see [Using Constraints with Temporal Tables](#).

CONSTRAINT *name*

Specifies the optional name for the constraint.

Note:

Transactiontime and validtime options, if used together, can be specified in any order, separated by AND.

CURRENT TRANSACTIONTIME

Specifies that only rows that are open in transaction time are to be checked for constraint violations.

CURRENT VALIDTIME

Specifies that only rows that are current and future in valid time are to be checked for constraint violations. History rows are not checked.

This is the default temporal qualifier for constraints applied to valid-time columns.

[SEQUENCED] VALIDTIME

Specifies that rows that are history, current, and future in valid time are to be checked for constraint violations.

This is the default type of valid-time constraint applied if the VALIDTIME keyword is specified without a CURRENT, SEQUENCED, or NONSEQUENCED prefix.

NONSEQUENCED VALIDTIME

Specifies that rows that are history, current, and future in valid time are to be checked for constraint violations. Nonsequenced constraints treat the valid-time column as a nontemporal column, so are similar to constraints on nontemporal tables. They apply to all open rows of the table.

CHECK (*boolean_condition*)

Specifies a boolean conditional expression that must be true, or else the row violates the check constraint.

Check constraints cannot be placed on valid-time or transaction-time columns.

Table Level NORMALIZE Definition

For more information on the NORMALIZE option, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

NORMALIZE

Specifies the rows are to be combined based on a specified period column, which can be a derived period column. The values for all other columns must be identical for rows to be combined. The period column values for combined rows must overlap or meet, and are coalesced to result in a single time period that is the union of the time periods for the combined rows.

The period column for normalization can be a derived period column.

Note:

NORMALIZE is not valid with CREATE TABLE AS. However, if you are copying a table that is normalized, and no further column specifications are given in the CREATE TABLE AS statement, the resulting table copy will be normalized as the original.

ALL BUT (*ignored_column_list*)

Specifies a column name or comma-separated list of column names that are to be ignored for purposes of the normalization. When rows are inserted or updated in the table, the value of the ignored columns in the result is non-deterministic. These values may reflect values from rows that have been recently added to the table, but precise values cannot be predicted nor necessarily reproduced.

Note:

Although temporal tables with transaction time can be normalized, the values in the transaction-time column are likely to be unique. Therefore most table rows will not qualify for normalization unless the transaction-time column is included in the *ignored_column_list*. Normalization is applicable to all open rows

ON *normalize_column*

Specifies the period or derived period column to be normalized. This cannot be a transaction-time column, but can be a valid-time or other period column.

Note:

When normalization is performed on tables having transaction time, only open rows qualify for normalization.

ON OVERLAPS

Specifies the normalize condition is that the period values for rows that will be coalesced must overlap. ON OVERLAPS OR MEETS is the default for NORMALIZE.

OR MEETS

Specifies the normalize conditions is that the period values for rows that will be coalesced must meet. ON OVERLAPS OR MEETS is the default for NORMALIZE.

Usage Notes

Primary Index and Primary AMP Index

Primary indexes for temporal tables must be non-unique (NUPIs).

Temporal columns, or the component columns of temporal derived period columns, cannot be included in primary indexes or primary AMP indexes.

If the CREATE TABLE statement for a nonpartitioned table does not specify either PRIMARY INDEX, PRIMARY AMP INDEX, or NOPI, automatic creation of a primary index is controlled by the setting of DBS Control field PrimaryIndexDefault. For more information see *Teradata Vantage™ - Database Utilities*, B035-1102.

If the CREATE TABLE statement for a partitioned table does not specify either PRIMARY INDEX, PRIMARY AMP INDEX, or NO PRIMARY INDEX, the table will be created as a NoPI table.

Resource Consumption

Because temporal DML statements may insert new rows into a table or logically delete rows from a table, a temporal table occupies more space than a nontemporal table. Additionally, tables with transaction time grow monotonically because rows are never physically deleted and removed from these tables (unless rows are explicitly removed using the `NONTEMPORAL DELETE` statement, which requires special privileges).

The increased number of row operations means that the `SELECT` and DML statements on temporal tables tend to be more resource intensive when compared to nontemporal tables. Because constraints defined on temporal tables are time aware, constraint checking is also more resource intensive on temporal tables than on nontemporal tables.

To mitigate the performance impacts from operations on temporal tables:

- Row partition temporal tables. Use the partitioning expressions recommended in this section.
- Define appropriate join indexes on temporal tables.

Derived Period Columns and `CREATE TABLE`

In addition to the regular rules for the use of derived period columns, which are described in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144, the following rules and restrictions apply to the use of derived `VALIDTIME` and `TRANSACTIONTIME` columns in temporal tables.

- The component columns of a derived period column used as a temporal column cannot be part of the primary index.
- When a derived period column is defined as a `TRANSACTIONTIME` column, neither of the component columns can be set by the user to `NULL`, neither of the columns can have user-specified default value, and both must have a data type of `TIMESTAMP(6) WITH TIME ZONE`.
- All rules for regular `VALIDTIME` and `TRANSACTIONTIME` columns apply also to the component columns of a derived period column used as a temporal column.
- `UNTIL_CHANGED` can be specified as one of the values to be compressed by multivalue compression for *end_column* for a `VALIDTIME` derived period column.
- If the `TRANSACTIONTIME` column is a derived period column, and if data for the *begin_column* or *end_column* for imported data contains leap seconds, the seconds portion is adjusted to 59.999999 with the precision truncated to the described precision for the input data. During this process, if the beginning and ending bounds of a transaction-time period become the same, the system generates an error.

For example, if the *begin_column* value is `TIMESTAMP '2006-12-31 23:59:59.999'` and the corresponding *end_column* value is `TIMESTAMP '2006-12-31 23:59:60.123'`, the ending bound value is adjusted to `TIMESTAMP '2006-12-31 23:59:59.999'` which is the same as the beginning bound, so the system generates an error.

- If the source table of a CREATE TABLE AS statement has any derived period columns, the target table that is created will have the same derived period columns.
- If a SEQUENCED qualifier is specified in a CREATE TABLE AS subquery, and the source is a temporal table with derived period columns for VALIDTIME or TRANSACTIONTIME, the resulting target table is a temporal table with a period data type column acting as a VALIDTIME or TRANSACTIONTIME column.

CREATE TABLE AS and Temporal Tables

CREATE TABLE AS, the syntax for copying all or a portion of a table, can be used to create a temporal or nontemporal table from a source temporal table. The syntax for CREATE TABLE AS is identical, whether used with a temporal or nontemporal source table. The syntax is described fully in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. However, note the following information, which is specific to using a temporal source table.

- If an AS subquery of a temporal table is used without a temporal qualifier the default temporal qualifier set for the session is used for the subquery.
- New columns defined for a table in the CREATE TABLE AS statement cannot be VALIDTIME or TRANSACTIONTIME columns.
- If the source table is a temporal table with a valid-time column, the target table created is a temporal table that has a valid-time column. In this case the following apply:
 - If WITH DATA is specified in the CREATE TABLE AS statement, the valid-time column values are copied to the target table.
 - Target tables can be created with column- and table-level temporal constraints. For tables with valid time, primary key and unique constraints automatically create system-defined join indexes (SJIs). These types of constraints can be defined on target tables in CREATE TABLE AS statements only if the WITH NO DATA option is used. (Note that NONSEQUENCED VALIDTIME primary key and unique constraints act as for nontemporal table, and create unique secondary indexes, rather than SJIs.)
- If the source table is a temporal table with a transaction-time column, the target table created is a temporal table that has a transaction-time column. In this case, the following apply:
 - The transaction-time values are copied to the target table only if the NONTEMPORAL qualifier is used with CREATE TABLE AS. (Note that the NONTEMPORAL privilege is required to use the NONTEMPORAL qualifier.) The NONTEMPORAL qualifier is ignored if the source table does not have a transaction-time column.
 - If the NONTEMPORAL qualifier is not used, the transaction-time values in the target table default to (TEMPORAL_TIMESTAMP, UNTIL_CLOSED).
 - Target tables can be created with column- and table-level temporal constraints. These constraints are applicable only to open rows. Target tables with transaction-time can only have primary key and unique constraints, if the WITH NO DATA option is used.

Related Information

For more information on...	See...
CREATE TABLE (regular form), CREATE TABLE AS	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
creating temporal tables	Creating Temporal Tables .
constraints and temporal tables	Using Constraints with Temporal Tables .
NONTEMPORAL qualifier and privilege	Nontemporal Operations and Usage Notes .
partitioning temporal tables	Partitioning Expressions for Temporal Tables .
transaction time and valid time	Transaction Time and Valid Time .

CREATE TRIGGER/REPLACE TRIGGER (Temporal Form)

CREATE TRIGGER creates a trigger on a subject table that is a temporal table.

REPLACE TRIGGER redefines an existing trigger or, if the specified trigger does not exist, creates a new trigger with the specified name.

CREATE TRIGGER/REPLACE TRIGGER Syntax (Temporal Form)

Note:

Teradata Vantage™ - Temporal Table Support describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
{ CREATE | REPLACE } TRIGGER [ database_name. ] trigger_name
[ ENABLED | DISABLED ]
{ BEFORE | AFTER }
[ CURRENT | SEQUENCED | NONSEQUENCED ] VALIDTIME ] | NONTEMPORAL ]
{ INSERT |
  DELETE |
  UPDATE [ OF { column_name [,...] | ( column_name [,...] ) } ]
}
ON [ database_name. ] table_name [ ORDER integer ]
[ REFERENCING reference_spec [...] ]
[ FOR EACH { ROW | STATEMENT } ]
```

```
[ WHEN ( search_condition ) ]
{ SQL_proc_spec | BEGIN ATOMIC SQL_proc_spec END } [;]
```

Note:

To ensure application portability to ANSI standards for temporal SQL, Teradata recommends explicit specification of all temporal qualifiers.

reference_spec

```
{ OLD [ROW] [AS] old_transition_variable_name |
  NEW [ROW] [AS] new_transition_variable_name |
  { OLD_TABLE | OLD TABLE } [AS] old_transition_table_name |
  { NEW_TABLE | NEW TABLE } [AS] new_transition_table_name |
  OLD_NEW_TABLE [AS] old_new_table_name ( old_value, new_value )
}
```

SQL_proc_spec

```
{ SQL_procedure_statement; [...] | ( SQL_procedure_statement; [...] ) }
```

CREATE TRIGGER/REPLACE TRIGGER Syntax Elements (Temporal Form)

database_name

Specifies an optional qualifier for *trigger_name*.

trigger_name

Specifies the name of the trigger to be created or replaced.

ENABLED

Specifies the keyword that enables a trigger to execute.

DISABLED

Specifies the keyword that disables a trigger from executing.

BEFORE

Specifies that the trigger performs before the triggering event, or triggering statement, is executed.

AFTER

Specifies that the trigger performs after the triggering event.

CURRENT VALIDTIME

Specifies that the trigger fires when the triggering event is current in the valid-time dimension.

The subject table must have valid time.

VALIDTIME

Specifies that the trigger fires when the triggering event is sequenced in the valid-time dimension.

The subject table must have valid time.

SEQUENCED VALIDTIME**NONSEQUENCED VALIDTIME**

Specifies that the trigger fires when the triggering event is nonsequenced in the valid-time dimension.

The subject table must have valid time.

NONTEMPORAL

Specifies that the trigger fires when the triggering event is nontemporal in the transaction-time dimension. In this case, the triggering statement must specify the NONTEMPORAL prefix.

The triggered action statement can modify values in the transaction-time column.

The subject table must have transaction time. If the subject table has valid time, the qualifier in the valid-time dimension defaults to NONSEQUENCED VALIDTIME.

The NONTEMPORAL privilege is required to use the NONTEMPORAL option.

INSERT

Specifies that the triggering event for this trigger is one of the following:

- INSERT
- INSERT SELECT
- Atomic Upsert
- MERGE

Note:

Triggers cannot be fired for rows that have been normalized. For more information on the NORMALIZE keyword, see [CREATE TABLE/CREATE TABLE AS \(Temporal Forms\)](#) and *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

DELETE

Specifies that the triggering event for this trigger is a DELETE.

UPDATE

Specifies that the triggering statement for this trigger is one of the following:

- UPDATE
- Atomic Upsert
- MERGE

Any number of rows, including none, can be updated.

Note:

Triggers cannot be fired for rows that have been normalized. For more information on the NORMALIZE keyword, see [CREATE TABLE/CREATE TABLE AS \(Temporal Forms\)](#) and *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

ON [database_name.]table_name

Specifies the subject table with which this trigger is associated.

If the subject table has valid time, and the triggering statement is not preceded by a valid-time qualifier, the default is CURRENT VALIDTIME. If the subject table has transaction time, and the triggering statement is not preceded by NONTEMPORAL, the default is CURRENT TRANSACTIONTIME.

ORDER integer

Specifies the order of trigger execution within a request when multiple triggers are defined on a subject table.

REFERENCING

Specifies a transition table that the WHEN condition and the triggered actions of a trigger can reference. The clause is optional and has no default.

FOR EACH ROW

Specifies keywords specifying that the trigger is to fire for each qualified row. That is, each row that evaluates to TRUE for any WHEN condition specified for the trigger.

FOR EACH STATEMENT

Specifies keywords specifying that the trigger is to fire once per processed SQL statement in the request whenever a WHEN condition for the trigger evaluates to TRUE.

WHEN (*search_condition*)

Specifies a search condition clause to refine the conditions that fire the trigger.

SQL_procedure_statement

Specifies one or more valid triggered action statements.

If the subject table is a nontemporal table and the trigger action references a temporal table, a current qualifier is applied to the trigger action statements.

If the subject table is a temporal table, all of the triggered action statements inherit the qualifier of the triggering statement. If an action requires a different qualifier, include the statement in a stored procedure and call the stored procedure in an action statement.

BEGIN ATOMIC

Specifies a keyword introducing multiple triggered action statements.

If you begin the triggered SQL statement clause with BEGIN ATOMIC, then you must also terminate it with the END keyword.

Usage Notes

Recursive Triggers and Multiset Tables

Temporal tables are multiset tables. If the subject table is referenced in the action, due to the multiset nature of the subject table, recursive triggers can hit the recursion limit and cause an error to be reported.

Related Information

For more information on...	See...
CREATE TRIGGER (regular form)	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144
NONTEMPORAL operations	<ul style="list-style-type: none"> • Nontemporal Operations • Usage Notes

CREATE VIEW/REPLACE VIEW (Temporal Forms)

CREATE VIEW defines a view on a set of tables or views or both.

REPLACE VIEW redefines an existing view or, if the specified view does not exist, creates a new view with the specified name.

CREATE VIEW/REPLACE VIEW Syntax (Temporal Forms)

Note:

Teradata Vantage™ - Temporal Table Support describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
{ CREATE | REPLACE } VIEW
  [ database_name. | user_name. ] view_name [ ( column_name [,...] ) ]
  AS { [ temporal_qualifier ] select_statement |
      ( [ temporal_qualifier ] select_statement )
    } [;]
```

Note:

To ensure application portability to ANSI standards for temporal SQL, Teradata recommends explicit specification of all temporal qualifiers.

temporal_qualifier

```
{ valid_time_qualifier [ AND transaction_time_qualifier ] |
  transaction_time_qualifier [ AND valid_time_qualifier ] |
  AS OF date_timestamp_expression
}
```

valid_time_qualifier

```
{ CURRENT VALIDTIME |
  VALIDTIME AS OF date_timestamp_expression |
```

```
{ [ SEQUENCED | NONSEQUENCED ] VALIDTIME } [ period_expression ]
}
```

transaction_time_qualifier

```
{ { CURRENT | SEQUENCED | NONSEQUENCED } TRANSACTIONTIME |
  TRANSACTIONTIME AS OF date_timestamp_expression
}
```

CREATE VIEW/REPLACE VIEW Syntax Elements (Temporal Forms)

view_name

Specifies the name of the view.

database_name**user_name

Specifies the name of the database or user to contain *view_name* if something other than the current database or user.

column_name

Specifies the name of a view column. If more than one column is specified, list their names in the order in which each column is to be displayed for the view.

CURRENT VALIDTIME

Specifies that *select_statement* is a current query in the valid-time dimension. The result set is a nontemporal table.

VALIDTIME AS OF *date_timestamp_expression*

Specifies that *select_statement* retrieves rows where the period of validity overlaps the specified AS OF expression. The result set is a nontemporal table.

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

VALIDTIME SEQUENCED VALIDTIME

Specifies that *select_statement* is a sequenced query. The result set is a valid-time table that includes an extra column for the overlapped valid-time period. If the list of columns for the view does not provide a name for the extra column, the default name is "VALIDTIME".

If *select_statement* is a sequenced query in the valid-time and transaction-time dimensions, and the list of columns for the view provides names for the extra columns, the first extra name is the name of the resulting valid-time column and the second extra name (if it exists) is the name of the resulting transaction-time column.

NONSEQUENCED VALIDTIME

Specifies that *select_statement* is a nonsequenced query in the valid-time dimension. If *period_expression* is specified, the nonsequenced query produces a table with valid time; otherwise, the result set is a nontemporal table.

The result set for a valid-time table includes an extra column for the overlapped valid-time period. If the list of columns for the view does not provide a name for the extra column, the default name is "VALIDTIME".

If *select_statement* is a nonsequenced query in the valid-time dimension and sequenced in the transaction-time dimension, and the list of columns for the view provides names for the extra columns, the first extra name is the name of the resulting valid-time column and the second extra name (if it exists) is the name of the resulting transaction-time column.

period_expression

Specifies the period of applicability for the sequenced or nonsequenced query.

CURRENT TRANSACTIONTIME

Specifies that *select_statement* is a current query in the transaction-time dimension. The result set is a nontemporal table.

TRANSACTIONTIME AS OF *date_timestamp_expression*

Specifies that *select_statement* retrieves rows whose transaction-time period in the row overlaps the specified AS OF expression. The result set is a nontemporal table.

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

SEQUENCED TRANSACTIONTIME

Specifies that *select_statement* is a sequenced query in the transaction-time dimension. The result set is a table with transaction time.

There can only be one table referenced in *select_statement* and the table must have transaction time.

NONSEQUENCED TRANSACTIONTIME

Specifies that *select_statement* is a nonsequenced query in the transaction-time dimension. A nonsequenced query produces a nontemporal table as a result set.

AS OF *date_timestamp_expression*

Specifies that *select_statement* retrieves rows whose valid-time and transaction-time periods overlap the specified AS OF expression.

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

select_statement

Specifies the existing SELECT statement syntax.

Usage Notes

SELECT Statement

All rules that apply to the temporal form of the SELECT statement are applicable for queries specified in views.

If no temporal qualifier is specified for the view and the view references any temporal tables, the temporal qualifier defaults to the applicable session temporal qualifier.

Selecting from a View on a Temporal Table

A query that selects from a view can specify a temporal qualifier that is different from the temporal qualifier of the SELECT statement in the view definition.

Updatable Views

Views created on temporal tables are said to be updatable if they satisfy all the existing rules of updatable views on nontemporal tables and:

- For a view on a valid-time table or on another view, the valid-time qualifier must be SEQUENCED VALIDTIME with the restriction that it must not specify a period of applicability.
- For a view on a transaction-time table or on another view, the transaction-time qualifier must be SEQUENCED TRANSACTIONTIME.
- For a view on a bitemporal table or on another view, the valid-time qualifier must be SEQUENCED VALIDTIME and the transaction-time qualifier must be SEQUENCED TRANSACTIONTIME.
- The table must not be specified with an AS OF clause.

Sequenced updatable views permit current, sequenced, nonsequenced, and nontemporal DML operations. A current or sequenced form of UPDATE issued on the view modifies the valid-time and transaction-time column values. An update privilege on the valid-time column is required in addition to

those privileges normally required to perform UPDATE on an updatable view. (No additional privilege is required for the transaction-time column.)

When the WITH CHECK OPTION is specified in the updatable view and the rows are updated using the view, only the modified rows are checked for violations. Those rows that are inserted by the system as part of the temporal semantics (for example close of a row in the transaction-time dimension) are excluded from such checks as they are created to maintain temporal semantics.

Temporal Sequenced View Support for Teradata ILDMs

Vantage supports the creation of sequenced views to be used in Teradata Industry Logical Data Models (ILDMS). These views can use all existing Vantage features with a SEQUENCED qualifier. Some of the features that are often used in ILDMs are:

- sequenced outer joins
- sequenced subqueries (inclusion, exclusion joins)
- sequenced aggregation
- sequenced OLAP

Although these views are created using the SEQUENCED temporal qualifier, they can only be used in queries having point-in-time (CURRENT or AS OF) temporal qualifiers.

Related Information

For more information on...	See...
CREATE VIEW (regular form)	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144
CREATE RECURSIVE VIEW (temporal form)	CREATE VIEW/REPLACE VIEW (Temporal Forms)
temporal table views	Views on Temporal Tables

SET SESSION (Session Temporal Qualifiers)

Sets the session temporal qualifier in the valid-time dimension, transaction-time dimension, or valid-time and transaction-time dimensions.

Note:

Best practice for maximal application portability is to explicitly qualify all queries and DML that operates on temporal tables, rather than relying on session temporal default qualifiers.

SET SESSION Syntax (Session Temporal Qualifiers)

Note:

Teradata Vantage™ - Temporal Table Support describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
{ SET SESSION | SS }
  { valid_time_qualifier [ AND transaction_time_qualifier ] |

    transaction_time_qualifier [ AND valid_time_qualifier ] |

    AS OF date_timestamp_expression |

    ANSIQUALIFIER
  } [;]
```

valid_time_qualifier

```
{ CURRENT VALIDTIME |

  VALIDTIME AS OF date_timestamp_expression |

  { [ SEQUENCED | NONSEQUENCED ] VALIDTIME } [ period_expression ]
}
```

transaction_time_qualifier

```
{ { CURRENT | NONSEQUENCED } TRANSACTIONTIME |

  TRANSACTIONTIME AS OF date_timestamp_expression
}
```

SET SESSION Syntax Elements (Session Temporal Qualifiers)

CURRENT VALIDTIME

Specifies that the session valid-time qualifier is current.

VALIDTIME AS OF *date_timestamp_expression*

Specifies that when a SELECT statement refers to a temporal table but omits a valid-time qualifier, the default behavior extracts a snapshot of the table in the valid-time dimension AS OF the date or timestamp value specified by *date_timestamp_expression*.

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

VALIDTIME SEQUENCED VALIDTIME

Specifies that the session valid-time qualifier is sequenced.

NONSEQUENCED VALIDTIME

Specifies that the session valid-time qualifier is nonsequenced.

period_expression

Specifies the period of applicability for the sequenced or nonsequenced session valid-time qualifier. The period value must be a constant expression or built-in function and cannot reference any columns.

CURRENT TRANSACTIONTIME

Specifies that the session transaction-time qualifier is current.

TRANSACTIONTIME AS OF *date_timestamp_expression*

Specifies that when a SELECT statement refers to a temporal table but omits a transaction-time qualifier, the default behavior extracts a snapshot of the table in the transaction-time dimension AS OF the date or timestamp value specified by *date_timestamp_expression*.

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

NONSEQUENCED TRANSACTIONTIME

Specifies that the session transaction-time qualifier is nonsequenced.

AS OF *date_timestamp_expression*

Specifies that when a SELECT statement refers to a temporal table but omits the valid-time qualifier, transaction-time qualifier, or both, the default behavior extracts a snapshot of the

table in the valid-time dimension, transaction-time dimension, or both dimensions AS OF the value specified by *date_timestamp_expression*.

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

ANSIQUALIFIER

This is the default for users of ANSI/ISO-compatible temporal tables and syntax.

The session temporal qualifier must be ANSIQUALIFIER in order to create ANSI-compatible temporal tables and to get ANSI-compatible database behavior for temporal queries and DML.

The session qualifier must also be ANSIQUALIFIER in order to convert existing temporal tables described in this document to ANSI-compatible temporal tables described in *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186. For more information on converting transaction-time tables to system-time tables, see [ANSI Temporal Tables](#).

Usage Notes

When a SELECT or DML statement refers to a temporal table but does not explicitly specify a valid-time qualifier, the system uses the session valid-time qualifier. Similarly, when a SELECT statement refers to a temporal table but does not explicitly specify a transaction-time qualifier, the system uses the session transaction-time qualifier.

The current session temporal qualifier is reported by the Temporal Qualifier field returned by the HELP SESSION statement.

The default temporal session qualifier depends on whether you use the original Teradata implementation of temporal tables and syntax, as described in this document, or the ANSI/ISO-compliant implementation of temporal tables, as described in *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186. This default is determined by Temporal Behavior setting in DBS Control, and is set by Teradata personnel in accordance with your use of temporal tables and syntax. For more information on DBS Control, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Related Information

For more information on...	See...
SET SESSION (regular form)	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144
session temporal qualifier	Session Temporal Qualifiers
ANSI/ISO-compatible temporal tables and syntax	<i>Teradata Vantage™ - ANSI Temporal Table Support</i> , B035-1186

SET SESSION TTGRANULARITY TO

Can be used to set the granularity of timestamping used for rows in temporal tables with transaction time.

SET SESSION TTGRANULARITY TO Syntax

```
SET SESSION TTGRANULARITY TO { LOGICALROW | REQUEST | TRANSACTION }
```

Syntax Elements

LOGICALROW

Row is timestamped with the time the row is processed by the AMP.

REQUEST

Row is timestamped with the time the request is submitted.

TRANSACTION

Row is timestamped with the time when the first non-locking reference is made to a temporal table, or when the built-in function `TEMPORAL_TIMESTAMP` is first accessed during the transaction.

Usage Notes

`SET SESSION TTGRANULARITY` is effective only in Teradata session mode.

The setting of `SET SESSION TTGRANULARITY` does not persist across system restarts.

`SS` may be used as a synonym for `SET SESSION`.

Related Information

For more information on...	See...
Timestamping rows in temporal tables	Timestamping
Teradata session modes	<ul style="list-style-type: none"> • <i>Teradata Vantage™ - SQL Fundamentals</i>, B035-1141 • <i>Teradata Vantage™ - SQL Request and Transaction Processing</i>, B035-1142

Usage Notes for Temporal Tables

Partitioning Expressions for Temporal Tables

To improve the performance of current queries on a temporal table, the table should be row partitioned:

- For tables with valid-time columns, the partitions logically separate the current and future rows from the history rows, so fewer rows need to be scanned for current queries.
- For tables with transaction-time columns, the partitions logically separate the open rows from the closed rows.
- For bitemporal tables, the partitions separate open current and future rows from open history rows from all closed rows.

Note:

Like other types of tables, temporal tables can have several levels of partitioning defined, including column partitioning. The temporal row partitioning described here should be one of those levels. For more information on column partitioning, see *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

The recommended partitioning expressions discussed for each table type below reference the built-in functions `CURRENT_DATE`, `CURRENT_TIMESTAMP`, or both.

As time passes, the values of `CURRENT_DATE` and `CURRENT_TIMESTAMP` differ from the values that were used to resolve `CURRENT_DATE` and `CURRENT_TIMESTAMP` in the partitioning expressions when the table was created. Because of this, the partition intended to hold the current and future rows might include rows that are not strictly current. The optimizer can nevertheless successfully find current rows within the partition. The presence of some history rows in the current partition will not adversely affect performance unless there are a great many history rows there.

Use the `ALTER TABLE TO CURRENT` statement periodically to move history rows out of the current partition into the history partition. `ALTER TABLE TO CURRENT` resolves the partitioning expressions again, transitioning rows to their appropriate partitions per the updated partitioning expressions. For more information on `ALTER TABLE TO CURRENT`, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Note:

To ensure optimal row partition elimination, the granularity of the time specification in `AS OF` and `SEQUENCED` queries on row partitioned temporal tables should be no finer than the valid-time column in the query with the coarsest time granularity.

Row Partitioning Valid-Time Tables

The following table describes the recommended partitioning expressions for a valid-time table, where *vtcolumn* represents the valid-time column.

Valid-Time Column Data Type	Recommended Partitioning Expressions
PERIOD(DATE) NOT NULL	<pre>PARTITION BY CASE_N(END(<i>vtcolumn</i>) >= CURRENT_DATE AT '-12:59', NO CASE)</pre> <p>(where AT '-12:59' is a shorthand form of AT INTERVAL '-12:59' HOUR TO MINUTE)</p> <p>Note: If the valid-time column is a derived period column, the component columns must be defined as NOT NULL, and the derived period column is considered to be defined as NOT NULL.</p>
PERIOD(DATE)	<pre>PARTITION BY CASE_N(END(<i>vtcolumn</i>) IS NULL OR END(<i>vtcolumn</i>) >= CURRENT_DATE AT '-12:59', NO CASE)</pre> <p>(where AT '-12:59' is a shorthand form of AT INTERVAL '-12:59' HOUR TO MINUTE)</p>
PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) NOT NULL	<pre>PARTITION BY CASE_N(END(<i>vtcolumn</i>) >= CURRENT_TIMESTAMP, NO CASE)</pre>
PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE])	<pre>PARTITION BY CASE_N(END(<i>vtcolumn</i>) IS NULL OR END(<i>vtcolumn</i>) >= CURRENT_TIMESTAMP, NO CASE)</pre>

By using the recommended physical partitioning for a valid-time table, the physical partitions are as follows:

- The current partition has rows that are or were valid as of the last resolved `CURRENT_DATE` or `CURRENT_TIMESTAMP` value for the partitioning expression and rows that are in the future with respect to that date or timestamp.

Note that rows with a valid-time column value as NULL are in the current partition (but are not considered current or valid rows).

- The history partition has rows that were no longer valid as of the last resolved `CURRENT_DATE` or `CURRENT_TIMESTAMP` value for the partitioning expression.

Note:

Most of the AS OF queries are concerned with times that have already past. Consequently, these queries will not benefit from the recommended partitioning expressions described above. If AS OF queries are expected to be frequent, one way to get the benefit of row partition elimination is to partition on `END(<VT_column>)` and, within each partition, by `BEGIN(<VT_column>)` such that there is a 20% distribution within each outer partition.

Row Partitioning Transaction-Time Tables

The following partitioning expression is recommended for a transaction-time table, where *ttcolumn* represents the transaction-time column.

```
PARTITION BY CASE_N (END(ttcolumn) >= CURRENT_TIMESTAMP, NO CASE)
```

By using the recommended physical partitioning for a transaction-time table, the physical partitions are as follows:

1. The current partition has rows that are or were open as of the last resolved `CURRENT_TIMESTAMP` value for the partitioning expression.
2. The history partition has rows that were closed as of the last resolved `CURRENT_TIMESTAMP` value for the partitioning expression.

Row Partitioning Bitemporal Tables

The following table describes the recommended partitioning expressions for a bitemporal table, where *vtcolumn* represents the valid-time column and *ttcolumn* represents the transaction-time column.

Valid-Time Column Data Type	Recommended Partitioning Expressions
PERIOD(DATE) NOT NULL	<pre>PARTITION BY CASE_N(END(vtcolumn) >= CURRENT_DATE AT '-12:59' AND END(ttcolumn) >= CURRENT_TIMESTAMP, END(vtcolumn) < CURRENT_DATE AT '-12:59' AND END(ttcolumn) >= CURRENT_TIMESTAMP, END(ttcolumn) < CURRENT_TIMESTAMP)</pre> <p>(where AT '-12:59' is a shorthand form of AT INTERVAL '-12:59' HOUR TO MINUTE) (The <code>END(ttcolumn) < CURRENT_TIMESTAMP</code> expression represents closed rows, and is used instead of NO CASE to achieve better partition elimination.)</p>
PERIOD(DATE)	<pre>PARTITION BY CASE_N((END(vtcolumn) IS NULL OR END(vtcolumn) >= CURRENT_DATE AT '-12:59')</pre>

Valid-Time Column Data Type	Recommended Partitioning Expressions
	<pre> AND END(<i>ttcolumn</i>) >= CURRENT_TIMESTAMP, END(<i>vtcolumn</i>) < CURRENT_DATE AT '-12:59' AND END(<i>ttcolumn</i>) >= CURRENT_TIMESTAMP, END(<i>ttcolumn</i>) < CURRENT_TIMESTAMP) </pre> <p>(where AT '-12:59' is a shorthand form of AT INTERVAL '-12:59' HOUR TO MINUTE)</p>
PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) NOT NULL	<pre> PARTITION BY CASE_N(END(<i>vtcolumn</i>) >= CURRENT_TIMESTAMP AND END(<i>ttcolumn</i>) >= CURRENT_TIMESTAMP, END(<i>vtcolumn</i>) < CURRENT_TIMESTAMP AND END(<i>ttcolumn</i>) >= CURRENT_TIMESTAMP, END(<i>ttcolumn</i>) < CURRENT_TIMESTAMP) </pre>
PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE])	<pre> PARTITION BY CASE_N((END(<i>vtcolumn</i>) IS NULL OR END(<i>vtcolumn</i>) >= CURRENT_ TIMESTAMP) AND END(<i>ttcolumn</i>) >= CURRENT_TIMESTAMP, END(<i>vtcolumn</i>) < CURRENT_TIMESTAMP AND END(<i>ttcolumn</i>) >= CURRENT_TIMESTAMP, END(<i>ttcolumn</i>) < CURRENT_TIMESTAMP) </pre>

By using the recommended row partitioning for a bitemporal table, the partitions are as follows:

- The current partition has rows that are or were valid and open as of the last resolved CURRENT_DATE or CURRENT_TIMESTAMP value for the partitioning expression.
- The valid-time history/transaction-time open partition has rows that are or were no longer valid but were still open as of the last resolved CURRENT_DATE or CURRENT_TIMESTAMP value for the partitioning expression.
- The transaction-time history partition has rows that were closed as of the last resolved CURRENT_DATE or CURRENT_TIMESTAMP value for the partitioning expression.

Related Information

For more information on...	See...
CREATE TABLE (temporal form)	CREATE TABLE/CREATE TABLE AS (Temporal Forms)
ALTER TABLE TO CURRENT (regular form)	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144
Multilevel partitioned primary indexes	<ul style="list-style-type: none"> • CREATE TABLE statement in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144 • <i>Teradata Vantage™ - Database Design</i>, B035-1094

Collecting Statistics on Current Data

Statistics can be collected on current data at regular intervals. This is more efficient than collecting data on all data in the table, especially when new data is loaded, and global statistics are refreshed, which can require reading and aggregating the entire table, both current and historical data.

While global statistics are useful to help optimize queries that involve both current and historical data, collecting these statistics can occur at a higher threshold of data change than statistics collection limited to current data. For example, you could refresh statistics on current data whenever the change involves more than 10% of the data, while the threshold for global statistics collection could be 20%.

Example: Collecting statistics on current data

This example shows how statistics could be collected on a bitemporal table that has been partitioned to separate the current from invalid and closed data.

```
CREATE MULTISET TABLE Policy_BiTemp (
  Policy_ID INTEGER,
  Customer_ID INTEGER,
  Policy_Type CHAR(2) NOT NULL,
  Policy_Details CHAR(40),
  Validity PERIOD(DATE) NOT NULL AS VALIDTIME
  Policy_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
  AS TRANSACTIONTIME
)
PRIMARY INDEX (Policy_ID)
PARTITION BY CASE_N(
  (END(Validity) IS NULL OR
  END(Validity) >= CURRENT_DATE -INTERVAL '2' DAY) AND
  END(Policy_Duration) >= CURRENT_TIMESTAMP,
  END(Validity) < CURRENT_DATE-INTERVAL '2' DAY AND
  END(Policy_Duration) >= CURRENT_TIMESTAMP,
  END(Policy_Duration) < CURRENT_TIMESTAMP);
```

The first partition created by the CASE_N expression is the current partition. Because most queries of the table will involve the current partition, statistics should be collected on the current data.

```
COLLECT STATISTICS
  COLUMN Policy_ID
  ,COLUMN Customer_ID
  ,COLUMN Policy_Type
  ON QUERY (SELECT Policy_ID, Customer_ID, Policy_Type
    FROM Policy_BiTemp
    WHERE (END(Validity) IS NULL OR
```

```

        END(Validity) >= CURRENT_DATE -
        INTERVAL '2' DAY)
    AND END(Policy_Duration) >=
        CURRENT_TIMESTAMP
    ) AS QS_Policy_Bitemp_Currdata;

```

For more information about COLLECT STATISTICS, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Using Constraints with Temporal Tables

Column or table level constraints defined on temporal tables can be associated with a time dimension, and are of three basic types:

Temporal Constraint Form	Description
CURRENT VALIDTIME	The constraint is applied to all current and future rows. History rows are not checked for constraint violations.
SEQUENCED VALIDTIME	The constraint is applied to all future, current, and open history rows and ensures that the constraint is not violated at any instant of time in the valid-time dimension.
NONSEQUENCED VALIDTIME	The constraint is applicable to all open rows in the table, and treats the valid-time column as a regular, nontemporal column. A nonsequenced constraint on a valid-time table is similar in semantics to a constraint defined on a nontemporal table.

For tables with transaction-time columns, nonreferential constraints are always considered to be current in the transaction-time dimension. These constraints are enforced only on rows that are open in transaction time. For bitemporal tables, constraints are enforced only on open rows that satisfy the valid-time constraint temporal qualifier.

Because constraints are not allowed on columns with a period data type, constraints are not allowed on valid-time or transaction-time columns.

Note:

To ensure application portability to ANSI standards for temporal SQL, Teradata recommends explicit specification of all temporal qualifiers.

Check Constraints

Check constraints ensure that the value in a constrained column always satisfies the boolean expression that was specified when the check constraint was defined.

- **CURRENT VALIDTIME CHECK** constraints are enforced only on rows that are current and future in valid time. Consequently, modifications to rows that are history rows in valid time are not checked for constraint violations.
- **SEQUENCED VALIDTIME CHECK** constraints are enforced on history, current, and future rows in valid time.
- **NONSEQUENCED VALIDTIME CHECK** constraints treat the valid-time column as a nontemporal column, so act like a check constraint on a nontemporal table, and are enforced on all open rows of the table. This means nonsequenced and sequenced check constraints have identical effects on temporal tables.

Primary Key and Unique Constraints

Primary key and unique constraints impose uniqueness on column values amongst the rows of a table.

- **CURRENT VALIDTIME PRIMARY KEY** and **CURRENT VALIDTIME UNIQUE** constraints ensure that the value for the constrained column in a row is unique for all instances of time from current time through the future. Current and future rows that have overlapping valid-time periods are prevented from having the same value in the constrained columns.
- **SEQUENCED VALIDTIME PRIMARY KEY** and **SEQUENCED VALIDTIME UNIQUE** constraints ensure that the value for the constrained column in a row is unique for all instances of time, including history, current, and future. Any rows that have overlapping valid-time periods are prevented from having the same value in the constrained columns.
- **NONSEQUENCED VALIDTIME PRIMARY KEY** and **NONSEQUENCED VALIDTIME UNIQUE** constraints treat the valid-time column as a nontemporal column. These constraints ensure that the value for the constrained column in a row is unique amongst all rows in the table. All open rows are prevented from having the same value in the constrained columns.

Note:

Nonsequenced PK/unique constraints are identical to PK/unique constraints on nontemporal tables. These types of constraints are rarely useful on temporal tables, and are not recommended. Due to the near duplication of rows that happens automatically as rows are modified in temporal tables, a nonsequenced PK/unique constraint would be violated very quickly. The same situation is true for USIs applied to temporal tables, which are also not recommended.

Example: Primary key and unique constraints

Assume that the **TEMPORAL_DATE** is November 2, 2006 (2006/11/02) and a valid-time table is defined with columns: **Col1**, **Col2**, and **VTCol** where **VTCol** is a valid-time column. Assume further that a **CURRENT VALIDTIME UNIQUE** constraint is defined on **Col2**. The following row:

Col1	Col2 (unique)	VTCol
5	24	('2006/10/20', '2007/10/20')

does not violate the constraint with the row:

6	24	('2008/01/20', '9999/12/31')
---	----	------------------------------

because the valid-time periods do not overlap. The same first row would violate the current unique constraint with the row:

7	24	('2007/09/20', '9999/12/31')
---	----	------------------------------

because the time periods overlap from 2007/09/20 to 2007/10/20.

If the table also had a transaction-time as the fourth column, the following row:

Col1	Col2	VTCol	TTCol
8	24	('2008/01/20', '9999/12/31')	('2006/09/20', '2006/09/25')

would not violate the current constraint because the row is closed in transaction time (has an end date prior to UNTIL_CLOSED), so this row is not considered for constraint checking.

Indexes for Primary Key and Unique Constraints

Because of the way rows are duplicated as a result of modifications to temporal tables, most primary key and unique constraints defined on temporal tables are implemented by means of system-defined join indexes (SJIs). These indexes enforce the uniqueness on an appropriate subset of rows, according to whether the constraint is current, sequenced, or nonsequenced.

Example: System-defined join index

A current unique constraint on a bitemporal table causes an SJI to be created and maintained automatically from selected columns of the temporal table. The primary index of the SJI is the constrained column or columns of the temporal table. The valid-time and transaction-time columns are selected using an appropriately qualified WHERE clause that limits the rows in the index to current and future rows:

```
CREATE JOIN INDEX tablename_TJI number
AS SELECT ROWID, ConstrainedColumn, VTColumn, TTcolumn FROM tablename WHERE
END(VTColumn) ≥ CURRENT_DATE - INTERVAL '2' DAY
AND END(TTcolumn) IS UNTIL_CLOSED
PRIMARY INDEX (ConstrainedColumn);
```

INTERVAL '2' DAY is required because current rows could be inserted in a time zone that is up to two days prior to the date for the time zone in which the index is created.

SJIs are created in the same database as the constrained temporal table. They are named automatically by the system using a naming convention of *tablename_TJI number*, where *tablename* is the first 121 characters of the name of the temporal table for which a PK or unique constraint has been defined, and

number is the index ID of the constraint, a unique number that identifies the SJI. Consequently, temporal tables should be named such that the first 121 characters provides per-table uniqueness in the name.

SJIs use the same map for data distribution as the constrained temporal table. If the temporal table uses a sparse map, the SJI also shares the same colocation name as the constrained table. For more information on contiguous and sparse maps, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - Database Design*, B035-1094.

Note:

SJIs are created, maintained, and deleted automatically as needed by the system. They should not be directly modified or deleted.

The PK or unique constraint is not allowed if the associated SJI would cause the maximum permitted number of secondary indexes to be exceeded for the table.

As time passes, the values of `CURRENT_DATE` and `CURRENT_TIMESTAMP` differ from the values that were used when the SJI was created. Because of this, current and future rows in SJIs, over time, become history rows, and therefore no longer needed in the index to enforce the current constraint.

Use the `ALTER TABLE TO CURRENT` statement periodically to update SJIs and PPIs created for temporal tables. `ALTER TABLE TO CURRENT` transitions history rows out the SJIs created for current primary key and unique constraints. For more information on `ALTER TABLE TO CURRENT`, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Because nonsequenced constraints treat temporal columns as if they were nontemporal, a nonsequenced valid-time PK or unique constraint on a valid-time table is implemented automatically by making the constrained column a USI. For bitemporal tables, a PK or unique constraint must be limited to rows that are open in transaction time, so an SJI is used. The SJI uses a `WHERE` clause to select only the open rows from the transaction-time column.

Note:

Because identity columns are not allowed in join indexes, PK and unique constraints cannot be defined on identity columns in temporal tables.

Temporal Referential Constraints

Referential constraints define a relationship between two tables whereby every value in the constrained column or columns (the foreign key (FK)) of the child table must exist in the corresponding referenced columns (the primary key (PK)) of the parent table. When a referential constraint involves temporal tables, the relationship can also be defined with respect to time.

Temporal referential constraints are not enforced by Vantage so are referred to as “soft” referential integrity. Definitions of these constraints must include `WITH NO CHECK OPTION` on the child column `REFERENCES` constraint, and no uniqueness is enforced on the referenced parent table columns.

Although these constraints are not enforced, the Optimizer can use them to eliminate redundant joins and improve query performance.

Note:

It is the responsibility of the user to ensure that these constraints are not violated. For more information and examples of validating and enforcing these constraints, see [Enforcing and Validating Temporal Referential Constraints](#).

The following table describes the different types of temporal referential constraints. Note that current time for valid-time columns is the value of TEMPORAL_TIMESTAMP or TEMPORAL_DATE. For more information, see [Timestamping](#).

Temporal Referential Constraint Form	Description
CURRENT TRANSACTIONTIME	Only open rows of the child and parent tables are considered. Every FK value in the open rows of the child table must exist somewhere in the PK column of the open rows in the parent table.
CURRENT VALIDTIME	Current and future rows of the child table are considered. History rows are not considered. Every value in the child table FK column must exist in the parent table PK column for the period starting from current time through the entire future portion of the child row valid-time period. If the child row value exists in more than one row in the parent table, the valid-time periods of these parent rows, when combined, must form a single, continuous period that includes current time, and extends through the entire future portion of the child row valid-time period.
SEQUENCED TRANSACTIONTIME	Both open and closed rows of the child and parent tables are considered. Every FK value in the child table must exist in the parent table PK column during the same period as the child-row transaction-time period. If the child row value exists in more than one row in the parent table, the transaction-time periods of these parent rows, when combined, must form a single, continuous period that contains the entire transaction-time period of the child row.
SEQUENCED VALIDTIME	History, current, and future rows of the child table are considered. Every value in the child table FK column must exist in the parent table PK column during the same period as the child row valid-time period. If the child row value exists in more than one row in the parent table, the valid-time periods of these parent rows, when combined, must form a single, continuous period that contains the entire valid-time period of the child row.
NONSEQUENCED VALIDTIME or NONSEQUENCED TRANSACTIONTIME	Ignores the time dimension, and behaves like a nontemporal referential constraint. Every FK value in the child table, must exist in the parent table: <ul style="list-style-type: none"> For child tables with valid time, history, current, and future rows are considered. For child tables with transaction time, open and closed rows are considered. Nonsequenced referential constraints are only allowed between a temporal child table and a nontemporal parent, or between a temporal child table and a

Temporal Referential Constraint Form	Description
	temporal parent that does not have the same kind of temporal column as the child table.

Example: Temporal referential constraint

Assume that a CURRENT VALIDTIME referential constraint is defined between the following two valid-time tables.

Col1	Col2 (FK)	VTColA
100	5	('2006/05/20', '2016/05/20')

ColA	ColB (PK)	VTColB
200	5	('2006/07/20', '9999/12/31')

As is true for any referential constraint, every value in the constrained referencing (FK) column of the child table must exist in the referenced (PK) column of the parent table. However, because these are temporal tables, and the constraint is CURRENT VALIDTIME, the portion of the child row valid-time that begins at current time and extends through future time must be contained in the valid-time of the corresponding parent table row or rows.

Whether the constraint is violated depends on the current time when the row is inserted in the child table:

- If TEMPORAL_DATE is 2006/11/20 at the time of the insertion, the constraint is not violated.

The parent row valid-time period contains the portion of the child row valid-time period starting from current time.

Although the valid-time of the parent row does not include the portion of the child row valid-time from 2006/05/20 through 2006/07/20), this is history, and the CURRENT VALIDTIME relational constraint does not consider history.

The value in the parent table can exist in more than one row, provided that the valid-time periods of all such rows combine to contain the current and future portions of the child row valid time. The CURRENT VALIDTIME relational constraints would not be violated if the parent table included the following rows.

ColA	ColB (PK)	VTColB
150	5	('2006/07/20', '2009/07/20')
250	8	('2004/07/20', '2005/07/20')
350	5	('2009/07/20', '2017/07/20')

- If TEMPORAL_DATE is 2006/06/20 at the time of the insertion, the constraint is violated.

The corresponding parent rows do not include the current portion of the child row valid time from 2006/06/20 to 2006/07/20.

CURRENT and SEQUENCED referential constraints can be defined only between tables having the same types of time, valid time or transaction time. NONSEQUENCED referential constraints can be defined between a child table having the type of time specified in the constraint (VALIDTIME or TRANSACTION TIME) and a parent table that lacks the corresponding time dimension.

The following table summarizes the kinds of referential constraints that can be created between different parent and child table types.

Child Table Type	Parent Table Type			
	Non-temporal (USI)	Valid Time	Transaction Time	Bitemporal
Non-temporal	R	TRC with open parent rows	R	TRC
Valid Time	NVT	CVT SVT	Invalid	CVT SVT
Transaction Time	NTT	NTT TRC	CTT STT	CTT TRC STT TRC
Bitemporal	NTT NVT	NTT CVT SVT	CTT STT NVT	CTT STT CVT SVT

Types of referential constraints represented in the table:

R=regular, nontemporal referential constraint

CVT=Current Valid Time

CTT=Current Transaction Time

SVT=Sequenced Valid Time

STT=Sequenced Transaction Time

NVT=Nonsequenced Valid Time

NTT=Nonsequenced Transaction Time

TRC=Temporal Relationship Constraint (see [Temporal Relationship Constraints](#))

Although temporal relational constraints are not enforced by Vantage, the following table describes the relationship that is assumed to exist between the type of temporal relational constraint on the child FK and the type of uniqueness constraint on the referenced parent columns.

Temporal Qualifier on Child Table Relational Constraint	Assumed Temporal Qualifier on Parent Table PK/ UNIQUE Constraint
CURRENT TRANSACTIONTIME	CURRENT TRANSACTIONTIME or SEQUENCED TRANSACTIONTIME
SEQUENCED TRANSACTIONTIME	SEQUENCED TRANSACTIONTIME
NONSEQUENCED TRANSACTIONTIME	Parent table cannot have a transaction-time column
CURRENT VALIDTIME	CURRENT VALIDTIME or SEQUENCED VALIDTIME
SEQUENCED VALIDTIME	SEQUENCED VALIDTIME
NONSEQUENCED VALIDTIME	Parent table cannot have a valid-time column

Examples: Sequenced validtime foreign key

The following example demonstrates a SEQUENCED VALIDTIME foreign key. The PrjAsgnmentDuration column holds the duration for which an employee is assigned to a project.

```
CREATE MULTISET TABLE employee
(
  empid INTEGER,
  address VARCHAR(200),
  jobduration PERIOD(DATE) AS VALIDTIME
)PRIMARY INDEX ( empid );

CREATE MULTISET TABLE project
(
  prjid INTEGER,
  empid INTEGER,
  PrjAsgnmentDuration PERIOD(DATE) AS VALIDTIME,
  SEQUENCED VALIDTIME
  FOREIGN KEY(empid) REFERENCES
    WITH NO CHECK OPTION employee (empid)
)PRIMARY INDEX (prjid );
```

The following example demonstrates a SEQUENCED VALIDTIME AND CURRENT TRANSACTIONTIME foreign key.

```
CREATE MULTISET TABLE employee
(
  empid INTEGER,
  address VARCHAR(200),
```

```

jobduration PERIOD(DATE) AS VALIDTIME,
tt PERIOD(TIMESTAMP(6) WITH TIME ZONE ) AS TRANSACTIONTIME NOT NULL
)
PRIMARY INDEX ( empid );

CREATE MULTISET TABLE project
(
prjid INTEGER,
empid INTEGER,
PrjAsgnmentDuration PERIOD(DATE) AS VALIDTIME,
tt PERIOD(TIMESTAMP(6) WITH TIME ZONE ) AS TRANSACTIONTIME NOT NULL,
SEQUENCED VALIDTIME AND CURRENT TRANSACTIONTIME
    FOREIGN KEY(empid) REFERENCES
        WITH NO CHECK OPTION employee (empid)
)
PRIMARY INDEX (prjid );

```

The following example demonstrates altering an existing temporal table to have a SEQUENCED VALIDTIME AND CURRENT TRANSACTIONTIME foreign key.

```

CREATE MULTISET TABLE employee
(
empid INTEGER,
address VARCHAR(200),
jobduration PERIOD(DATE) AS VALIDTIME,
tt PERIOD(TIMESTAMP(6) WITH TIME ZONE ) AS TRANSACTIONTIME NOT NULL
)
PRIMARY INDEX ( empid );

CREATE MULTISET TABLE project
(
prjid INTEGER,
empid INTEGER,
PrjAsgnmentDuration PERIOD(DATE) AS VALIDTIME,
tt PERIOD(TIMESTAMP(6) WITH TIME ZONE ) AS TRANSACTIONTIME NOT NULL
)
PRIMARY INDEX (prjid );

ALTER TABLE project add SEQUENCED VALIDTIME AND CURRENT TRANSACTIONTIME
    FOREIGN KEY(empid) REFERENCES
        WITH NO CHECK OPTION employee (empid) ;

```

Temporal Relationship Constraints

A Temporal Relationship Constraint (TRC) is a referential relationship that is defined between a child table that does not have a valid-time column and a parent table that has a valid-time column. The FK of the child table must include a column, the TRC column, that references the valid-time column in the PK of the parent table. The value in the TRC column of the child table is constrained because it must exist within the time period defined by the valid-time column of the corresponding row of the valid-time table.

No special temporal syntax or qualifiers is required to create a TRC. Use the standard REFERENCES WITH NO CHECK OPTION syntax that is also used for creating other types of soft referential constraints. The difference is that for TRC the child table cannot have a valid-time column, and the parent table must have a valid-time column.

Because the parent table is a temporal table with valid-time, the value of the child table FK (excluding the TRC column value) can exist in more than one row of the parent table. In this case, the corresponding parent table rows must have non-overlapping, contiguous valid-time periods.

Like other temporal referential constraints, TRC is a soft constraint that is not enforced by the database. The primary reason to define TRC is to improve performance by allowing the Optimizer to eliminate redundant joins.

Examples of Temporal Relationship Constraints

The following statement creates a table and constrains the sale_date column value of each row to be a TIMESTAMP value that lies within the period defined by the valid-time column (vtcol) of the corresponding row in the parent valid-time table.

```
CREATE MULTISET TABLE sales (
  id INTEGER,
  description VARCHAR (100),
  sale_date TIMESTAMP(6),
  FOREIGN KEY (id, sale_date)
    REFERENCES WITH NO CHECK OPTION product(prod_id, vtcol)
) PRIMARY INDEX(id);
```

More than one TRC can be defined for a child table, but only one column can be the TRC column. In the case of the following example, this is the sale_date column of the child table:

```
CREATE MULTISET TABLE sales (
  id INTEGER,
  id2 INTEGER,
  description VARCHAR(100),
  sale_date TIMESTAMP(6),
  FOREIGN KEY (id, sale_date) REFERENCES WITH NO CHECK OPTION
    product(prod_id, vtcol),
```

```
FOREIGN_KEY (id2, sale_date) REFERENCES WITH NO CHECK OPTION
product(prod_id2, vtcol) ,
) PRIMARY INDEX(id);
```

When there are two DateTime columns in the foreign key, the one that corresponds to the parent table valid-time column becomes the TRC column. In the example below column 'c' will be treated as the TRC column:

```
CREATE MULTISET TABLE Parent_Table
(
  a INT,
  b INT,
  c DATE,
  vt PERIOD(DATE) NOT NULL AS VALIDTIME, d DATE
)
PRIMARY INDEX(a);

CREATE MULTISET TABLE Child_Table(
  a INT,
  b INT,
  c DATE,
  d DATE,
  FOREIGN KEY (b, c, d)
    REFERENCES WITH NO CHECK OPTION Parent_Table(b, vt, d)
);
```

Related Information

For more information on...	See...
CREATE TABLE (temporal form)	CREATE TABLE/CREATE TABLE AS (Temporal Forms)
creating temporal tables	Creating Temporal Tables
join elimination	<i>Teradata Vantage™ - SQL Request and Transaction Processing</i> , B035-1142
validating temporal referential constraints	Enforcing and Validating Temporal Referential Constraints

SQL HELP and SHOW Statements

HELP and SHOW statements display information that is relevant to temporal tables:

- HELP COLUMN output includes a Temporal Column field that displays V, T, R, or N, to indicate a valid-time, transaction-time, temporal relationship constraint, or nontemporal column, respectively.

- **HELP COLUMN** also includes fields named **Current ValidTime Unique**, **Sequenced ValidTime Unique**, **NonSequenced ValidTime Unique**, and **Current TransactionTime Unique**, which indicate the type of primary key or unique temporal constraint, if any, that is defined on the column. The values of these fields can be Y or N.
- **HELP CONSTRAINT** has a **Type** field that shows information about named constraints, including temporal constraints.
- **HELP SESSION** includes a **Temporal Qualifier** field that shows information about temporal session attributes.
- **HELP TRIGGER** includes **ValidTime Type** and **TransactionTime Type** fields that display C, S, N, or T to indicate trigger types that are current, sequenced, nonsequenced, or nontemporal, respectively. If there is no time dimension defined on the trigger, these fields are NULL.
- **SHOW TABLE** indicates valid-time and transaction-time columns, and shows temporal primary key and unique constraint definitions.
- **SHOW JOIN INDEX** shows the temporal qualifier associated with a join index created on a temporal table, which is the temporal qualifier associated with the **SELECT** statement that created the join index.

SQL Data Manipulation Language (Temporal Forms)

This section describes the SQL DML statements for temporal tables.

This material covers the syntax, rules, and other details that are specific to temporal table support.

All existing rules that apply to the corresponding non-temporal DML statements also apply to the statements here. For more information about these rules, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Note:

DML statements that do not use the NONTEMPORAL qualifier are applied only to rows that are open in the transaction-time dimension.

Note:

To ensure application portability to ANSI standards for temporal SQL, Teradata recommends explicit specification of all temporal qualifiers.

ABORT (Temporal Form)

Terminates the current transaction and rolls back its updates.

ABORT Syntax (Temporal Form)

Note:

Teradata Vantage™ - Temporal Table Support describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
[ valid_time_qualifier [ AND transaction_time_qualifier ] |
  transaction_time_qualifier [ AND valid_time_qualifier ] |
  AS OF date_timestamp_expression
]
ABORT [ 'message' ] [ FROM option ] [ WHERE abort_condition ]
```

Note:

To ensure application portability to ANSI standards for temporal SQL, Teradata recommends explicit specification of all temporal qualifiers.

valid_time_qualifier

```
{ { CURRENT | NONSEQUENCED } VALIDTIME |
  VALIDTIME AS OF date_timestamp_expression |
  [ SEQUENCED ] VALIDTIME [ period_expression ]
}
```

transaction_time_qualifier

```
{ { CURRENT | NONSEQUENCED } TRANSACTIONTIME |
  TRANSACTIONTIME AS OF date_timestamp_expression
}
```

ABORT Syntax Elements (Temporal Form)

CURRENT VALIDTIME

Specifies that only rows that are currently valid participate in the evaluation of the abort condition.

At least one table referenced in the statement must have valid time.

VALIDTIME AS OF *date_timestamp_expression*

Specifies a given time that must overlap the valid time of a row for that row to participate in the evaluation of the abort condition.

At least one table referenced in the statement must have valid time.

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

SEQUENCED VALIDTIME

Specifies a period of applicability that must overlap the period of validity of a row for that row to participate in the evaluation of the abort condition.

For more information see [Sequenced Valid-Time Queries](#).

period_expression

Specifies the period of applicability for the DML statement.

The period of applicability must be a period constant expression that does not reference any columns, but can reference parameterized values and the `TEMPORAL_DATE` or `TEMPORAL_TIMESTAMP` built-in functions.

The period of applicability can also be a self-contained noncorrelated scalar subquery that is always nonsequenced in the time dimensions regardless of the temporal qualifier for the DML statement.

Note:

If a *period_expression* is specified, the valid-time column cannot be specified or referenced anywhere in the query. If the valid-time column is a derived period column, the component columns cannot be specified or referenced anywhere in the query.

If *period_expression* is omitted, the period of applicability defaults to `PERIOD('0001-01-01, UNTIL_CHANGED')` for a `PERIOD(DATE)` valid-time column or `PERIOD('0001-01-01 00:00:00.000000+00:00, UNTIL_CHANGED')` for a `PERIOD(TIMESTAMP(n) WITH TIME ZONE)` valid-time column, where precision *n* and `WITH TIME ZONE` are optional.

NONSEQUENCED VALIDTIME

Specifies that rows that participate in the evaluation of the abort condition are not further evaluated for qualification in the valid-time dimension.

At least one table referenced in the statement must have valid time.

AND

Specifies a keyword for specifying both a valid-time qualifier and a transaction-time qualifier.

CURRENT TRANSACTIONTIME

Specifies that only rows that are open participate in the evaluation of the abort condition.

At least one table referenced in the statement must have transaction time.

TRANSACTIONTIME AS OF *date_timestamp_expression*

Specifies a given time that must overlap the transaction time of a row for that row to participate in the evaluation of the abort condition.

At least one table referenced in the statement must have transaction time.

NONSEQUENCED TRANSACTIONTIME

Specifies that rows that participate in the evaluation of the abort condition are not further evaluated for qualification in the transaction-time dimension.

At least one table referenced in the statement must have transaction time.

AS OF *date_timestamp_expression*

Specifies that only rows that overlap *date_timestamp_expression* in the valid-time and transaction-time dimension participate in the evaluation of the abort condition.

'message'

Specifies the text of the message to be returned when the transaction is terminated.

FROM *option*

Specifies the temporal tables that are further qualified in the WHERE clause.

WHERE *abort_condition*

Specifies an expression where the result must evaluate to TRUE for Vantage to roll back the transaction.

Usage Notes

Omitting a Valid-Time Qualifier

If no temporal qualifier is specified in the valid-time dimension in the statement, the system uses the temporal qualifier of the session. If none is explicitly specified for the session, the default of CURRENT is assumed and only rows that are currently valid participate in further processing.

Omitting a Transaction-Time Qualifier

If no temporal qualifier is specified in the transaction-time dimension in the statement, the system uses the temporal qualifier of the session. If none is specified for the session, the default of CURRENT is assumed and only rows that are open participate in further processing.

Temporal Qualifier and Subqueries

The temporal qualifier of a statement applies to all subqueries; no separate temporal qualifier is allowed for the subquery.

Related Information

For more information on...	See...
ABORT statement	<i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146
FROM clause syntax and usage	FROM Clause (Temporal Form)

DELETE (Temporal Form)

Deletes or modifies one or more rows from a temporal table.

Required Privileges

The privileges required are the same as those required for a conventional DELETE statement.

If the DELETE statement specifies the NONTEMPORAL qualifier, the NONTEMPORAL privilege is also required on the temporal table.

DELETE Syntax (Temporal Form)

Note:

Teradata Vantage™ - Temporal Table Support describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
[ { CURRENT | NONSEQUENCED } VALIDTIME |
  { [ SEQUENCED ] VALIDTIME } [ period_expression ] |
  NONTEMPORAL
]
DELETE FROM table_name
  [ [AS] correlation_name [, joined_table_name ] [...] ]
  [ WHERE condition ] [;]
```

Note:

To ensure application portability to ANSI standards for temporal SQL, Teradata recommends explicit specification of all temporal qualifiers.

DELETE Syntax Elements (Temporal Form)

CURRENT VALIDTIME

Specifies that the delete is current in the valid-time dimension if the target table supports valid time.

Note:

A current DELETE affects only current rows. Future rows with valid-time periods that do not overlap TEMPORAL_TIMESTAMP or TEMPORAL_DATE will not be deleted.

If the target table does not support valid time, at least one of the referenced tables must be a table with valid time. The delete is not a current delete. The CURRENT VALIDTIME qualifier is used to qualify rows from the referenced tables in the valid-time dimension. In the transaction-time dimension, open rows qualify.

Note:

CURRENT DML modifications can cause serializability issues for concurrent transactions. See [Potential Concurrency Issues with Current Temporal DML](#) for information on avoiding these issues.

VALIDTIME and SEQUENCED VALIDTIME

Specifies that the delete is sequenced in the valid-time dimension if the target table supports valid time.

If the target table does not support valid time, at least one of the referenced tables must be a table with valid time. The delete is not a sequenced delete. The VALIDTIME or SEQUENCED VALIDTIME qualifier is used to qualify rows from the referenced tables in the valid-time dimension. In the transaction-time dimension, open rows qualify.

period_expression

Specifies the period of applicability for the DML statement.

The period of applicability must be a period constant expression that does not reference any columns, but can reference parameterized values and the TEMPORAL_DATE or TEMPORAL_TIMESTAMP built-in functions.

The period of applicability can also be a self-contained noncorrelated scalar subquery that is always nonsequenced in the time dimensions regardless of the temporal qualifier for the DML statement.

Note:

If a *period_expression* is specified, the valid-time column cannot be specified or referenced anywhere in the query. If the valid-time column is a derived period column, the component columns cannot be specified or referenced anywhere in the query.

If *period_expression* is omitted, the period of applicability defaults to PERIOD'(0001-01-01, UNTIL_CHANGED)' for a PERIOD(DATE) valid-time column or PERIOD '(0001-01-01 00:00:00.000000+00:00, UNTIL_CHANGED)' for a PERIOD(TIMESTAMP(*n*) WITH TIME ZONE) valid-time column, where precision *n* and WITH TIME ZONE are optional.

NONSEQUENCED VALIDTIME

Specifies that the delete is nonsequenced in the valid-time dimension if the target table supports valid time.

If the target table does not support valid time, at least one of the referenced tables must be a table with valid time. The delete is not a nonsequenced delete. The NONSEQUENCED VALIDTIME qualifier is used to qualify rows from the referenced tables in the valid-time dimension. In the transaction-time dimension, open rows qualify.

NONTEMPORAL

Specifies that the delete is nonsequenced in the valid-time dimension and nontemporal in the transaction-time dimension.

The target table must support transaction time.

table_name

Specifies the name of the target table from which the delete operation is to remove rows.

[AS] *correlation_name*

Specifies an optional table alias name.

joined_table_name

Specifies the name of a joined table referenced in the WHERE clause.

WHERE *condition*

Specifies a condition for filtering the rows to be deleted.

Usage Notes

Unless the NONTEMPORAL qualifier is specified, deletion on temporal tables with a transaction-time column is always limited to only those rows that are open in transaction time.

Current Delete

NOTICE

CURRENT DML modifications can cause serializability issues for concurrent transactions. See [Potential Concurrency Issues with Current Temporal DML](#) for information on avoiding these issues.

For a table with valid time, current rows qualify for deletion. Any additional search conditions are applied only on these rows. The conditions in the WHERE clause or join ON conditions can be specified on valid-time or transaction-time columns.

Note:

A current DELETE affects only current rows. Future rows with valid-time periods that do not overlap TEMPORAL_TIMESTAMP or TEMPORAL_DATE will not be deleted.

The following table describes the current delete operation of a qualified row.

IF the table is a ...	AND the beginning bound is ...	THEN the ...
valid-time table and the element type of the valid-time column is DATE	equal to TEMPORAL_DATE	qualified row is physically deleted.
	less than TEMPORAL_DATE	period of validity of the qualified row is modified with the ending bound set to TEMPORAL_DATE.
valid-time table and the element type of the valid-time column is TIMESTAMP	equal to TEMPORAL_TIMESTAMP	qualified row is physically deleted.
	less than TEMPORAL_TIMESTAMP	period of validity of the qualified row is modified with the ending bound set to TEMPORAL_TIMESTAMP.
bitemporal table and the element type of the valid-time column is DATE	equal to TEMPORAL_DATE	qualified row is closed out in transaction time; that is, it is logically deleted.
	less than TEMPORAL_DATE	qualified row is closed out in transaction time and a copy of the old row is inserted with the beginning bound of the period of validity set to the same value as the closed out row and the ending bound of the period of validity set to TEMPORAL_DATE.
bitemporal table and the element type of the valid-time column is TIMESTAMP	equal to TEMPORAL_TIMESTAMP	qualified row is closed out in transaction time; that is, it is logically deleted.

IF the table is a ...	AND the beginning bound is ...	THEN the ...
	less than TEMPORAL_TIMESTAMP	qualified row is closed out in transaction time and a copy of the old row is inserted with the beginning bound of the period of validity set to the same value as the closed out row and the ending bound of the period of validity set to TEMPORAL_TIMESTAMP.

Sequenced Delete

For a table with valid time, rows that overlap with the period of applicability qualify for deletion. Any additional search conditions are applied only on these rows. The conditions in the WHERE clause or join ON conditions can be specified on valid-time or transaction-time columns.

Sequenced delete on a table with transaction time operates only on open rows.

The following table describes the sequenced delete operation of a qualified row.

IF the table is a ...	AND the period of applicability ...	THEN ...
valid-time table	contains the period of validity of the qualified row, including the case where they are equal	the qualified row is physically deleted.
	does not contain the period of validity of the qualified row	<p>If a portion of the period of validity exists before the beginning of the period of applicability, a copy of the row is inserted with the beginning bound of its period of validity set to the same value as the qualified row and the ending bound set to the beginning bound of the period of applicability.</p> <p>If a portion of the period of validity exists after the end of the period of applicability, a copy of the row is inserted with the ending bound of its period of validity set to the same value as the qualified row and the beginning bound set to the ending bound of the period of applicability.</p> <p>Note that two rows can be inserted if the preceding conditions are both true.</p>
bitemporal table	contains the period of validity of the qualified row, including the case where they are equal	the qualified row is closed out (logically deleted) in transaction time.
	does not contain the period of validity of the qualified row	<p>the qualified row is closed out in transaction time.</p> <p>If a portion of the period of validity exists before the beginning of the period of applicability, a copy of the row is inserted with the beginning bound of its period of validity set to the same value as the qualified row and the ending bound set to the beginning bound of the period of applicability.</p>

IF the table is a ...	AND the period of applicability ...	THEN ...
		<p>If a portion of the period of validity exists after the end of the period of applicability, a copy of the row is inserted with the ending bound of its period of validity set to the same value as the qualified row and the beginning bound set to the ending bound of the period of applicability.</p> <p>Note that two rows can be inserted if the preceding conditions are both true.</p>

Nonsequenced Delete

A nonsequenced DELETE statement deletes the specified rows across all states. This ignores valid-time semantics when deleting rows from a table with valid time.

A nonsequenced delete operates on open rows only. A nonsequenced delete treats a valid-time column like a regular column.

For a bitemporal table, a nonsequenced delete of a row closes out the existing qualified row. For a valid-time table, the nonsequenced delete is like a conventional delete statement that physically deletes the qualified row.

Nontemporal Delete

Note:

Rows that are closed in transaction time provide a history of all modifications and deletions on tables that have a transaction-time column. The automatic history that tables with transaction time provide can be used for regulatory compliance auditing, so these rows are generally inaccessible to DML modifications. Because NONTEMPORAL DML statements can modify closed rows, the special NONTEMPORAL privilege is required. For more information on the NONTEMPORAL privilege, see [Usage Notes](#).

A nontemporal delete, also referred to as vacuuming the table, physically deletes the qualifying rows from the table.

NOTICE

The best practice is to back up the data before performing a nontemporal delete. Tracking of any deleted rows will be lost.

All rows, both open and closed, are considered for qualification conditions specified in the DELETE statement. If multiple transaction-time tables are referenced, they are joined using nonsequenced select semantics.

If the table being deleted has valid time, both valid and rows that are no longer valid are considered for qualification conditions specified in the query.

The only difference between a nonsequenced delete and a nontemporal delete on a table with transaction time is that a nonsequenced delete performs a logical delete of rows whereas a nontemporal delete performs a physical delete of rows.

Related Information

For more information on...	See...
DELETE statement	<i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146.
deleting rows from temporal tables	Modifying Temporal Tables .

INSERT/INSERT SELECT (Temporal Forms)

Adds a new row to a named temporal table by directly specifying the row data to be inserted (valued form) or by retrieving the new row data from another table (selected, or INSERT SELECT form).

Required Privileges

The privileges required are the same as those required for a conventional INSERT statement.

If the INSERT statement specifies the NONTEMPORAL qualifier, the NONTEMPORAL privilege is required on the temporal table.

INSERT/INSERT SELECT Syntax (Temporal Forms)

Note:

Teradata Vantage™ - Temporal Table Support describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
[ { CURRENT | NONSEQUENCED } VALIDTIME |
  { [ SEQUENCED ] VALIDTIME } [ period_expression ] |
  NONTEMPORAL
]
INSERT INTO table_name
  { [ ( column_name [,...] ) ] { VALUES ( values_spec ) | subquery } |
```

```

    DEFAULT VALUES
  } [;]

```

Note:

To ensure application portability to ANSI standards for temporal SQL, Teradata recommends explicit specification of all temporal qualifiers.

INSERT/INSERT SELECT Syntax Elements (Temporal Forms)

values_spec

```
{ expression [,...] | column_name = expression [,...] }
```

CURRENT VALIDTIME

Specifies that the insert is current in valid time.

At least one of the referenced tables or the target table must be a table with a valid-time column. The CURRENT VALIDTIME qualifier is used to qualify rows from the referenced tables in the valid-time dimension. In the transaction-time dimension, open rows qualify.

VALIDTIME and SEQUENCED VALIDTIME

Specifies that the insert is sequenced in valid time.

At least one of the referenced tables or the target table must be a table with a valid-time column. The VALIDTIME or SEQUENCED VALIDTIME qualifier is used to qualify rows from the referenced tables in the valid-time dimension. In the transaction-time dimension, open rows qualify.

Use a sequenced valid-time insert to insert history, current, or future rows.

period_expression

Specifies the period of applicability for the DML statement.

The period of applicability must be a period constant expression that does not reference any columns, but can reference parameterized values and the TEMPORAL_DATE or TEMPORAL_TIMESTAMP built-in functions.

The period of applicability can also be a self-contained noncorrelated scalar subquery that is always nonsequenced in the time dimensions regardless of the temporal qualifier for the DML statement.

Note:

If a *period_expression* is specified, the valid-time column cannot be specified or referenced anywhere in the query. If the valid-time column is a derived period column, the component columns cannot be specified or referenced anywhere in the query.

If *period_expression* is omitted, the period of applicability defaults to PERIOD'(0001-01-01, UNTIL_CHANGED)' for a PERIOD(DATE) valid-time column or PERIOD '(0001-01-01 00:00:00.000000+00:00, UNTIL_CHANGED)' for a PERIOD(TIMESTAMP(*n*) WITH TIME ZONE) valid-time column, where precision *n* and WITH TIME ZONE are optional.

The period of applicability can be specified only for an INSERT SELECT statement, where the statement specifies a subquery.

NONSEQUENCED VALIDTIME

Specifies that the insert is nonsequenced in valid time.

At least one of the referenced tables or the target table must be a table with a valid-time column.

Use a nonsequenced valid-time insert to insert history, current, or future rows.

NONTEMPORAL

Specifies that the insert is a nontemporal insert on a table with transaction time.

table_name

Specifies the name of a temporal table.

***column_name* [... , *column_name*]**

Specifies a named list of columns in the target table.

***expression* [... , *expression*]**

Specifies a positional assignment list of values to insert into the target table.

***column_name* = *expression* [... , *column_name* = *expression*]**

Specifies an assignment list of column names in the target table and the values to assign them.

DEFAULT VALUES

Specifies that a row consisting of default values is to be added to *table_name*.

Usage Notes

All check, primary key, and temporal unique (current, sequenced, nonsequenced) constraints defined on the table are checked only on rows that are open in transaction time.

DML operations on tables defined with NORMALIZE produce a normalized set of modified rows. Some unmodified rows may be deleted from the target table as a result of the normalization.

When the target table is a normalized temporal table with transaction time, rows that are deleted as a result of the normalization are closed in transaction time.

General Rules

The following general rules apply to any variant of INSERT into a temporal table. Rules for specific types of inserts appear in succeeding sections.

- If a valid-time qualifier is neither specified for the session nor specified in the INSERT statement, the statement is a current INSERT statement.
- The value assigned to validtime-column cannot use CURRENT_DATE or CURRENT_TIMESTAMP.
- If the AS OF temporal qualifier is set as the session temporal qualifier, and an INSERT with no temporal qualifier is issued on a temporal table, the session temporal attribute is ignored and the INSERT defaults to current.
- If the target table is a table without a valid-time column, and multiple other tables that support valid time are specified in the INSERT SELECT statement, the qualifier for the valid-time dimension defaults to current (unless the NONTEMPORAL qualifier was specified). In the transaction-time dimension, only open rows qualify.
- No qualifier can be specified for the transaction-time dimension. The session attribute is not applied to the transaction-time dimension. (The NONTEMPORAL qualifier can be used to specify a transaction time if the user has the NONTEMPORAL privilege. For more information see [Usage Notes](#).)

Current Inserts

When an INSERT or INSERT SELECT statement is issued on a temporal table, and the statement either specifies the CURRENT VALIDTIME qualifier or does not specify any temporal qualifier, the insert is a current insert.

Note:

You should always explicitly specify the value of the valid-time column when performing a current insert on a table that has a valid-time column. The valid-time period must overlap TEMPORAL_TIMESTAMP or TEMPORAL_DATE, depending on the type of the valid-time column.

The following information applies to current inserts:

- Do not use a positional assignment list to specify column values for a current insert.

If an INSERT statement uses a positional assignment list, the list can not include a value for the valid-time column. The valid-time column position is skipped when mapping values to the columns, and the column is always set to the system-defined default value.

The system-defined default values for a valid-time column are (TEMPORAL_TIMESTAMP to UNTIL_CHANGED) and (TEMPORAL_DATE to UNTIL_CHANGED), depending on the period data type of the column.

If an INSERT SELECT statement uses a positional assignment list, the values of the valid-time and transaction-time columns from the source table are not copied into the target table, but are replaced by the system-defined default value. This is true even if the statement specifies SELECT * in the SELECT statement or projects the valid-time and transaction-time columns explicitly.

- If an INSERT statement uses a named list or assignment list, the list can specify a value for the valid-time column. If the named list or assignment list does not specify the valid-time column, the valid-time value is set to the system-defined default value.

If an INSERT SELECT statement uses a named list, a valid-time column value or any Period column value can be copied into the target table if the named list specifies a valid-time column. If the SELECT * asterisk notation is used in combination with explicitly specification of a temporal column, use the table_name.* notation. For more information see [Asterisks in Select Lists](#).

Note:

If a CURRENT VALIDTIME INSERT specifies a value for the valid-time column, the period must overlap the current time. To insert valid-time history or future rows into a table, use the SEQUENCED VALIDTIME qualifier to the INSERT statement.

- Values can never be specified for a transaction-time column, unless the NONTEMPORAL qualifier is used. This column is automatically maintained by the system. For current inserts, the system defined default value for the transaction-time column is (TT_TIMESTAMP to UNTIL_CLOSED), where TT_TIMESTAMP is the timestamp value read from the system clock by each AMP during timestamping. The transaction-time column position is always skipped when mapping the inserted values to columns.
- The SELECT statement of an INSERT SELECT is executed as a current SELECT. The result rows of a current SELECT are inserted into the target table with period of validity set to (TEMPORAL_DATE to UNTIL_CHANGED) or (TEMPORAL_TIMESTAMP to UNTIL_CHANGED).

Sequenced Inserts

A sequenced insert is similar to a current insert, but allows the user to specify the valid-time of the inserted rows. Inserted rows can be history, current, or future rows.

The INSERT statement cannot specify explicit values for the transaction-time column if the table has transaction time. The system maintains the transaction time. The transaction-time column name cannot

be specified in the named list or assignment list of a sequenced INSERT statement. The system skips the transaction-time column position when mapping the values to the columns.

The period value specified for the valid-time column must be assignable to the valid-time column of the target table, and cannot be NULL.

For a target table with valid time where the INSERT statement does not specify a SELECT:

- If a positional assignment list is specified, the valid-time value is read from the value list in the same position as that of the valid-time column position. If a value is specified, the value is assigned to the valid-time column. If no value is specified for the valid-time column in the positional assignment list, its value is set to the default value of the valid-time column.
- The valid-time column name can be specified in a named list or assignment list to specify a value. If the named list or assignment list does not specify a valid-time column, the valid-time value is set to its default value.

Note:

If a SEQUENCED VALIDTIME INSERT specifies a value for the valid-time column, the period must not be NULL. To insert a row that has NULL in the valid-time column, use a NONSEQUENCED VALIDTIME INSERT statement.

For a target table with valid time and an INSERT SELECT statement:

- The SELECT subquery must reference a table that has a valid-time column.
- Do not specify the valid-time column or value in the select list. If the target is a bitemporal table, do not specify the valid-time or transaction-time column names or values in the select list. The values from the SELECT list are positionally assigned to the corresponding columns in the target table as if valid-time and transaction-time columns do not exist in the target.
- The SELECT statement is executed as a sequenced SELECT if at least one table is a table with valid time. The result rows of a SELECT are inserted into the target table with their valid time period set to the overlap (P_INTERSECT) of the source row period of validity with the SELECT statement period of applicability. The precision of the resulting overlapped valid-time value must be assignable to the valid-time period of the target table.
- If a valid-time column in the target table is a derived period column, the beginning and end bounds of the inserted rows are assigned to the component columns of the derived period valid-time column in the target table.

Nonsequenced Inserts

For a table with transaction time, the INSERT statement cannot specify a value for the transaction-time column. The system maintains the transaction time. If the INSERT statement uses a positional assignment list, the transaction-time column position is skipped when mapping the values to the columns.

The Period value specified for the valid-time column must be assignable to the valid-time column of the target table; otherwise, an appropriate Period data type error is reported.

The SELECT statement of a nonsequenced INSERT SELECT is executed as a nonsequenced SELECT. The result rows of the SELECT are inserted into the target table with the period of validity set to the corresponding Period value in the selected list.

If the valid-time column is not specified in the positional assignment list or in the named list and no default value is specified for the valid-time column, it defaults to NULL. A NULL in a valid-time column can be modified only by using a nonsequenced update because a current or sequenced update can never qualify this row.

Nontemporal Inserts

Note:

Rows that are closed in transaction time provide a history of all modifications and deletions on tables that have a transaction-time column. The automatic history that tables with transaction time provide can be used for regulatory compliance auditing, so these rows are generally inaccessible to DML modifications. Because NONTEMPORAL DML statements can modify closed rows, the special NONTEMPORAL privilege is required. For more information on the NONTEMPORAL privilege, see [Usage Notes](#).

The positional assignment list or named list of the INSERT statement can specify the valid-time and transaction-time column values. The values must be assignable to the appropriate columns or the system reports an error.

The following rules apply to the transaction-time column value in an INSERT statement:

- The beginning bound must not be greater than the value read from the system clock during the insert.
- The ending bound must be UNTIL_CLOSED or must be less than or equal to the value read from the system clock during the insert.

The SELECT statement of a nontemporal INSERT SELECT is executed as a nonsequenced SELECT in both the valid-time and transaction-time dimensions. The result rows of the SELECT are inserted into the target table.

INSERT SELECT and Error Logging Tables

You can create an error logging table that you associate with a temporal table when you want Vantage to log information about insert errors that occur during an INSERT SELECT operation on the temporal table.

To create an error logging table, use the CREATE ERROR TABLE statement. Error logging tables are nontemporal tables. When you create an error logging table that you associate with a temporal table, the temporal columns in the temporal table are included in the error logging table as regular period data type columns.

If the temporal table uses derived period columns for the temporal columns, the corresponding columns in the error table will also be derived period columns.

To enable error logging for an INSERT SELECT statement, specify the LOGGING ERRORS option. The behavior of the error logging facilities for INSERT SELECT errors is the same for nontemporal and temporal tables.

For details on how to create an error logging table, see the CREATE ERROR TABLE statement in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. For details on how to specify error handling for the INSERT SELECT statement, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Related Information

For more information on...	See...
INSERT statement	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144
inserting rows into temporal tables	Modifying Temporal Tables

MERGE (Temporal Form)

Performs a temporal merge of a source row set into a target table based on the temporal qualifier and whether any target rows satisfy a specified matching condition with the source row.

Required Privileges

The temporal form of MERGE requires the same privileges as the conventional form of MERGE.

MERGE Syntax (Temporal Form)

Note:

Teradata Vantage™ - Temporal Table Support describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144, *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146, and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
[ { CURRENT | NONSEQUENCED } VALIDTIME |
  [ SEQUENCED ] VALIDTIME [ period_expression ]
]
merge_statement [ ; ]
```

Note:

To ensure application portability to ANSI standards for temporal SQL, Teradata recommends explicit specification of all temporal qualifiers.

MERGE Syntax Elements (Temporal Form)

CURRENT VALIDTIME

Specifies that the merge is current in the valid-time dimension.

Either the target table or the source table must support the valid-time dimension.

NOTICE

CURRENT DML modifications can cause serializability issues for concurrent transactions. See [Potential Concurrency Issues with Current Temporal DML](#) for information on avoiding these issues.

VALIDTIME and SEQUENCED VALIDTIME

Specifies that the merge is sequenced in the valid-time dimension.

Either the target table or the source table must support the valid-time dimension.

period_expression

Specifies the period of applicability for the DML statement.

The period of applicability must be a period constant expression that does not reference any columns, but can reference parameterized values and the TEMPORAL_DATE or TEMPORAL_TIMESTAMP built-in functions.

The period of applicability can also be a self-contained noncorrelated scalar subquery that is always nonsequenced in the time dimensions regardless of the temporal qualifier for the DML statement.

Note:

If a *period_expression* is specified, the valid-time column cannot be specified or referenced anywhere in the query. If the valid-time column is a derived period column, the component columns cannot be specified or referenced anywhere in the query.

If *period_expression* is omitted, the period of applicability defaults to PERIOD'(0001-01-01, UNTIL_CHANGED)' for a PERIOD(DATE) valid-time column or PERIOD '(0001-01-01 00:00:00.000000+00:00, UNTIL_CHANGED)' for a PERIOD(TIMESTAMP(*n*) WITH TIME ZONE) valid-time column, where precision *n* and WITH TIME ZONE are optional.

NONSEQUENCED VALIDTIME

Specifies that the merge is nonsequenced in the valid-time dimension.

Either the target table or the source table must support the valid-time dimension.

merge_statement

Specifies the syntax for the conventional form of the MERGE statement.

For details, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Usage Notes

General

For a table that supports transaction time, the temporal qualifier in the transaction-time dimension is CURRENT TRANSACTIONTIME.

All check, primary key, and temporal unique (current, sequenced, nonsequenced) constraints defined on the table are checked only on rows that are open in transaction time.

If either the target table or source table is a temporal table and the MERGE statement does not specify a temporal qualifier, the value of the session valid-time qualifier is used for the temporal table.

If the source row set is specified by a value list, the source is considered nontemporal.

A select subquery or a table specified as a source row set in the merge statement can reference a temporal table. The derived table can result in a temporal or nontemporal table, depending on the temporal qualifier used.

MERGE is not supported on tables that do not have primary indexes and column-partitioned tables.

The UPDATE portion of the MERGE statement follows the rules of the temporal UPDATE statement semantics and the INSERT portion of the MERGE statement follows the rules of the temporal INSERT statement semantics.

DML operations on tables defined with NORMALIZE produce a normalized set of modified rows. Some unmodified rows may be deleted from the target table as a result of the normalization.

When the target table is a normalized temporal table with transaction time, rows that are deleted as a result of the normalization are closed in transaction time.

Error Logging Tables

You can create an error logging table that you associate with a temporal table when you want Vantage to log information about update and insert errors that occur during a MERGE operation on the temporal table.

To create an error logging table, use the CREATE ERROR TABLE statement. Error logging tables are nontemporal tables. When you create an error logging table that you associate with a temporal table, the temporal columns in the temporal table are included in the error logging table as regular period data type columns.

If a valid-time column in the target table is a derived period column, the beginning and end bounds of the inserted rows are assigned to the component columns of the derived period valid-time column in the target table.

To enable error logging for a MERGE statement, specify the LOGGING ERRORS option.

For the temporal form of MERGE, the Vantage error logging facilities consider the following errors as local errors:

- Sequenced duplicate rows
- Check constraint violations
- Errors during row build such as division by zero

Nonlocal errors are complex updates and nonsequenced USI updates. Unique constraint violations on a temporal table are join index errors instead of nonlocal errors as on a nontemporal table.

For details on how to create an error logging table, see the CREATE ERROR TABLE statement in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. For details on how to specify error handling for the INSERT SELECT statement, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Modification of Rows

The following table describes the modification of rows for a temporal merge:

Temporal Merge Type	Modification Details
Current	<p>A current merge results in a current update if matching rows are found; otherwise, it results in a current insert.</p> <p>The SET clause of the UPDATE portion cannot reference the valid-time column or the transaction-time column as the name of a column to update.</p> <p>The system may modify the temporal columns during an update of the matched row. For a row-partitioned table, if the modification of the temporal columns causes the rows to change partitions within the same AMP, the plan is sub-optimal since the row must be deleted and a new row inserted. The insert requires a spool.</p> <p>The specification of a valid-time value is allowed in the INSERT portion in the merge when the INSERT uses a named list or assignment list. This inserted row can be into a different partition from the matched partition within the same AMP. This can be achieved if the row to be inserted is spooled in the same way as done when the matched row changes partition.</p>
Sequenced	<p>If the source and target rows satisfy the matching condition for a sequenced merge, a sequenced update is performed.</p>

Temporal Merge Type	Modification Details
	<ul style="list-style-type: none"> When both tables are temporal, the valid-time portion updated is the intersection of period of applicability, the period of validity of the target table row, and the period of validity of the source row. The intersection of the period of validity of the source row and the period of applicability must be contained in the period of validity of the target row. When only the target table is temporal, the valid-time portion updated is the intersection of the period of applicability and the period of validity of the target table row. The SET clause of the UPDATE portion cannot reference the valid-time column or the transaction-time column as the name of a column to update. <p>If the source and target rows do not satisfy the matching condition, a sequenced insert is performed. The period of applicability, if specified, is ignored for the insert. The insert specification follows the rules for a simple sequenced insert and INSERT can specify a valid-time column value. The inserted row with the specified valid-time column value can be into a different partition from the matched partition within the same AMP.</p>
Nonsequenced	<p>If matching rows are found for a nonsequenced merge, a nonsequenced update is performed; otherwise, a nonsequenced insert is performed.</p> <p>The SET clause of the UPDATE portion can reference the valid-time column as the name of a column to update.</p> <p>For a row-partitioned table where partitioning is on the valid-time column, the valid-time column can be modified.</p> <p>The insert specification follows the rules of nonsequenced simple insert and the valid-time column value can be specified. The inserted row with the specified valid-time column value can be into a different partition from the matched partition within the same AMP.</p>

Matching Process

The following table describes the merge matching process when the target table is a temporal table.

Temporal Merge Type	Matching Process
Current	<p>The matching condition is applied on all current rows in the target table. For a target table with transaction time, all conditions and modifications are applied only on open rows.</p> <p>If the source table is temporal, only the current rows from the source table participate in the merge process. For a source table that has transaction time, only open rows participate in the merge process. To get the source row set, the system rewrites the source to be a derived table.</p>
Sequenced	<p>The matching condition is applied on all target rows that overlap the optional period of applicability. For a target table with transaction time, all conditions and modifications are applied only on open rows.</p> <p>If the source table is temporal, only those source rows that overlap the optional period of applicability participate in the merge process. For a source table that has transaction</p>

Temporal Merge Type	Matching Process
	<p>time, only open rows participate in the merge process. To get the source row set, the system rewrites the source to be a derived table.</p> <p>When both the source and target tables are temporal, sequenced join semantics are applied, meaning that the matching condition of the ON clause additionally includes the system-added overlap condition on the temporal columns of the source and target.</p>
Nonsequenced	<p>All of the existing restrictions that apply to the matching condition of the ON clause for the conventional form of MERGE apply in the valid-time dimension in a nonsequenced merge.</p> <p>For a target table with transaction time, all conditions and modifications are applied only on open rows. For a source table that has transaction time, only open rows participate in the merge process. To get the source row set, the system rewrites the source to be a derived table.</p>

Merging into a Row-Partitioned Valid-Time Table

For all MERGE statements, an ON clause match condition must specify an equality constraint on the primary index of the target table. (Note that MERGE is not supported on tables without primary indexes and column-partitioned tables.) To qualify for the ANSI MERGE path, which provides enhanced optimization, if the target table is row partitioned, the equality must also include the partitioning column to qualify a single partition. For information about the ANSI MERGE optimizer path, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Because the recommended partitioning expressions for temporal tables use only the END bound of the temporal column time periods, the match condition can similarly use the END condition in the equality constraint. END (*valid_time_column*) IS UNTIL_CHANGED and END (*transaction_time_column*) IS UNTIL_CLOSED can be used as equality constraints on temporal columns for temporal tables that use the recommended partitioning expressions (see [Partitioning Expressions for Temporal Tables](#)).

The ending bound of the valid-time of the target rows is seldom known in advance. A solution is to pre-join the source and target tables, using the same conditions in the USING clause, to determine the valid-time values in the target table.

The pre-join should use the NONSEQUENCED VALIDTIME qualifier (AND CURRENT TRANSACTIONTIME, if applicable). The pre-join must be a left outer join from the source in order to preserve the non-matching row set for insertion into the target table.

Example: Merging into a Row-Partitioned Valid-time Table

The following example uses a nontemporal source table for the merge. Note that the values to be inserted can be any values, even those that would go into a different partition.

```
CREATE MULTISET TABLE bi_tgt_tbl
(
```

```

    pkey_field INTEGER,
    int2_field INTEGER,
    vtc1 PERIOD(DATE) NOT NULL AS VALIDTIME,
    ttcol PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
        AS TRANSACTIONTIME)
PRIMARY INDEX ( pkey_field )
PARTITION BY CASE_N(
((END(vtc1)) >= DATE ) AND ((END(ttcol)) >= CURRENT_TIMESTAMP(6)),
((END(vtc1)) < DATE ) AND ((END(ttcol)) >= CURRENT_TIMESTAMP(6)),
(END(ttcol )) < CURRENT_TIMESTAMP(6));

CREATE SET TABLE src_tbl
(
    pkey_field INTEGER,
    int2_field INTEGER)
PRIMARY INDEX ( pkey_field );

SEQUENCED VALIDTIME
MERGE INTO bi_Tgt_tbl
USING /* This block prejoins and determines the target valid time
      values */
(
    NONSEQUENCED VALIDTIME PERIOD (DATE'2009-12-15', DATE'2009-12-18')
    AND CURRENT TRANSACTIONTIME
    SELECT s.pkey_field, s.int2_field, END(b_t.vtc1) vtend,
           END(b_t.ttcol) ttend
    FROM src_tbl s LEFT OUTER JOIN bi_tgt_tbl b_t
    ON s.pkey_field = b_t.pkey_field
) AS nonbi_srct ( pkey, int2, vtend, ttend)
ON (pkey_field = nonbi_srct.pkey)
AND END(vtc1) = vtend
AND END(ttcol) = ttend

WHEN MATCHED THEN
    UPDATE SET int2_field = nonbi_srct.int2
WHEN NOT MATCHED THEN
    INSERT ( nonbi_srct.pkey, nonbi_srct.int2,
            PERIOD(TEMPORAL_DATE, UNTIL_CHANGED )
    );

```

Related Information

For more information on...	See...
MERGE statement	<i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146
INSERT (temporal form)	INSERT/INSERT SELECT (Temporal Forms)
row partitioning temporal tables	Partitioning Expressions for Temporal Tables
UPDATE (temporal form)	UPDATE (Temporal Form)

ROLLBACK (Temporal Form)

Terminates and rolls back the current transaction.

ROLLBACK Syntax (Temporal Form)

Note:

Teradata Vantage™ - Temporal Table Support describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
[ valid_time_qualifier [ AND transaction_time_qualifier ] |
  transaction_time_qualifier [ AND valid_time_qualifier ] |
  AS OF date_timestamp_expression
]
ROLLBACK [ WORK ] [ 'message' ] [ FROM option ] [ WHERE abort_condition ]
```

Note:

To ensure application portability to ANSI standards for temporal SQL, Teradata recommends explicit specification of all temporal qualifiers.

ROLLBACK Syntax Elements (Temporal Form)

valid_time_qualifier

```
{ { CURRENT | NONSEQUENCED } VALIDTIME |
  VALIDTIME AS OF date_timestamp_expression |
```

```
[ SEQUENCED ] VALIDTIME [ period_expression ]
}
```

transaction_time_qualifier

```
{ { CURRENT | NONSEQUENCED } TRANSACTIONTIME |
  TRANSACTIONTIME AS OF date_timestamp_expression
}
```

CURRENT VALIDTIME

Specifies that only rows that are currently valid participate in the evaluation of the abort condition.

At least one table referenced in the statement must have valid time.

VALIDTIME AS OF *date_timestamp_expression*

Specifies a given time that must overlap the valid time of a row for that row to participate in the evaluation of the abort condition.

At least one table referenced in the statement must have valid time.

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

SEQUENCED VALIDTIME

Specifies a period of applicability that must overlap the period of validity of a row for that row to participate in the evaluation of the abort condition.

For more information see [Sequenced Valid-Time Queries](#).

period_expression

Specifies the period of applicability for the DML statement.

The period of applicability must be a period constant expression that does not reference any columns, but can reference parameterized values and the TEMPORAL_DATE or TEMPORAL_TIMESTAMP built-in functions.

The period of applicability can also be a self-contained noncorrelated scalar subquery that is always nonsequenced in the time dimensions regardless of the temporal qualifier for the DML statement.

Note:

If a *period_expression* is specified, the valid-time column cannot be specified or referenced anywhere in the query. If the valid-time column is a derived period column, the component columns cannot be specified or referenced anywhere in the query.

If *period_expression* is omitted, the period of applicability defaults to PERIOD'(0001-01-01, UNTIL_CHANGED)' for a PERIOD(DATE) valid-time column or PERIOD '(0001-01-01 00:00:00.000000+00:00, UNTIL_CHANGED)' for a PERIOD(TIMESTAMP(*n*) WITH TIME ZONE) valid-time column, where precision *n* and WITH TIME ZONE are optional.

NONSEQUENCED VALIDTIME

Specifies that rows that participate in the evaluation of the abort condition are not further evaluated for qualification in the valid-time dimension.

At least one table referenced in the statement must have valid time.

AND

Specifies a keyword for specifying both a valid-time qualifier and a transaction-time qualifier.

CURRENT TRANSACTIONTIME

Specifies that only rows that are open participate in the evaluation of the abort condition.

At least one table referenced in the statement must have transaction time.

TRANSACTIONTIME AS OF *date_timestamp_expression*

Specifies a given time that must overlap the transaction time of a row for that row to participate in the evaluation of the abort condition.

At least one table referenced in the statement must have transaction time.

NONSEQUENCED TRANSACTIONTIME

Specifies that rows that participate in the evaluation of the abort condition are not further evaluated for qualification in the transaction-time dimension.

At least one table referenced in the statement must have transaction time.

AS OF *date_timestamp_expression*

Specifies that only rows that overlap *date_timestamp_expression* in the valid-time and transaction-time dimension participate in the evaluation of the abort condition.

WORK

Specifies an optional keyword.

'message'

Specifies the text of the message to be returned when the transaction is terminated.

FROM option

Specifies the temporal tables that are further qualified in the WHERE clause.

WHERE *abort_condition*

Specifies an expression where the result must evaluate to TRUE for Vantage to roll back the transaction.

Usage Notes

Omitting a Valid-Time Qualifier

If no temporal qualifier is specified in the valid-time dimension in the statement, the system uses the temporal qualifier of the session. If none is explicitly specified for the session, the default of CURRENT is assumed and only rows that are currently valid participate in further processing.

Omitting a Transaction-Time Qualifier

If no temporal qualifier is specified in the transaction-time dimension in the statement, the system uses the temporal qualifier of the session. If none is specified for the session, the default of CURRENT is assumed and only rows that are open participate in further processing.

Temporal Qualifier and Subqueries

The temporal qualifier of a statement applies to all subqueries; no separate temporal qualifier is allowed for a subquery.

Related Information

For more information on...	See...
ROLLBACK statement	<i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146
FROM clause syntax and usage	FROM Clause (Temporal Form)

SELECT/SELECT INTO (Temporal Forms)

Select returns specific row data from a temporal table in the form of a result table.

SELECT INTO selects at most one row from a temporal table and assigns the values in that row to local variables or parameters in stored procedures.

SELECT Syntax (Temporal Form)

Note:

Teradata Vantage™ - Temporal Table Support describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
[ valid_time_qualifier [ AND transaction_time_qualifier ] |
  transaction_time_qualifier [ AND valid_time_qualifier ] |
  AS OF date_timestamp_expression
]
select_statement
```

valid_time_qualifier

```
{ CURRENT VALIDTIME |
  VALIDTIME AS OF date_timestamp_expression |
  [ SEQUENCED | NONSEQUENCED ] VALIDTIME [ period_expression ]
}
```

transaction_time_qualifier

```
{ { CURRENT | NONSEQUENCED } TRANSACTIONTIME |
  TRANSACTIONTIME AS OF date_timestamp_expression
}
```

SELECT INTO Syntax (Temporal Form)

```
[ valid_time_qualifier [ AND transaction_time_qualifier ] |
  transaction_time_qualifier [ AND valid_time_qualifier ] |
  AS OF date_timestamp_expression
]
select_into_statement
```

valid_time_qualifier

```
{ CURRENT VALIDTIME |
  VALIDTIME AS OF date_timestamp_expression |
  { [ SEQUENCED | NONSEQUENCED ] VALIDTIME } [ period_expression ]
}
```

transaction_time_qualifier

```
{ { CURRENT | NONSEQUENCED } TRANSACTIONTIME |
  TRANSACTIONTIME AS OF date_timestamp_expression
}
```

Note:

To ensure application portability to ANSI standards for temporal SQL, Teradata recommends explicit specification of all temporal qualifiers.

SELECT/SELECT INTO Syntax Elements (Temporal Forms)

CURRENT VALIDTIME

Specifies that the query is current in the valid-time dimension.

At least one table referenced in the query, including tables or views or derived tables mentioned in the FROM clause of a subquery, must be a table that supports valid time.

VALIDTIME AS OF *date_timestamp_expression*

Specifies that the query is AS OF in the valid-time dimension.

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

The expression can be any DATE or TIMESTAMP[(n)] [WITH TIME ZONE] expression, including parameterized values and built-in functions such as CURRENT_DATE or TEMPORAL_DATE, that does not reference any columns. One exception to this rule is that the expression can be a self-contained noncorrelated scalar subquery. A noncorrelated scalar subquery is always assumed to be nonsequenced in the time dimensions regardless of the temporal query qualifier.

If the *date_timestamp_expression* specifies TEMPORAL_DATE or TEMPORAL_TIMESTAMP, the values of these built-in functions evaluate to the time of the transaction.

At least one table referenced in the query, including tables or views or derived tables mentioned in the FROM clause of a subquery, must be a table that supports valid time.

VALIDTIME and SEQUENCED VALIDTIME

Specifies that the query is sequenced in the valid-time dimension.

At least one table referenced in the query, including tables or views or derived tables mentioned in the FROM clause of a subquery, must be a table that supports valid time.

A sequenced valid-time query results in a valid-time table. The valid-time period for each row in the result set is the overlap of original row valid-time with the *period_expression* specified in the query. The valid-time column in the result set a new column named VALIDTIME, which is automatically appended to the results.

If *period_expression* is omitted, the period of applicability for a sequenced query defaults to PERIOD('0001-01-01, UNTIL_CHANGED') where the data type is PERIOD(DATE) or PERIOD('0001-01-01 00:00:00.000000+00:00, UNTIL_CHANGED') where the data type is PERIOD(TIMESTAMP). In these cases, the valid-time periods of the rows in the result set matches the valid-time periods of the original rows in the queried temporal table.

NONSEQUENCED VALIDTIME

Specifies that the query is nonsequenced in the valid-time dimension. The system does not associate any special meaning to the valid-time column. The query can use the valid-time column like any other column.

At least one table referenced in the query, including tables or views or derived tables mentioned in the FROM clause of a subquery, must be a table that supports valid time.

period_expression

Specifies the period of applicability for the DML statement.

The period of applicability must be a period constant expression that does not reference any columns, but can reference parameterized values and the TEMPORAL_DATE or TEMPORAL_TIMESTAMP built-in functions.

The period of applicability can also be a self-contained noncorrelated scalar subquery that is always nonsequenced in the time dimensions regardless of the temporal qualifier for the DML statement.

Note:

If a *period_expression* is specified, the valid-time column cannot be specified or referenced anywhere in the query. If the valid-time column is a derived period column, the component columns cannot be specified or referenced anywhere in the query.

If *period_expression* is omitted, the period of applicability defaults to PERIOD'(0001-01-01, UNTIL_CHANGED)' for a PERIOD(DATE) valid-time column or PERIOD '(0001-01-01 00:00:00.000000+00:00, UNTIL_CHANGED)' for a PERIOD(TIMESTAMP(*n*) WITH TIME ZONE) valid-time column, where precision *n* and WITH TIME ZONE are optional.

For a nonsequenced query, the period of applicability can either be a period value expression that does not reference any column names or an alias name to a period column or period expression.

When an SELECT INTO statement specifies *period_expression*, the number of elements in the INTO target list (that specifies the variables into which the selected values must be saved) must be one more than the projected elements in the select list. This additional target element saves the resulting period of validity of the output row.

AND

Specifies a keyword for specifying both a valid-time qualifier and a transaction-time qualifier.

CURRENT TRANSACTIONTIME

Specifies that the query is current in the transaction-time dimension.

At least one table referenced in the query, including tables or views or derived tables mentioned in the FROM clause of a subquery, must be a table that supports transaction time.

TRANSACTIONTIME AS OF *date_timestamp_expression*

Specifies that the query is AS OF in the transaction-time dimension.

At least one table referenced in the query, including tables or views or derived tables mentioned in the FROM clause of a subquery, must be a table that supports transaction time.

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

NONSEQUENCED TRANSACTIONTIME

Specifies that the query is nonsequenced in the transaction-time dimension.

In a nonsequenced query, the system does not associate any special meaning to the transaction-time column. The query can use the transaction-time column like any other column.

At least one table referenced in the query, including tables or views or derived tables mentioned in the FROM clause of a subquery, must be a table that supports transaction time.

AS OF *date_timestamp_expression*

Specifies the query is an AS OF query in both the valid-time and transaction-time dimensions.

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

All tables with valid time referenced in the query use the specified qualifier to qualify in the valid-time dimension. Similarly, all tables with transaction time referenced in the query use the qualifier to qualify in the transaction-time dimension.

At least one table referenced in the query must be a temporal table. If only valid-time tables and nontemporal tables are referenced, the AS OF qualifier is equivalent to specifying `VALIDTIME AS OF date_timestamp_expression`. If only transaction-time tables and nontemporal tables are referenced, the AS OF qualifier is equivalent to specifying `TRANSACTIONTIME AS OF date_timestamp_expression`. If both valid-time and transaction-time tables are referenced, the AS OF qualifier is equivalent to specifying `VALIDTIME AS OF date_timestamp_expression AND TRANSACTIONTIME AS OF date_timestamp_expression`.

select_statement

Specifies conventional SELECT statement syntax, with temporal table support enhancements to the FROM clause.

For details on FROM, see [FROM Clause \(Temporal Form\)](#).

select_into_statement

Specifies conventional SELECT INTO statement syntax, with temporal table support enhancements to the FROM clause.

For details on FROM, see [FROM Clause \(Temporal Form\)](#).

Usage Notes

General

A temporal qualifier can be specified for the outermost SELECT statement, in a derived table, view, join index and other DDL statements.

A temporal qualifier cannot be specified in a subquery. A subquery inherits the temporal qualifier of its parent query, with one exception. When the qualifier is `NONSEQUENCED VALIDTIME period_expression`, the subquery does not inherit the specified period of applicability.

The absence of a valid-time qualifier in the statement makes the query current in the valid-time dimension if no session valid-time qualifier is available. The absence of a transaction-time qualifier in the statement makes the query current in the transaction-time dimension if no session transaction-time qualifier is available.

The following table provides the meanings for the various combinations of qualifiers for the temporal form of a query.

Temporal SELECT and Options	Meaning
<ul style="list-style-type: none"> • SELECT • CURRENT VALIDTIME SELECT • CURRENT TRANSACTIONTIME SELECT • CURRENT VALIDTIME AND CURRENT TRANSACTIONTIME SELECT 	Query is current in valid time and current in transaction time.
<ul style="list-style-type: none"> • VALIDTIME SELECT • SEQUENCED VALIDTIME SELECT • VALIDTIME AND CURRENT TRANSACTIONTIME SELECT • SEQUENCED VALIDTIME AND CURRENT TRANSACTIONTIME SELECT 	Query is sequenced in valid time and current in transaction time.
<ul style="list-style-type: none"> • VALIDTIME period_expression SELECT • SEQUENCED VALIDTIME period_expression SELECT • VALIDTIME period_expression AND CURRENT TRANSACTIONTIME SELECT • SEQUENCED VALIDTIME period_expression AND CURRENT TRANSACTIONTIME SELECT 	Query is sequenced in valid time and current in transaction time. The rows of interest in the valid-time dimension are all those rows whose valid time overlaps the specified <i>period_expression</i> .
<ul style="list-style-type: none"> • NONSEQUENCED VALIDTIME SELECT • NONSEQUENCED VALIDTIME AND CURRENT TRANSACTIONTIME SELECT 	Query is nonsequenced in valid time and current in transaction time.
<ul style="list-style-type: none"> • NONSEQUENCED VALIDTIME period_expression SELECT • NONSEQUENCED VALIDTIME period_expression AND CURRENT TRANSACTIONTIME SELECT 	Query is nonsequenced in valid time and current in transaction time. The result of the query is a valid-time result with the valid-time value set as the specified <i>period_expression</i> .
<ul style="list-style-type: none"> • VALIDTIME AS OF date_timestamp_expression SELECT • VALIDTIME AS OF date_timestamp_expression AND CURRENT TRANSACTIONTIME SELECT 	Query is AS OF in valid time and current in transaction time.
<ul style="list-style-type: none"> • NONSEQUENCED TRANSACTIONTIME SELECT • CURRENT VALIDTIME AND NONSEQUENCED TRANSACTIONTIME SELECT 	Query is nonsequenced in transaction time and current in valid time.
<ul style="list-style-type: none"> • VALIDTIME AND NONSEQUENCED TRANSACTIONTIME SELECT • SEQUENCED VALIDTIME AND NONSEQUENCED TRANSACTIONTIME SELECT 	Query is nonsequenced in transaction time and sequenced in valid time.

Temporal SELECT and Options	Meaning
<ul style="list-style-type: none"> VALIDTIME <i>period_expression</i> AND NONSEQUENCED TRANSACTIONTIME SELECT SEQUENCED VALIDTIME <i>period_expression</i> AND NONSEQUENCED TRANSACTIONTIME SELECT 	Query is nonsequenced in transaction time and sequenced in valid time. The rows of interest in the tables with valid time support are those rows whose valid time overlaps the specified <i>period_expression</i> .
NONSEQUENCED VALIDTIME AND NONSEQUENCED TRANSACTIONTIME SELECT	Query is nonsequenced in transaction time and nonsequenced in valid time.
NONSEQUENCED VALIDTIME <i>period_expression</i> AND NONSEQUENCED TRANSACTIONTIME SELECT	Query is nonsequenced in transaction time and nonsequenced in valid time. The result of the query is a valid-time result with the valid-time value set to the specified <i>period_expression</i> .
VALIDTIME AS OF <i>date_timestamp_expression</i> AND NONSEQUENCED TRANSACTIONTIME SELECT	Query is nonsequenced in transaction time and AS OF in valid time.
<ul style="list-style-type: none"> TRANSACTIONTIME AS OF <i>date_timestamp_expression</i> SELECT CURRENT VALIDTIME AND TRANSACTIONTIME AS OF <i>date_timestamp_expression</i> SELECT 	Query is AS OF in transaction time and current in valid time.
<ul style="list-style-type: none"> VALIDTIME AND TRANSACTIONTIME AS OF <i>date_timestamp_expression</i> SELECT SEQUENCED VALIDTIME AND TRANSACTIONTIME AS OF <i>date_timestamp_expression</i> SELECT 	Query is AS OF in transaction time and sequenced in valid time.
<ul style="list-style-type: none"> VALIDTIME <i>period_expression</i> AND TRANSACTIONTIME AS OF <i>date_timestamp_expression</i> SELECT SEQUENCED VALIDTIME <i>period_expression</i> AND TRANSACTIONTIME AS OF <i>date_timestamp_expression</i> SELECT 	Query is AS OF in transaction time and sequenced in valid time. The rows of interest in the valid-time tables are those rows whose valid time overlaps the specified <i>period_expression</i> .
NONSEQUENCED VALIDTIME AND TRANSACTIONTIME AS OF <i>date_timestamp_expression</i> SELECT	Query is AS OF in transaction time and nonsequenced in valid time.
NONSEQUENCED VALIDTIME <i>period_expression</i> AND TRANSACTIONTIME AS OF <i>date_timestamp_expression</i> SELECT	Query is AS OF in transaction time and nonsequenced in valid time. The result of the query is a valid-time result with the valid-time value in the rows set to <i>period_expression</i> value.
<ul style="list-style-type: none"> VALIDTIME AS OF <i>date_timestamp_expression</i> AND TRANSACTIONTIME AS OF <i>date_timestamp_expression</i> SELECT AS OF <i>date_timestamp_expression</i> SELECT 	Query is AS OF in transaction time and AS OF in valid time.

Asterisks in Select Lists

For nontemporal tables, an asterisk (*) in a select list causes all table columns to be returned by the SELECT statement. For temporal tables with any qualifier other than nonsequenced, only the nontemporal columns from the original table are returned. To have temporal columns from the original table returned with SELECT statements that use the asterisk to return all nontemporal columns, preface the asterisk with the table name and a period, and follow the asterisk with a comma, then list the temporal columns (or component columns of a temporal column defined as a derived period type) to be returned with the nontemporal columns.

Current Valid-Time Queries

A current valid-time query on a table with valid time considers only open rows where the period of validity overlaps with TEMPORAL_DATE or TEMPORAL_TIMESTAMP in the valid-time dimension. Such rows are called current rows of a table with valid time.

Current valid-time queries on tables with valid time produce snapshot tables as result sets.

The following rules apply to current valid-time queries on valid-time tables.

- If the session valid-time qualifier is implicitly or explicitly set to current, a conventional SELECT statement that does not specify a temporal qualifier is current in the valid-time dimension for a valid-time table. If the query references a valid-time table, the SELECT can specify an optional CURRENT VALIDTIME.
- A current query can reference the valid-time column anywhere in the query. For valid-time columns defined using derived period columns, the query can reference the component columns that define the valid time. The valid-time column is treated as a conventional Period column, or for valid-time columns that are derived, the component columns are treated as conventional DateTime columns. All conditions, including those specified on valid-time columns, apply only to the current rows. Temporal column references can appear in conditions to further filter the output.
- An asterisk (*) in the projection list includes nontemporal columns only.
- Current query processing is as follows:
 1. Extract the current rows of each of the valid-time tables specified in the query and treat the query as if it is specified on a nontemporal table. The resulting table, regardless of whether the projection list includes the valid-time column, is a nontemporal table without the valid-time dimension.
 2. Execute the query as if it were a conventional query that was issued on tables without valid time.

Because current query processing considers a snapshot of valid time, all operations, such as joins and aggregations, are the same as they are for conventional queries.

A current query supports the join of two temporal tables of the same or differing valid-time granularities.

If the query involves a bitemporal or transaction-time table, refer to the following topics for additional information that applies to the transaction-time dimension:

- [Current Transaction-Time Queries](#)
- [AS OF date_time_expression in Transaction-Time Queries](#)
- [Nonsequenced Transaction-Time Queries](#)

NOTICE

CURRENT DML modifications can cause serializability issues for concurrent transactions. See [Potential Concurrency Issues with Current Temporal DML](#) for information on avoiding these issues.

Changing the Behavior of CURRENT VALIDTIME SELECT

CURRENT VALIDTIME SELECT statements normally qualify rows for selection by choosing rows where the valid time overlaps TEMPORAL_TIMESTAMP. Within the transaction that contains the SELECT statement, TEMPORAL_TIMESTAMP reflects the time the transaction was begun, and remains fixed throughout the transaction.

For lengthy transactions this behavior may not be desirable, because rows inserted or changed after the transaction has begun would not be selected. Such rows would have a valid-time period that begins after TEMPORAL_TIMESTAMP, and would therefore be considered future rows, not current rows. For example, if another user adds a row to a table after a transaction has begun, and the transaction performs a CURRENT VALIDTIME SELECT, the new row would not be selected.

The following statement changes the behavior of CURRENT VALIDTIME SELECT statements to qualify rows for selection according to whether the valid time of the row overlaps CURRENT_TIMESTAMP. This allows the SELECT to match rows with the latest timestamps.

```
DIAGNOSTIC SET CURRENT VALIDTIME SELECT AS LATEST [ NOT ] ON FOR SESSION
```

Because CURRENT_TIMESTAMP is not fixed at the time the transaction was begun, this causes CURRENT VALIDTIME SELECT statements to match even the latest rows that were added or changed after the transaction containing the SELECT statement was begun.

This setting affects SELECT statements and subqueries, derived tables, and views within those SELECT statements. It has no effect on other types of DML, or on subqueries, derived tables, and views within those DML statements. The setting is in effect until it is explicitly disabled using the NOT form of the statement.

Note:

Cached SELECT statements are not affected by this diagnostic statement.

AS OF date_time_expression in Valid-Time Queries

When the AS OF clause is specified as a temporal qualifier either explicitly in the statement or implicitly as a session temporal qualifier, its usage covers the entire query. (The AS OF clause can also be specified in

the FROM clause, but such usage covers only the corresponding table in the FROM clause. For details, see [FROM Clause \(Temporal Form\)](#).)

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

The following rules apply to As Of valid-time queries on valid-time tables:

- As Of valid-time queries on valid-time tables produce snapshot tables as results.
- When the AS OF temporal qualifier is specified in the valid-time dimension, it applies to all valid-time tables in the query. The valid-time columns of the valid-time tables are in the scope of the query and they can be used anywhere in the query block, including the WHERE condition and JOIN condition.
- The behavior of a query with an AS OF temporal qualifier is as if a current query was issued at the specified AS OF time. However, a current query reads only valid rows and an As Of query can read rows that are no longer valid.
- Specifying the AS OF qualifier in the valid-time dimension serves as an additional qualification criteria that only rows with a period of validity that overlaps the specified time are eligible to participate in the query. Thereafter, the query treats all the underlying tables as non-valid-time tables. Operations such as joins, aggregations, and set operations are not impacted by this qualifier.
- The data type of *date_time_expression* must be comparable with the element type of the valid-time columns. The following rules apply.

Data Type of <i>date_time_expression</i>	Element Type of Temporal Column	Details
DATE	DATE	The data types are comparable.
DATE	TIMESTAMP[(n)] [WITH TIME ZONE]	The DATE value is cast to TIMESTAMP(n) and used for qualification. The time portion of the converted timestamp value is 00:00:00 (hh:mi:ss) in the session time zone. The row is qualified based on the UTC timestamp values.
TIMESTAMP[(n)] [WITH TIME ZONE]	DATE	The temporal column value is cast to TIMESTAMP and used for qualification. The time portion of the converted timestamp value is 00:00:00 (hh:mi:ss) in the session time zone. The row is qualified based on the UTC timestamp values.
TIMESTAMP[(n)] [WITH TIME ZONE]	TIMESTAMP[(m)] [WITH TIME ZONE]	The timestamp value with coarser precision is converted to the finer precision and then the rows are qualified.
Any other data type	DATE	The system reports an error.
	TIMESTAMP[(n)] [WITH TIME ZONE]	

If the query involves a bitemporal or transaction-time table, refer to the following topics for additional information that applies to the transaction-time dimension:

- [Current Transaction-Time Queries](#)

- [AS OF date_time_expression in Transaction-Time Queries](#)
- [Nonsequenced Transaction-Time Queries](#)

Sequenced Valid-Time Queries

Sequenced temporal queries allow the extraction of the past, current, or future sequence of states of a temporal table. A query that is sequenced in valid time spans those rows with a period of validity that overlaps the period of applicability of the query. Additional conditions can be specified on the valid-time column to further filter the rows as required.

A query that is sequenced in valid time extracts the states of the tables at each point of time as specified in the period of applicability. The resulting table is a valid-time table. The query is over one or more valid-time tables and produces a valid-time result.

Rows having NULL in the valid-time column are not in the result.

The result set includes a new column named VALIDTIME, which is automatically appended by the system. VALIDTIME shows the valid time of the rows in the result set of the query. This is different from the valid times that were originally defined for the rows. Because a sequenced query specifies a particular time period of applicability, the period of validity of the results is limited by that period of applicability. Therefore, the valid time of each row in the result set is the intersection of the period of applicability of the query with the periods of validity of the qualified rows.

Aggregate Functions in Sequenced Queries

Aggregate Functions in Sequenced Queries

The result of a sequenced valid-time query is a temporal table with rows that include a VALIDTIME column. The VALIDTIME column shows the valid time, the time for which the row information is valid, for the rows in the result set. This result set valid time is the intersection of the period of applicability of the query with the periods of validity of the qualified rows in the original queried table.

You can include aggregate functions in sequenced valid-time queries. The aggregation is performed on every distinct duration defined by the combination of VALIDTIME periods in the result set.

Note:

SEQUENCED TRANSACTIONTIME aggregations are not supported.

A valid-time table represents state information that is considered valid for a specified duration, the valid time of the row. The valid data can be constant over the period of time, or may represent values that accumulate over the duration of the valid time period. Examples of state information would be:

- A quantity of an inventory item that represents the amount of the item in stock during a specified period.
- An insurance policy that has a specific duration of time for which it is valid.

- The number of hours worked per unit time, such as hours per day worked on a project, where the valid time is expressed in units of whole days, using a PERIOD (DATE) data type.

It is important to carefully evaluate and choose the columns to which aggregations are applied in temporal tables. If an aggregation over time is applied to a non-state column, the results can be misleading and inaccurate.

- When used on valid-time temporal tables, aggregations such as SUM, AVG, MIN, and MAX should be performed only on state columns. Using such aggregate functions on non-state columns can lead to meaningless or misleading results.
- The COUNT aggregation is generally safe to perform and gives straightforward results. However, you should ensure that the results are interpreted in an appropriate way that corresponds to the columns used for the aggregation.

Nonsequenced Valid-Time Queries

A nonsequenced query operates on all the valid-time states of the underlying table (history, current, future) simultaneously. Such a query is very powerful because it can link across states.

Use a nonsequenced valid-time query to ignore the time-varying nature of a table or when the computation of a single state of the result table utilizes the information from a state at a different time.

A nonsequenced query treats the valid-time column as if it is simply a regular column that contains a period value. The query can specify this column anywhere in the query, just as any column.

A nonsequenced query can select rows with NULL in the valid-time column. None of the other temporal SELECT qualifiers will select such rows.

If the NONSEQUENCED VALIDTIME qualifier does not specify *period_expression*, the nonsequenced query on a valid-time table results in a non-valid-time table.

If *period_expression* is specified, the result of the query is a valid-time table. However, the valid-time column of the results table is not the same column as the valid-time column in the queried table. The results table includes an additional column named VALIDTIME, that serves as the valid-time column. The value of VALIDTIME in each row is the period of applicability that was specified in the query. This form of a nonsequenced valid-time query can be used to convert a nontemporal table to a table with valid time. Such a query is not permitted in a request that specifies a SET operator (UNION, INTERSECT and MINUS).

If the projection list is *, the valid-time column is projected with all other non-valid-time columns.

A reference to VALIDTIME in the same select block in other clauses such as WHERE condition follows the existing resolution rules, and cannot reference the system-projected column. For example, if none of the referenced tables has a column named VALIDTIME an error results. No column in the projection list can use VALIDTIME as an alias.

If an alias name is specified as the period of applicability, the alias name must be in the scope of the select (projection) list and unambiguously referenced in the projection list.

If the query involves a bitemporal or transaction-time table, refer to the following topics for additional information that applies to the transaction-time dimension:

- [Current Transaction-Time Queries](#)
- [AS OF date_time_expression in Transaction-Time Queries](#)
- [Nonsequenced Transaction-Time Queries](#)

Current Transaction-Time Queries

A query that specifies CURRENT TRANSACTIONTIME or a query that omits a temporal qualifier and the session does not set a session temporal qualifier is a current query in the transaction-time dimension if the query references a temporal table with transaction time.

A current query on a temporal table with transaction time considers only those rows in the table that are open in the transaction-time dimension. The result is a snapshot table without transaction time.

References to a transaction-time column can appear anywhere in the query that references to a non-transaction-time column can appear. A transaction-time column is treated as a conventional Period column on the selected current rows whenever it is used. Using this column in conditions can further filter the output over the current rows.

A projection list of * indicates all nontemporal columns only.

A current transaction-time query can join a table with transaction time to a table without transaction time.

The following rules apply to current transaction-time queries on transaction-time tables:

- Current query processing is as follows:
 1. Extract the current rows of each of the transaction-time tables specified in the query and treat the query as if the current snapshot equivalent table is specified in the query. The resulting table, regardless of whether the projection list includes the transaction-time column, is a nontemporal table without the transaction-time dimension.
 2. Execute the query as if it were a conventional query that was issued on nontemporal tables.

If the query involves a bitemporal or valid-time table, refer to the following topics for additional information that applies to the valid-time dimension:

- [Current Valid-Time Queries](#)
- [AS OF date_time_expression in Valid-Time Queries](#)
- [Sequenced Valid-Time Queries](#)
- [Nonsequenced Valid-Time Queries](#)

AS OF date_time_expression in Transaction-Time Queries

date_timestamp_expression can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

The data type of *date_time_expression* must be comparable with `TIMESTAMP(6) WITH TIME ZONE`. The following rules apply.

Data Type of <i>date_time_expression</i>	Details
DATE	The DATE value is cast to <code>TIMESTAMP(6) WITH TIME ZONE</code> and used for qualification. The time portion of the converted timestamp value is 00:00:00 (hh:mi:ss) in the session time zone. The row is qualified based on the UTC timestamp values.
<code>TIMESTAMP[(n)] [WITH TIME ZONE]</code>	If the timestamp value has a coarser precision, it is converted to <code>TIMESTAMP(6) WITH TIME ZONE</code> and then the rows are qualified.
Any other data type	The system reports an error.

References to a transaction-time column can appear anywhere in the scope of the query and anywhere in the query block, including a WHERE condition or JOIN condition.

The AS OF qualifier serves as an additional qualification criteria such that only rows with a transaction-time value that overlaps the given time are eligible to participate in the query. Thereafter, the query treats all the underlying tables as non-transaction-time tables. Operations such as joins, aggregation, set operations, and so forth are not impacted by this qualifier.

If *date_timestamp_expression* in the transaction-time dimension uses `TEMPORAL_DATE` or `TEMPORAL_TIMESTAMP`, the value of the built-in function evaluates to the time of the transaction. If *date_timestamp_expression* is a value that is in the future, the qualifier is as if it is current in the transaction-time dimension.

If the query involves a bitemporal or valid-time table, refer to the following topics for additional information that applies to the valid-time dimension:

- [Current Valid-Time Queries](#)
- [AS OF date_time_expression in Valid-Time Queries](#)
- [Sequenced Valid-Time Queries](#)
- [Nonsequenced Valid-Time Queries](#)

Nonsequenced Transaction-Time Queries

Nonsequenced transaction-time queries on transaction-time tables produce nontemporal tables as a result. Use the `NONSEQUENCED TRANSACTIONTIME` qualifier to query and compare across all transaction-time states simultaneously.

The nonsequenced query treats a transaction-time table as a table with a regular Period column with no special temporal semantics. References to the transaction-time column can appear anywhere in the query.

If the projection list specifies *, the transaction-time column is also projected.

If the query involves a bitemporal or valid-time table, refer to the following topics for additional information that applies to the valid-time dimension:

- [Current Valid-Time Queries](#)
- [AS OF date_time_expression in Valid-Time Queries](#)
- [Sequenced Valid-Time Queries](#)
- [Nonsequenced Valid-Time Queries](#)

Derived Period Columns and SELECT

In addition to the regular rules for use of derived period columns with SELECT described in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146, the following rules and restrictions apply to the use of derived VALIDTIME and TRANSACTIONTIME columns in temporal tables.

- Component columns of derived VALIDTIME and TRANSACTIONTIME columns are not projected for CURRENT and SEQUENCED SELECT * requests.
- Component columns of derived VALIDTIME and TRANSACTIONTIME columns are projected for NONSEQUENCED SELECT * requests.
- The system generated VALIDTIME column that is projected for a SEQUENCED VALIDTIME query will be a true period data type column, even if the valid-time column of the queried table is a derived period column.

Temporal Queries in Set Operations

When temporal tables are referenced in queries involving set operations (UNION, INTERSECT, MINUS, and EXCEPT) all queries inherit the temporal qualifier from the topmost query.

If temporal qualifications are required at the level of individual queries, add them to the FROM clause or place the query with the required qualification in a derived table.

EXPLAIN Request Modifier

You can use the EXPLAIN, STATIC EXPLAIN, and DYNAMIC EXPLAIN request modifiers to report temporal semantic operations. The explain text provides the type of qualifier applied on the temporal tables being operated upon. It reports whether the query is current, sequenced, or nonsequenced.

Here is an example of the EXPLAIN report for a current query on a row-partitioned bitemporal table:

```
EXPLAIN SELECT * FROM Policy;
```

Explanation

- ```

1) First, we lock DBASE.Policy for read on a reserved RowHash
 in all partitions to prevent global deadlock for DBASE.Policy.
```

```

2) Next, we lock DBASE.Policy for read.
3) We do an all-AMPs RETRIEVE step from a single partition of
 DBASE.Policy (with temporal qualifier as "CURRENT
VALIDTIME AND CURRENT TRANSACTIONTIME") with a condition of (
"((BEGIN(DBASE.Policy.Validity))<= DATE '2010-02-18')
AND (((END(DBASE.Policy.Policy_Duration))= TIMESTAMP
'9999-12-31 23:59:59.999999+00:00') AND
((END(DBASE.Policy.Validity))> DATE '2010-02-18'))")
 into Spool 1 (group_amps), which is built locally on the AMPs.
 The size of Spool 1 is estimated with no confidence to be 1 row (
 85 bytes). The estimated time for this step is 0.03 seconds.
4) Finally, we send out an END TRANSACTION step to all AMPs involved
 in processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
 statement 1. The total estimated time is 0.03 seconds.

```

## Time Series Expansion Support

Time series expansion is indicated using the EXPAND ON clause for SELECT statements, and provides the ability to create a regular time series of rows based on period values in the input rows. The intent is to expand a period column and produce value equivalent rows, one for each granule in a given period of interest.

For example, suppose an application uses PERIOD(DATE) values to record inventory data of slowly moving items. A SELECT statement can use the EXPAND ON clause to expand the period values by one-week intervals to get the moving average by week of the inventory cost for a specified period of interest.

To expand the system-generated VALIDTIME column that is automatically appended to the result set of a SEQUENCED VALIDTIME query, the EXPAND ON clause can include the VALIDTIME column name in the expand expression.

## SELECT on Normalized Tables

The NORMALIZE keyword can be used optionally in SELECT statements to have the results set normalized. The NORMALIZE option of SELECT can be used on tables that do or do not have NORMALIZE in their table definitions.

The following considerations apply to the use of NORMALIZE with SELECT requests on temporal tables:

- At least one column in the select list must be a period column or derived period column.
- The first period column in the select list is the column that is normalized.
- When a SEQUENCED VALIDTIME SELECT uses NORMALIZE, and no period column is projected in the select list, normalization happens on the system projected "VALIDTIME" column.

- Normalization cannot be performed using a SELECT statement on a derived period column. As a work around, a true period column can be constructed from the column components of the derived period column.
- Note that normalization is rarely effective for sequenced select statements on temporal tables that include transaction-time columns, because most of the values in the transaction-time column are likely to be unique, preventing normalization.
- NORMALIZE can be specified in subqueries.

## Examples

### Example: Asterisks in SELECT Lists

```
SELECT *
```

```
FROM policy;
```

| Policy_ID | Customer_ID | Policy_Type | Policy_Details    |
|-----------|-------------|-------------|-------------------|
| 541077    | 766492008   | AU          | STD-CH-344-YXY-00 |
| 541008    | 246824626   | AU          | STD-CH-345-NXY-00 |
| 541145    | 616035020   | AU          | STD-CH-348-YXN-01 |

```
SELECT policy.*, validity
```

```
FROM Policy;
```

| Policy_ID | Customer_ID | Policy_Type | Policy_Details    | Validity                 |
|-----------|-------------|-------------|-------------------|--------------------------|
| 541077    | 766492008   | AU          | STD-CH-344-YXY-00 | ('09/12/21', '99/12/31') |
| 541008    | 246824626   | AU          | STD-CH-345-NXY-00 | ('09/10/01', '99/12/31') |
| 541145    | 616035020   | AU          | STD-CH-348-YXN-01 | ('09/12/03', '10/12/01') |

#### Note:

Valid-time columns cannot be referenced in SEQUENCED VALIDTIME queries that include a period of applicability.

### Example: Sequenced Valid-Time Query

```
SEQUENCED VALIDTIME PERIOD '(2009-01-01, 2009-12-31)'
```

```
SELECT *
```

```
FROM Policy;
```

| Policy_ID | Customer_ID | Policy_Type | Policy_Details    | VALIDTIME                |
|-----------|-------------|-------------|-------------------|--------------------------|
| 541077    | 766492008   | AU          | STD-CH-344-YXY-00 | ('09/12/21', '09/12/31') |

|        |              |                   |                          |
|--------|--------------|-------------------|--------------------------|
| 541008 | 246824626 AU | STD-CH-345-NXY-00 | ('09/10/01', '09/12/31') |
| 541145 | 616035020 AU | STD-CH-348-YXN-01 | ('09/12/03', '09/12/31') |

Although VALIDTIME is the valid-time column of the result set, Validity is the valid-time column of the originally queried Policy table. To show the Validity column in the results requires a subquery, because the valid-time column name cannot appear anywhere in a query that includes a PA. Use a sequenced validtime subquery that does not specify a PA. Because the Validity column is not the valid-time column of the derived table, it can be retrieved using a sequenced outer query that includes a PA:

### Example: Sequenced Valid-Time Query Returning Original Valid-time Column

```
SEQUENCED VALIDTIME PERIOD '(2009-01-01, 2009-12-31)'
SELECT Policy_ID, Customer_ID, Validity FROM (
 SEQUENCED VALIDTIME SELECT Policy.*, Validity
 FROM Policy) AS my_derived_table;
```

| Policy_ID | Customer_ID | Validity                 | VALIDTIME                |
|-----------|-------------|--------------------------|--------------------------|
| 541077    | 766492008   | ('09/12/21', '99/12/31') | ('09/12/21', '09/12/31') |
| 541008    | 246824626   | ('09/10/01', '99/12/31') | ('09/10/01', '09/12/31') |
| 541145    | 616035020   | ('09/12/03', '10/12/01') | ('09/12/03', '09/12/31') |

Comparing the Validity and VALIDTIME columns demonstrates that the valid-time period of the result set is the intersection of the valid-time period in the original Policy table Validity column with the PA specified in the sequenced select statement.

This example uses the SELECT \* asterisk notation in combination with explicit specification of a temporal column. For more information on the asterisk see [Asterisks in Select Lists](#)

Other clauses of the SELECT statement, such as WHERE, cannot reference the new VALIDTIME column, and VALIDTIME cannot be used as an alias for any other projected columns.

If a view or a derived table results in a valid-time table, it is treated like any other valid-time table specified in the query. Otherwise, it is treated like a nontemporal table.

The sequenced form of a temporal query is limited to a simple select from a single valid-time table or a simple select with inner joins from multiple tables. A noncorrelated scalar subquery can be used in the temporal query.

The following are not supported for sequenced queries:

- Outer joins
- Set operations
- Ordered analytic functions
- Subqueries other than noncorrelated scalar subqueries
- WITH, WITH RECURSIVE, TOP *n*, DISTINCT

The following rules apply to sequenced inner joins:

- The term *row* or *rows* mentioned in the context of sequenced inner joins implies all the rows of a table where the period of validity overlaps with the period of applicability of the query and satisfies any specified single table conditions.
- The joined row contains the projected columns in the specified order and a valid-time column whose value is set to the result of the intersection of the period of applicability, the period of validity of the left row, and the period of validity of the right row.
- If the valid-time column of each table involved in the join has the same granularity as the period of applicability, the granularity of the resulting period of validity is the same as the granularity of the period of applicability. Otherwise, the granularity of the result period of validity is the finest amongst them and each valid-time column is implicitly converted to the finest valid-time granularity before the join is performed.

For example, consider a sequenced join with a `PERIOD(DATE)` period of applicability, a table with a `PERIOD(TIMESTAMP(3))` valid-time column, and a table with a `PERIOD(TIMESTAMP(5))` valid-time column. The resulting `VALIDTIME` column has a `PERIOD(TIMESTAMP(5))` data type.

The granularity difference does not usually cause an issue. However, it can be confusing for some types of projected columns. For example a column that represents charge per day in the original table can be confusing when it is projected in a result having a validity of hours. Such cases can be avoided if tables with similar granularities are joined, or the data model specifies the same granularity for tables that are likely to be joined.

Similarly if a non-temporal table is joined with a temporal table, the resulting rows reflect the validity of the temporal table. This validity can be misleading if applied to data that was not originally of a temporal nature.

- For a sequenced inner join, the left row is joined with a matching row from the right table only if the periods of validity of the rows overlap.

A sequenced query with an `ORDER BY` clause may specify ordering on the resulting `VALIDTIME` column. To specify this column in the `ORDER BY` clause, use the `VALIDTIME` keyword or use its name delimited by double quotation marks ("`VALIDTIME`"). If the ordering clause does not specify `VALIDTIME`, the resulting `VALIDTIME` column is automatically made a part of the `ORDER BY` list as the last element in the list with `ASC` (ascending) default ordering

If the query involves a bitemporal or transaction-time table, refer to the following topics for additional information that applies to the transaction-time dimension:

- [Current Transaction-Time Queries](#)
- [AS OF date\\_time\\_expression in Transaction-Time Queries](#)
- [Nonsequenced Transaction-Time Queries](#)

## Example: COUNT Aggregation in a Sequenced Query

Consider the following information about three jobs performed by an aircraft service company:

| ID  | Job_Type     | Charge | Duration                | NumWorkersAssigned |
|-----|--------------|--------|-------------------------|--------------------|
| 123 | Wing         | 80     | 4 Jan 2011 – 8 Jan 2011 | 5                  |
| 123 | Fuselage     | 20     | 5 Jan 2011 – 7 Jan 2011 | 3                  |
| 123 | Landing Gear | 6      | 6 Jan 2011 – 9 Jan 2011 | 1                  |

If the information is stored in a temporal table, where duration is the valid time column, the valid times for all the rows can be laid out like this:

```

Jan 4 5 6 7 8 9
 |-----|
 |---|
 |-----|

```

Aggregations in sequenced queries partition the combined valid-times of all rows into overlapping and non-overlapping durations. In the example, the valid time periods describe five distinct durations, defined by the boundaries of all the valid-time periods for all the rows in the table:

```

Jan 4 5 6 7 8 9
 |-|-|-|-|-|

```

Jan 4-5, 5-6, 6-7, 7-8, and 8-9.

A sequenced query of the whole table (with no WHERE clause qualification) would return aggregations across each of the five distinct durations:

```

SEQUENCED VALIDTIME
SELECT id, COUNT(*) jobcount
FROM aircraft_service
GROUP BY 1
ORDER BY VALIDTIME;

```

Returns:

| ID  | Jobcount | VALIDTIME                |
|-----|----------|--------------------------|
| 123 | 1        | ('11/01/04', '11/01/05') |
| 123 | 2        | ('11/01/05', '11/01/06') |
| 123 | 3        | ('11/01/06', '11/01/07') |
| 123 | 2        | ('11/01/07', '11/01/08') |
| 123 | 1        | ('11/01/08', '11/01/09') |

**Note:**

If you use an ORDER BY VALIDTIME clause with a sequenced valid-time query that includes an aggregation, the ordering takes into account the generated VALIDTIME distinct durations that were calculated to perform the aggregation.

If you use a GROUP BY VALIDTIME clause, the grouping does not use the distinct durations, but instead uses the same VALIDTIME values assigned to the results of nonaggregate queries: the intersection of the period of applicability of the query with the periods of validity of the qualified rows in the original queried table.

**Example: MIN and MAX Aggregations in a Sequenced Query**

MIN and MAX functions can be used in sequenced aggregations with temporal tables, however it is important to apply them to appropriate columns. For the aircraft service example, the information in the Charge column is not a state value. Applying MIN and MAX to this column in a time dependent sequenced aggregation would give results that are easily misinterpreted or not meaningful.

However, it would be meaningful to ask for the minimum and maximum number of workers assigned to all jobs combined at any time:

```
SEQUENCED VALIDTIME
SELECT id,
 min(NumWorkersAssigned) as Minworkers,
 max(NumWorkersAssigned) as Maxworkers
FROM aircraft_service
GROUP BY 1
ORDER BY VALIDTIME;
```

| ID  | Minworkers | Maxworkers | VALIDTIME                |
|-----|------------|------------|--------------------------|
| 123 | 5          | 5          | ('11/01/04', '11/01/05') |
| 123 | 3          | 5          | ('11/01/05', '11/01/06') |
| 123 | 1          | 5          | ('11/01/06', '11/01/07') |
| 123 | 1          | 5          | ('11/01/07', '11/01/08') |
| 123 | 1          | 1          | ('11/01/08', '11/01/09') |

**Example: SUM and AVG Aggregations in a Sequenced Query**

SUM and AVG aggregations can be used over state columns of valid-time temporal tables. In our aircraft service example, the information in the NumWorkersAssigned column is a state value, valid for the

duration specified in the valid-time column. It is meaningful to ask about the total or average number of workers assigned to an aircraft, as in the following example.

```

SEQUENCED VALIDTIME
SELECT id,
 SUM (NumWorkersAssigned) TotalWorkersAssigned,
 AVG (NumWorkersAssigned) AvgWorkersAssigned,
FROM aircraft_service
GROUP BY 1
ORDER BY VALIDTIME;

```

| ID  | TotalWorkersAssigned | AvgWorkersAssigned | VALIDTIME                |
|-----|----------------------|--------------------|--------------------------|
| 123 | 5                    | 5                  | ('11/01/04', '11/01/05') |
| 123 | 8                    | 4                  | ('11/01/05', '11/01/06') |
| 123 | 9                    | 3                  | ('11/01/06', '11/01/07') |
| 123 | 6                    | 3                  | ('11/01/07', '11/01/08') |
| 123 | 1                    | 1                  | ('11/01/08', '11/01/09') |

## Example: Ensuring Meaningful Results from a Sequenced Aggregation

To use aggregate functions meaningfully on the charge column, it would need to be expressed as a cumulative value that matches the granularity of the valid time column. For example, if the charge for each aircraft service job is expressed, in this case, as a charge per day, sequenced aggregations such as SUM and AVG can be applied with meaningful results.

For example, assume the data for the aircraft company had the charge for each type of service job expressed as a charge per day.

| ID  | Job_Type     | ChargePerDay | Duration                |
|-----|--------------|--------------|-------------------------|
| 123 | Wing         | 20           | 4 Jan 2011 – 8 Jan 2011 |
| 123 | Fuselage     | 10           | 5 Jan 2011 – 7 Jan 2011 |
| 123 | Landing Gear | 2            | 6 Jan 2011 – 9 Jan 2011 |

The following sequenced aggregate query yields meaningful results.

```

SEQUENCED VALIDTIME
SELECT id,
 SUM (ChargePerDay) TotalChargePerDay ,
 AVG (ChargePerDay) AvgChargePerDay

```

```
FROM aircraft_service
GROUP BY 1
ORDER BY VALIDTIME;
```

| ID  | TotalChargePerDay | AveChargePerDay | VALIDTIME                |
|-----|-------------------|-----------------|--------------------------|
| 123 | 20                | 20              | ('11/01/04', '11/01/05') |
| 123 | 30                | 15              | ('11/01/05', '11/01/06') |
| 123 | 32                | 11              | ('11/01/06', '11/01/07') |
| 123 | 22                | 11              | ('11/01/07', '11/01/08') |
| 123 | 2                 | 2               | ('11/01/08', '11/01/09') |

### Example: Using a sequenced aggregation with GROUP BY

Sequenced aggregations together with GROUP BY in the query includes "empty" valid-time periods during which none of the temporal table rows is valid. Assume the temporal table from the example above included another row for which the valid-time period was not contiguous with any of the existing valid time periods.

| ID  | Job_Type | ChargePerDay | Duration                |
|-----|----------|--------------|-------------------------|
| 123 | Cockpit  | 40           | 1 Jan 2012 – 1 Mar 2012 |

The query above would have returned two additional rows, one for the new duration, and one for the period that included no aircraft maintenance work:

| ID  | TotalChargePerDay | AvgChargePerDay | VALIDTIME                |
|-----|-------------------|-----------------|--------------------------|
| 123 | ? (NULL)          | ? (NULL)        | ('11/01/09', '12/01/01') |
| 123 | 40                | 40              | ('12/01/01', '12/03/01') |

This allows you to answer questions such as "During the time covered by the table, when were no aircraft undergoing service?"

```
SEQUENCED VALIDTIME PERIOD(date'2011-01-01', date'2012-03-01')
SELECT id FROM aircraft_service
 HAVING COUNT(ChargePerDay)= 0
 GROUP BY 1
 ORDER BY 1;
```

| ID  | VALIDTIME                |
|-----|--------------------------|
| 123 | ('11/01/09', '12/01/01') |

To prevent the "empty" valid-time period from being returned, add a condition to the query using the HAVING clause to filter out these rows.

## Example: Proper form of temporal queries using set operations

The following query would yield an error:

```
VALIDTIME AS OF DATE '2009-05-06'
SELECT *
FROM v1

MINUS

VALIDTIME AS OF DATE '2009-05-05'
SELECT *
FROM v1

ORDER BY 1;
```

The following query would run properly to yield the desired results:

```
SELECT *
FROM v1 VALIDTIME AS OF DATE '2009-05-06'

MINUS

SELECT *
FROM v1 VALIDTIME AS OF DATE '2009-05-05'

ORDER BY 1;
```

## Related Information

| For more information on...                                      | See...                                                                        |
|-----------------------------------------------------------------|-------------------------------------------------------------------------------|
| SELECT statement, including information on the NORMALIZE option | <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146         |
| SELECT INTO statement                                           | <i>Teradata Vantage™ - SQL Stored Procedures and Embedded SQL</i> , B035-1148 |
| EXPAND ON clause                                                | <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146         |
| querying temporal tables                                        | <a href="#">Querying Temporal Tables</a>                                      |

| For more information on...      | See...                                 |
|---------------------------------|----------------------------------------|
| types of temporal table queries | <a href="#">Temporal Table Queries</a> |

## FROM Clause (Temporal Form)

Defines the set of tables, derived tables, or views that are referenced by the SELECT request.

## FROM Clause Syntax (Temporal Form)

### Note:

*Teradata Vantage™ - Temporal Table Support* describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
FROM source_spec [, ...]
```

### *source\_spec*

```
{ table_name [temporal_qualifier] [[AS] correlation_name] |
 (subquery) [AS] derived_table_name
 [(column_name [, ...])] [temporal_qualifier] |
 joined_table [temporal_qualifier]
 { CROSS JOIN single_table |
 [INNER | { LEFT | RIGHT | FULL } [OUTER]]
 JOIN joined_table [temporal_qualifier] ON search_condition
 }
}
```

### *temporal\_qualifier*

```
{ valid_time_qualifier [AND transaction_time_qualifier] |
 transaction_time_qualifier [AND valid_time_qualifier] |
```

```
AS OF { date_timestamp_expression | (date_timestamp_expression) }
}
```

***valid\_time\_qualifier***

```
{ VALIDTIME AS OF
 { date_timestamp_expression | (date_timestamp_expression) } |
 { CURRENT | NONSEQUENCED } VALIDTIME
}
```

***transaction\_time\_qualifier***

```
{ TRANSACTIONTIME AS OF
 { date_timestamp_expression | (date_timestamp_expression) } |
 { CURRENT | NONSEQUENCED } TRANSACTIONTIME
}
```

**FROM Clause Syntax Elements (Temporal Form)*****table\_name***

Specifies the name of a base table, temporal table, derived table, or view.

**VALIDTIME AS OF *date\_timestamp\_expression*****VALIDTIME AS OF (*date\_timestamp\_expression*)**

Specifies that the retrieval of rows from *table\_name* only includes rows where the period of validity overlaps the specified AS OF date or timestamp.

*date\_timestamp\_expression* can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

The data type of *date\_timestamp\_expression* must be comparable with the element type of the valid-time column.

**AND**

Specifies a keyword for specifying both a valid-time AS OF qualifier and a transaction-time AS OF qualifier.

**TRANSACTIONTIME AS OF *date\_timestamp\_expression*****TRANSACTIONTIME AS OF (*date\_timestamp\_expression*)**

Specifies that the retrieval of rows from *table\_name* only includes rows where the transaction-time period in the rows overlaps the specified AS OF date or timestamp.

*date\_timestamp\_expression* can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a date or timestamp value.

The data type of *date\_timestamp\_expression* must be comparable with the element type of the transaction-time column.

### CURRENT VALIDTIME

Specifies that the retrieval of rows from *table\_name* includes only current rows in the valid-time dimension, which are rows having a period of validity that overlaps TEMPORAL\_DATE or TEMPORAL\_TIMESTAMP. The valid-time column of *table\_name* is considered to be in the scope of the query, and can therefore be used as any other column in the table, such as in a WHERE or JOIN condition.

CURRENT VALIDTIME in a FROM clause takes precedence over a validtime qualifier at the statement level. For example, consider the following query:

```
SEQUENCED VALIDTIME SELECT Policy.policy_id
FROM Policy CURRENT VALIDTIME, Policy_History;
```

In this case, the SEQUENCED VALIDTIME at the statement level is ignored for Policy, which has a temporal qualifier in the FROM clause. SEQUENCED VALIDTIME is applied to Policy\_History, however, which does not have a temporal qualifier in the FROM clause.

### NONSEQUENCED VALIDTIME

Specifies that *table\_name* is treated as a non-temporal table. Results in a table without a valid-time column.

### CURRENT TRANSACTIONTIME

Specifies that the retrieval of rows from *table\_name* includes only rows that are open in the transaction-time dimension.

CURRENT TRANSACTIONTIME in a FROM clause takes precedence over a transactiontime qualifier at the statement level. For example, consider the following query:

```
NONSEQUENCED TRANSACTIONTIME SELECT Policy_Types.Policy_Name
FROM Policy_Types CURRENT TRANSACTIONTIME, Policy_History;
```

In this case, the NONSEQUENCED TRANSACTIONTIME at the statement level is ignored for Policy\_Types, which has a temporal qualifier in the FROM clause. NONSEQUENCED TRANSACTIONTIME is applied to Policy\_History, however, which does not have a temporal qualifier in the FROM clause.

**NONSEQUENCED TRANSACTIONTIME**

Specifies that *table\_name* is treated as a non-temporal table. Results in a table without a transaction-time column.

**AS OF *date\_timestamp\_expression*****AS OF (*date\_timestamp\_expression*)**

Specifies a date or timestamp value that qualifies the retrieval of rows from *table\_name* in all existing time dimensions.

The data type of *date\_timestamp\_expression* must be comparable with the element types of all temporal columns.

**[AS] *correlation\_name***

Specifies an alias for the table that is referenced by *table\_name*.

**(*subquery*)**

Specifies the subquery that defines the derived table contents.

**[AS] *derived\_table\_name***

Specifies an assigned name for the temporary derived table.

***column\_name***

Specifies a list of column names or expressions listed in the subquery. Allows referencing subquery columns by name.

***joined\_table***

Specifies either a single table name with optional alias name, or a joined table, indicating nested joins.

---

**Note:**

AS OF is valid only when *joined\_table* is a single table name with optional alias name.

---

**CROSS JOIN**

Specifies a cross join.

A CROSS JOIN is an unconstrained, or extended, Cartesian join.

Cross joins return the concatenation of all rows from the tables specified in its arguments.

Two joined tables can be cross joined.

***single\_table***

Specifies the name of a single base or derived table or view on a single table to be cross joined with *joined\_table*.

**[INNER] JOIN**

Specifies a join in which qualifying rows from one table are combined with qualifying rows from another table according to some join condition.

This is the default join type.

**LEFT OUTER JOIN**

Specifies a left outer join.

LEFT indicates the table that was listed first in the FROM clause.

In a LEFT OUTER JOIN, matching rows as well as the rows from the left table that are not returned in the result of the inner join of the two tables, are returned in the outer join result and extended with nulls.

**RIGHT OUTER JOIN**

Specifies a right outer join.

RIGHT indicates the table that was listed second in the FROM clause.

In a RIGHT OUTER JOIN, matching rows as well as the rows from the right table that are not returned in the result of the inner join of the two tables, are returned in the outer join result and extended with nulls.

**FULL OUTER JOIN**

Specifies a full outer join.

FULL OUTER JOIN returns rows from both tables.

In a FULL OUTER JOIN, matching rows as well as rows from both tables that have not been returned in the result of the inner join, are returned in the outer join result and extended with nulls.

**ON *search\_condition***

Specifies one or more conditional expressions that must be satisfied by the result rows.

An ON condition clause is required for each INNER JOIN or OUTER JOIN specified in an outer join expression.

## Temporal Qualifier Usage Notes

- Temporal qualifiers can only be used on tables with corresponding temporal dimensions. For example you can only use a VALIDTIME qualifier on a valid-time or bitemporal table.
- A temporal qualifier in a FROM clause takes precedence over a qualifier for the same temporal dimension at the statement level. For example, consider the following query:

```
SEQUENCED VALIDTIME SELECT Policy.policy_id
FROM Policy CURRENT VALIDTIME, Policy_History;
```

In this case, the SEQUENCED VALIDTIME at the statement level is ignored for Policy, which has a temporal qualifier in the FROM clause. SEQUENCED VALIDTIME is applied to Policy\_History, however, which does not have a temporal qualifier in the FROM clause.

- The valid-time or transaction-time column of *table\_name* that corresponds to the FROM clause temporal qualifier is considered to be in the scope of the query, and can therefore be referenced anywhere in the query, such as in a WHERE or JOIN condition.
- You can apply the AS OF qualifier to valid-time and transaction-time dimensions independently.
- If *date\_timestamp\_expression* of an AS OF qualifier references a table column, Vantage applies the AS OF qualifier to the temporal table for each value in the referenced column, and returns the combination of all results.

```
SELECT Policy.customer_id, Incident.Id
FROM Policy AS OF Incident.Injury_date
WHERE Policy.customer_id = Incident.customer_id ;
```

- If *date\_timestamp\_expression* references a table column, and that table is involved in a join using an explicit JOIN clause, the conditions resulting from the AS OF qualification are added to the WHERE clause, not the ON clause. For example, this SELECT statement:

```
SELECT Policy.customer_id, Incident.Id
FROM Policy AS OF Incident.Injury_date
LEFT OUTER JOIN Incident ON Policy.customer_id = Incident.customer_id;
```

would be equivalent to this SELECT statement:

```
SELECT Policy.customer_id, Incident.Id
FROM (NONSEQUENCED VALIDTIME SELECT * FROM Policy)Policy
LEFT OUTER JOIN Incident ON Policy.customer_id = Incident.customer_id
WHERE Policy.Policy_Term CONTAINS Incident.Injury_date;
```

- The table associated with a temporal qualifier in the FROM clause of an UPDATE or DELETE statement cannot be the table that the DELETE or UPDATE statement modifies.

## Related Information

| For more information on... | See...                                                                |
|----------------------------|-----------------------------------------------------------------------|
| FROM clause                | <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146 |

## UPDATE (Temporal Form)

Modifies column values in existing rows of a temporal table.

## Required Privileges

The following privileges are required in addition to those privileges required for a conventional UPDATE statement:

- If the UPDATE statement specifies the NONTEMPORAL qualifier, the NONTEMPORAL privilege is also required on the temporal table.
- For a current or sequenced update to a table with valid time, the UPDATE privilege is required on the valid-time column of the table.

## UPDATE Syntax (Temporal Form)

### Note:

*Teradata Vantage™ - Temporal Table Support* describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
[{ CURRENT | NONSEQUENCED } VALIDTIME |
 [SEQUENCED] VALIDTIME [period_expression] |
 NONTEMPORAL
]
update_statement [;]
```

### Note:

To ensure application portability to ANSI standards for temporal SQL, Teradata recommends explicit specification of all temporal qualifiers.

## UPDATE Syntax Elements (Temporal Form)

### CURRENT VALIDTIME

Specifies that the update is current in the valid-time dimension if the target table supports valid time.

#### Note:

A current UPDATE affects only current rows. Future rows with valid-time periods that do not overlap TEMPORAL\_TIMESTAMP or TEMPORAL\_DATE will not be updated

If the target table does not support valid time, at least one of the referenced tables must be a table with valid time. The update is not a current update. The CURRENT VALIDTIME qualifier is used to qualify rows from the referenced tables in the valid-time dimension. In the transaction-time dimension, open rows qualify.

If the session temporal qualifier is not set and the temporal qualifier is omitted from the UPDATE statement, the default qualifier is CURRENT VALIDTIME.

### NOTICE

CURRENT DML modifications can cause serializability issues for concurrent transactions. See [Potential Concurrency Issues with Current Temporal DML](#) for information on avoiding these issues.

### VALIDTIME and SEQUENCED VALIDTIME

Specifies that the update is sequenced in the valid-time dimension if the target table supports valid time.

If the target table does not support valid time, at least one of the referenced tables must be a table with valid time. The update is not a sequenced update. The VALIDTIME or SEQUENCED VALIDTIME qualifier is used to qualify rows from the referenced tables in the valid-time dimension. In the transaction-time dimension, open rows qualify.

A sequenced update on a target table that supports valid time sets the valid-time value to the intersection of the valid-time column value and *period\_expression*.

### *period\_expression*

Specifies the period of applicability for the DML statement.

The period of applicability must be a period constant expression that does not reference any columns, but can reference parameterized values and the TEMPORAL\_DATE or TEMPORAL\_TIMESTAMP built-in functions.

The period of applicability can also be a self-contained noncorrelated scalar subquery that is always nonsequenced in the time dimensions regardless of the temporal qualifier for the DML statement.

---

**Note:**

If a *period\_expression* is specified, the valid-time column cannot be specified or referenced anywhere in the query. If the valid-time column is a derived period column, the component columns cannot be specified or referenced anywhere in the query.

---

If *period\_expression* is omitted, the period of applicability defaults to PERIOD('0001-01-01, UNTIL\_CHANGED') for a PERIOD(DATE) valid-time column or PERIOD('0001-01-01 00:00:00.000000+00:00, UNTIL\_CHANGED') for a PERIOD(TIMESTAMP( *n* ) WITH TIME ZONE) valid-time column, where precision *n* and WITH TIME ZONE are optional.

### NONSEQUENCED VALIDTIME

Specifies that the update is nonsequenced in the valid-time dimension if the target table supports valid time.

If the target table does not support valid time, at least one of the referenced tables must be a table with valid time. The update is not a nonsequenced update. The NONSEQUENCED VALIDTIME qualifier is used to qualify rows from the referenced tables in the valid-time dimension. In the transaction-time dimension, open rows qualify.

### NONTEMPORAL

Specifies that the update is nonsequenced in the valid-time dimension and nontemporal in the transaction-time dimension.

A nontemporal update treats the transaction-time column as a nontemporal column.

The table must support transaction time.

### *update\_statement*

Specifies conventional syntax for the UPDATE statement.

For a current or sequenced update on a table with valid time, the SET clause cannot reference the valid-time column as the name of the column whose data is to be updated.

For an update on a table with transaction time, the SET clause cannot reference the transaction-time column as the name of the column whose data is to be updated, unless the NONTEMPORAL qualifier is used.

The SET clause cannot use CURRENT\_DATE or CURRENT\_TIMESTAMP in the expressions that are used to assign a value to the valid-time column.

## Usage Notes

### General

Unless the `NONTEMPORAL` qualifier is specified, updates on temporal tables are always current in the transaction-time dimension.

All check, primary key, and temporal unique (current, sequenced, nonsequenced) constraints defined on the table are checked only on rows that are open in transaction time.

DML operations on tables defined with `NORMALIZE` produce a normalized set of modified rows. Some unmodified rows may be deleted from the target table as a result of the normalization.

When the target table is a normalized temporal table with transaction time, rows that are deleted as a result of the normalization are closed in transaction time.

### Current Updates

#### NOTICE

CURRENT DML modifications can cause serializability issues for concurrent transactions. See [Potential Concurrency Issues with Current Temporal DML](#) for information on avoiding these issues.

A current update affects only rows that are valid at the current time. These are rows in valid-time or bitemporal tables with a PV that overlaps current time at the time of the update: Rows that have a valid-time period that contains or begins at `TEMPORAL_TIMESTAMP` (or `TEMPORAL_DATE`, depending on the type of the valid-time column).

If additional search conditions are specified in the `UPDATE` statement, they are applied to these current rows. The search condition may specify conditions on both the valid-time and transaction-time columns.

The following types of rows do not qualify for current updates:

- Rows in valid-time or bitemporal tables with a PV that ends before or at current time. These rows are history rows in valid time, so do not qualify for a current update.
- Rows in valid-time or bitemporal tables with a PV that begins after current time. These rows are future rows in valid time, so do not qualify for a current update.
- Rows in bitemporal tables with a transaction-time period that ends before `UNTIL_CLOSED` (before 9999-12-31 23:59:59.999999+00:00). These rows are considered closed in transaction time, and are unavailable to most SQL. They are part of the automatic internal history of changes maintained by the database for tables with transaction time, and do not qualify for a current update.

A current update to a row can result in zero, one, or two additional rows being added to the database, depending on the relationship between the PV of the qualifying row and the current time, and on whether the table is a valid-time or bitemporal table.

## Current Updates to Qualifying Rows in Valid-Time Tables

If the PV of the row contains the current time at the time of the update, the operation modifies one row and inserts a new row into the table:

- The valid-time period for the original row, is set to end at the time of the update (TEMPORAL\_TIMESTAMP or TEMPORAL\_DATE, depending on the type of the valid-time column). No other column values are changed.

This means that the original row is no longer valid. It becomes a history row, showing the original column values that were valid before the update.

- A copy of the original row is inserted which has the new values in the updated columns. The valid-time period of this row is set to begin at the time of the update, and end at the same time as the original row.

If the PV of the row begins at the time of the update:

- The current update operation updates the row.

The PA of a current update begins at TEMPORAL\_TIMESTAMP at the time of the update, and ends at UNTIL\_CHANGED, an indefinite time in the future when the row is changed or deleted. Because this PA matches or contains the PV of the qualified row, the change is valid for the entire PV of the row. New rows do not need to be inserted in the database to account for row states that existed before or after the change.

## Current updates to Qualifying Rows in Bitemporal Tables

Bitemporal tables include both a valid-time column and a transaction-time column. The results of a current update operation on a bitemporal table with respect to the valid-time column are the same as those for a valid-time table. Due to the transaction-time column, every row that is changed as a result of the current update generates an additional row in the database to track the change in transaction time by creating a snapshot of the row prior to the change:

If the PV of the row contains the current time at the time of the update, the operation modifies one row and inserts two new rows into the table:

- The transaction-time period for the original row is set to end at the time of the update (TT\_TIMESTAMP), marking the row as closed in the transaction-time dimension. No other values are changed in the row.

This preserves the original row with the original values that existed before the modification, including the original period of validity. Because the row is closed in transaction-time, it becomes inaccessible to further modifications.

- A copy of the original row is inserted. The row has the valid-time period set to end at the time of the update (TEMPORAL\_TIMESTAMP or TEMPORAL\_DATE, depending on the type of the valid-time column). No other column values are changed.

This means that the original row is no longer valid. It becomes a history row, showing the original column values that were valid before the update.

The value of the transaction-time column is set to (TT\_TIMESTAMP, UNTIL\_CLOSED), as it is for any new row that is inserted to a table having a transaction-time dimension. The row is therefore open in transaction time, and remains accessible as a history row to valid-time SQL.

- A copy of the original row, which has the new values in the updated columns, is inserted.

The valid-time period of this row is set to begin at the time of the update and end at the same time as the original row.

The value of the transaction-time column is set to (TT\_TIMESTAMP, UNTIL\_CLOSED), as it is for any new row that is inserted to a table having a transaction-time dimension. The row is therefore open in transaction time, and remains accessible to valid-time SQL.

If the PV of the row begins at the time of the update, the operation modifies one row and inserts one new row into the table:

- The transaction-time period for the original row is set to end at the time of the update (TT\_TIMESTAMP), marking the row as closed in the transaction-time dimension. No other values are changed in the row.

This preserves the original row with the original values that existed before the modification, including the original period of validity. Because the row is closed in transaction-time, it becomes inaccessible to further modifications.

- The current update operation updates the row.

A copy of the original row is inserted. The row has the new values in the updated columns. The PA of a current update begins at TEMPORAL\_TIMESTAMP at the time of the update, and ends at UNTIL\_CHANGED, an indefinite time in the future when the row is changed or deleted. Because this PA matches or contains the PV of the qualified row, the change is valid for the entire PV of the row. New rows do not need to be inserted in the database to account for row states that existed before or after the change.

The value of the transaction-time column is set to (TT\_TIMESTAMP, UNTIL\_CLOSED), as it is for any new row that is inserted to a table having a transaction-time dimension. The row is therefore open in transaction time, and remains accessible to valid-time SQL.

## Usage Notes

- The value of TEMPORAL\_TIMESTAMP used to stamp the valid-time column is the same for all rows produced as a result of a single update operation.
- The value of TT\_TIMESTAMP used to stamp the transaction-time column is the same for all rows produced as a result of a single update operation.
- If an update to a qualified row does not actually change any column values in a row, temporal operations that close, open, and create new rows are not performed on the row. However, the activity count of the update operation includes these rows, and an update trigger qualifies these rows.
- The modified and inserted rows must not violate any constraints on the table. If there are no uniqueness constraints, inserted rows are not checked for duplicates. If the table has any constraints defined, inserted rows are validated to ensure that the rows do not violate the constraints.

## Sequenced Updates

A sequenced update modifies the specified rows at each point in time that is covered in the period of applicability. That is, rows whose period of validity overlaps the period of applicability are modified for the overlapping portion.

A sequenced update can modify current, history, or future rows in the valid-time dimension depending on the selection time period (period of applicability) specified in UPDATE statement. For bitemporal tables, which include a transaction-time dimension, only rows that are open in transaction time can qualify for the sequenced update.

If the columns modified in the SET clause do not change the values for a row, the row is not changed with any temporal update semantics. The activity count of the update includes the row and an update trigger qualifies such a row but the modification semantics that close, open, or create new rows are not performed.

Only open rows whose period of validity overlaps the period of applicability of the sequenced update are candidates for the update. Optionally, additional qualifications can be placed on the values of valid-time and transaction-time columns to further filter the rows that will qualify for the update.

A sequenced update of a row in a valid-time table results in the modification of the old row and, potentially, the insertion of new rows, depending on the relationship between the period of validity of the row and the period of applicability of the update.

A sequenced update of a row in a bitemporal table first closes out the old row in transaction time. A copy of the row is made, open in transaction time, and modifications are made appropriate to the update syntax. These modifications can include simply updating the row, or splitting the row into two or three rows, depending on the relationship between the period of validity of the row and the period of applicability specified by the update statement.

The following table describes the sequenced update operation in a table with valid time.

| IF the period of applicability of the update ...    | THEN ...                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-----------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| is contained within the period of validity of a row | <p>the row qualifies for the update, but only during the portion of the period of validity that overlaps the period of applicability of the update. Because the row values are updated for only a portion of the original row period of validity, the original row values remain valid before and after the period of applicability of the update. Therefore, the update operation results in three rows:</p> <ul style="list-style-type: none"> <li>• The original row that qualified for the update is modified to have the valid time period end at the beginning of the period of applicability of the update.</li> <li>• A new row is inserted with the updated values. Its valid time period reflects the entire period of applicability of the update.</li> <li>• A new row is inserted with the same values as the original row that qualified for the update. However, the valid-time period is set to reflect the portion of the original row valid-time period that remains after the period of applicability of the update. The valid-time period begins at</li> </ul> |

| IF the period of applicability of the update ...                                                                                                                  | THEN ...                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                                                                                                                   | the end of the update period of applicability, and ends at the time the original row valid-time period ended.                                                                                                                                                                                                                                                                                                                                                    |
| overlaps the period of validity such that the beginning bound of the period of applicability is between the beginning and ending bounds of the period of validity | the update results in one new row and one old existing row: <ul style="list-style-type: none"> <li>• The new row contains the modified columns with its period of validity set to the portion that is common between the period of applicability and the period of validity.</li> <li>• The period of validity of the existing row is set to the portion of the period of validity that exists before the beginning of the period of applicability.</li> </ul>   |
| overlaps the period of validity such that the ending bound of the period of applicability is between the beginning and ending bounds of the period of validity    | the update results in one new row and one old existing row: <ul style="list-style-type: none"> <li>• The new row contains the modified columns and the period of validity is set to the portion that is common between the period of applicability and the period of validity.</li> <li>• The period of validity of the old existing row is set to the portion of the period of validity that exists after the ending of the period of applicability.</li> </ul> |
| contains the period of validity, including the case where the period of applicability equals the period of validity                                               | the existing row is updated for the specified columns. There is no change in the period of validity.                                                                                                                                                                                                                                                                                                                                                             |

Temporal constraints that are defined on a table being updated apply to both the existing row that is modified and to the rows that are newly inserted.

## Nonsequenced Updates

A nonsequenced update modifies the specified rows across all states or any state. A nonsequenced update ignores valid-time semantics when updating a row of a table with valid time.

A nonsequenced update operates on only open rows for a table with transaction time. A nonsequenced update treats a valid-time column like a regular column for a table with valid time. For a valid-time table, a nonsequenced update does not create multiple rows like in a current or sequenced update.

For a table with transaction time, a nonsequenced update of a row first closes out the existing qualified row and inserts a new row with the updated columns only when the column values change. If there are no changes made to the row, the existing row is not closed. For a valid-time table, a nonsequenced update modifies the existing qualified row like a regular update.

Because a nonsequenced update permits updates to the valid-time column like a conventional Period column, the valid-time column can be used in the assignment list.

All modifications on a transaction-time table or a bitemporal table cause changes to be recorded, regardless of whether the modifications are in the same transaction.

A nonsequenced update that joins two or more tables is like a regular join. The valid-time column may participate in the join like a regular column.

## Nontemporal Updates

### Note:

Rows that are closed in transaction time provide a history of all modifications and deletions on tables that have a transaction-time column. The automatic history that tables with transaction time provide can be used for regulatory compliance auditing, so these rows are generally inaccessible to DML modifications. Because NONTEMPORAL DML statements can modify closed rows, the special NONTEMPORAL privilege is required. For more information on the NONTEMPORAL privilege, see [Usage Notes](#).

A nontemporal update is similar to a conventional update, but the transaction-time column is treated as any other column in the table. The transaction-time column values can be explicitly specified in the SET clause of the statement. A nontemporal update can be issued to update closed or open rows.

The qualification condition in the UPDATE statement considers both the open and closed rows in the table. Additionally, for a table with valid time, both valid and no-longer-valid rows participate in the update. If the statement references multiple temporal tables, a nonsequenced form of join is performed on the tables.

If the transaction-time column is modified, the following rules apply:

- The beginning bound must not be greater than the system time at the time of the update.
- The ending bound must be UNTIL\_CLOSED or less than or equal to the system time at the time of the update.

## Related Information

| For more information on... | See...                                                                |
|----------------------------|-----------------------------------------------------------------------|
| UPDATE statement           | <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146 |

## UPDATE (Temporal Upsert Form)

Updates column values in a specified row or, if the row does not exist, inserts it into the table with a specified set of initial column values.

## Required Privileges

UPDATE (Temporal Upsert Form) requires the same privileges as conventional UPDATE (Upsert Form).

## UPDATE Syntax (Temporal Upsert Form)

### Note:

*Teradata Vantage™ - Temporal Table Support* describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
[{ CURRENT | NONSEQUENCED } VALIDTIME |
 [SEQUENCED] VALIDTIME [period_expression] |
 NONTEMPORAL
]
upsert_statement [;]
```

### Note:

To ensure application portability to ANSI standards for temporal SQL, Teradata recommends explicit specification of all temporal qualifiers.

## UPDATE Syntax Elements (Temporal Upsert Form)

### CURRENT VALIDTIME

Specifies that the upsert is current in the valid-time dimension.

The CURRENT VALIDTIME qualifier is used to qualify rows from the referenced tables in the valid-time dimension. In the transaction-time dimension, open rows qualify.

If the session temporal qualifier is not set and the temporal qualifier is omitted from the upsert statement, the default qualifier is CURRENT VALIDTIME.

For a table that supports transaction time, the temporal qualifier in the transaction-time dimension is CURRENT TRANSACTIONTIME.

CURRENT DML modifications can cause serializability issues for concurrent transactions. See [Potential Concurrency Issues with Current Temporal DML](#) for information on avoiding these issues.

### VALIDTIME and SEQUENCED VALIDTIME

Specifies that the upsert is sequenced in the valid-time dimension if the target table supports valid time.

The target table must have a valid-time column.

A sequenced upsert on a target table that supports valid time sets the valid-time value to *period\_expression*, or, if omitted, to PERIOD'(0001-01-01, UNTIL\_CHANGED)' for a PERIOD(DATE) valid-time column or PERIOD '(0001-01-01 00:00:00.000000+00:00, UNTIL\_CHANGED)' for a PERIOD(TIMESTAMP) valid-time column.

For a table that supports transaction time, the temporal qualifier in the transaction-time dimension is CURRENT TRANSACTIONTIME.

### ***period\_expression***

Specifies the period of applicability for the DML statement.

The period of applicability must be a period constant expression that does not reference any columns, but can reference parameterized values and the TEMPORAL\_DATE or TEMPORAL\_TIMESTAMP built-in functions.

The period of applicability can also be a self-contained noncorrelated scalar subquery that is always nonsequenced in the time dimensions regardless of the temporal qualifier for the DML statement.

---

#### **Note:**

If a *period\_expression* is specified, the valid-time column cannot be specified or referenced anywhere in the query. If the valid-time column is a derived period column, the component columns cannot be specified or referenced anywhere in the query.

---

If *period\_expression* is omitted, the period of applicability defaults to PERIOD'(0001-01-01, UNTIL\_CHANGED)' for a PERIOD(DATE) valid-time column or PERIOD '(0001-01-01 00:00:00.000000+00:00, UNTIL\_CHANGED)' for a PERIOD(TIMESTAMP( *n* ) WITH TIME ZONE) valid-time column, where precision *n* and WITH TIME ZONE are optional.

### **NONSEQUENCED VALIDTIME**

Specifies that the upsert is nonsequenced in the valid-time dimension if the target table supports valid time.

The target table must have a valid-time column.

For a table that supports transaction time, the temporal qualifier in the transaction-time dimension is CURRENT TRANSACTIONTIME.

### ***upsert\_statement***

Specifies existing UPDATE (Upsert Form) statement syntax.

For details, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

The following restrictions apply to the SET clause of the UPDATE portion of *upsert\_statement*:

- For a current or sequenced update on a table with valid time, the SET clause cannot reference the valid-time column as the name of the column whose data is to be updated.
- For an update on a table with transaction time, the SET clause cannot reference the transaction-time column as the name of the column whose data is to be updated.
- The SET clause cannot use `CURRENT_DATE` or `CURRENT_TIMESTAMP` in the expressions that are used to assign a value to the valid-time column.
- For a row-partitioned table, the SET clause cannot reference a partitioning column

## Usage Notes

### General

The UPDATE portion of upsert follows the semantics described for the temporal form of UPDATE. The INSERT portion of upsert follows the semantics described for the temporal form of INSERT. All restrictions that apply to the conventional form of upsert also apply to the temporal form of upsert, with the exception of the relaxed restrictions documented in the following topics.

If the upsert does not require any temporal operations, such as opening or closing a row, use a `NONSEQUENCED VALIDTIME UPSERT`. The SET clause can reference the valid-time column to set the valid-time value.

All check, primary key, and temporal unique (current, sequenced, nonsequenced) constraints defined on the table are checked only on rows that are open in transaction time.

DML operations on tables defined with `NORMALIZE` produce a normalized set of modified rows. Some unmodified rows may be deleted from the target table as a result of the normalization.

When the target table is a normalized temporal table with transaction time, rows that are deleted as a result of the normalization are closed in transaction time.

### Qualifying a Row for Upsert

In addition to the conditions specified in the UPDATE statement, the test for qualification of a row is also based on the overlap of the valid-time column value with the period of applicability. For a sequenced upsert, the row is considered for the update even if the period of validity is contained in the period of applicability; no row is inserted for the extra period of applicability.

The UPDATE portion of the upsert must qualify a single row.

When qualifying a row for a nonsequenced upsert, all of the restrictions that apply to the conventional form of upsert apply in the valid-time and transaction-time dimensions.

The following rules apply to temporal tables that have a partitioned primary index:

- The UPDATE portion of a current upsert must qualify a single row from a single current partition.

- The UPDATE portion of a sequenced upsert with an optional period of applicability and a matching PI value must also qualify a single row from single partition.
- For a current or sequenced upsert, when partitioning is defined on the valid-time column, transaction-time column, or valid-time and transaction-time columns, the upsert can omit the equality condition on the valid-time and transaction-time columns. The equality condition can be specified on the bound functions that were used in the partitioning expression. The conditions IS UNTIL\_CHANGED and IS UNTIL\_CLOSED are also considered equality conditions on the END bound function (these conditions are valid only on the END bound function).

**Note:**

Upserts are not supported for tables that do not have primary indexes or for column-partitioned tables.

## Modification of a Row for Upsert

A current upsert results in a current update if the qualified current row is found; otherwise, it results in a current insert. For a row-partitioned table, a current update must not result in rows moving to different partitions.

The insert portion of a current upsert can specify values for the valid-time or transaction-time column and the insert must be to the same partition as referenced by the update portion.

In a sequenced upsert, if the qualified row overlaps with the specified (or default) period of applicability, a sequenced form of update is performed on that row; otherwise, a sequenced form of insert is performed. For a row-partitioned table, the sequenced form of update must not result in rows moving to different partitions.

The insert portion of a sequenced upsert can specify values for the valid-time or transaction-time column and the insert must be to the same partition as referenced by the update portion. If the insert portion of a sequenced upsert specifies a value for the valid-time column, the specified period of applicability in the temporal qualifier is ignored for the insert portion of the upsert.

For a nonsequenced upsert, a nonsequenced update is performed if a row is found; otherwise, a nonsequenced insert is performed.

## Related Information

| For more information on...     | See...                                                                |
|--------------------------------|-----------------------------------------------------------------------|
| UPDATE statement (upsert form) | <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146 |

## Cursors and Temporal Queries

A cursor is a data structure that stored procedures and Preprocessor2 use at runtime to point to the result rows in a response set returned by an SQL query. Procedures and embedded SQL also use cursors to manage inserts, updates, execution of multistatement requests, and SQL macros.

The DML semantics described for temporal tables apply to DML associated with a cursor, with some limitations that relate to positioned (updatable) cursors:

- Only the current form of positioned cursors is allowed. The SELECT statement specified in the iteration statement FOR, DECLARE CURSOR, and positioned DELETE and UPDATE statements must all be qualified as CURRENT.
- A SELECT statement on a temporal table can open an updatable cursor if the statement conforms to all existing rules on updatable cursors.
- A SELECT statement that opens an updatable cursor has the same syntax as described in [SELECT/SELECT INTO \(Temporal Forms\)](#) when the statement references a temporal table, but the FROM clause must not specify an AS OF clause in the transaction-time dimension.
- A positioned DELETE must be a CURRENT delete, and must not specify an AS OF clause as part of the FROM clause.
- A positioned UPDATE must be a CURRENT update.

### Related Information

| For more information on... | See...                                                                        |
|----------------------------|-------------------------------------------------------------------------------|
| cursors                    | <i>Teradata Vantage™ - SQL Stored Procedures and Embedded SQL</i> , B035-1148 |

# SQL Data Control Language (Temporal Forms)

This section describes the GRANT and REVOKE statements for temporal tables.

This material covers the syntax, rules, and other details that are specific to temporal table support.

All existing rules that apply to the corresponding non-temporal DCL statements also apply to the statements here. For more information about these rules, see *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

## GRANT (Temporal Form)

Assigns the NONTEMPORAL and other explicit privileges on a database, user, table, or view to a user or group of users.

## GRANT Syntax (Temporal Form)

### Note:

*Teradata Vantage™ - Temporal Table Support* describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
GRANT privilege_spec ON item TO
 { user_spec [,...] | PUBLIC }
 [WITH GRANT OPTION] [;]
```

### *privilege\_spec*

```
{ ALL [PRIVILEGES] |
 [ALL BUT] { NONTEMPORAL | privilege } [,...]
}
```

### *item*

```
{ database_name |
 user_name |
```

```
[database_name. | user_name.] { table_name | view_name }
}
```

***user\_spec***

```
[ALL] user_name
```

## GRANT Syntax Elements (Temporal Form)

**ALL [PRIVILEGES]**

Specifies that the specified user is to receive all privileges that can be granted on the specified object.

GRANT ALL includes the NONTEMPORAL privilege if the InclNTforGrntOrRevokAll field of the DBS Control utility is set to TRUE. Otherwise, GRANT ALL excludes the NONTEMPORAL privilege.

All of the privileges, including the NONTEMPORAL privilege, that the grantor has on the object are granted if a user has them with WITH GRANT OPTION.

**NONTEMPORAL**

Specifies that the user can use the NONTEMPORAL prefix to perform nontemporal operations on transaction-time and bitemporal tables contained in *database\_name* or *user\_name*, on *table\_name*, or on *view\_name*.

***privilege***

Specifies a privilege other than NONTEMPORAL.

For details, see *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

**ALL BUT**

Specifies that the named user is to receive all privileges that can be granted on the specified object except for those specified in the privilege list. As in ALL, only those object privileges owned by the grantor WITH GRANT OPTION are granted.

**ON *database\_name***

Specifies the name of a database that contains or may contain transaction-time or bitemporal tables or both.

**ON *user\_name***

Specifies the name of a user that contains or may contain transaction-time tables, bitemporal tables, or both.

**ON *table\_name***

Specifies the name of a transaction-time or bitemporal table.

**ON *view\_name***

Specifies the name of an updatable view created on a transaction-time or bitemporal table.

**TO [ALL] *username***

Specifies the name of an existing database or user that identifies the recipient.

If you specify ALL, then the object privileges are granted to the named database or user and to every database or user owned by that database or user now and in the future.

**WITH GRANT OPTION**

Specifies that the grantee receives the granted privileges WITH GRANT OPTION.

## Usage Notes

The NONTEMPORAL privilege is required to use the NONTEMPORAL prefix with ALTER TABLE, CREATE TABLE AS, DELETE, INSERT, and UPDATE statements.

These nontemporal operations allow modifications to closed rows, which are normally prohibited on tables that have transaction time. For example, they allow direct modification of the transaction time column values, even for rows that are closed in transaction time. They also allow physical deletion of rows that are closed in transaction time. Closed rows are normally inaccessible to modifications. They provide a history of prior row modifications and deletions, and are saved indefinitely in the table.

Because nontemporal operations can be used to circumvent the normal processing of tables with transaction time, the use of nontemporal operations is discouraged. However, nontemporal operations may be required for certain kinds of database maintenance, as when very old history rows must be archived and deleted for space considerations, and for correcting problems or corruptions in temporal tables.

The availability of nontemporal operations at the system level is controlled by the NONTEMPORAL privilege and by the EnabNonTempoOp setting in DBS Control. By default, nontemporal operations are not allowed, regardless of the NONTEMPORAL privilege. To enable nontemporal operations for users with the NONTEMPORAL privilege, use the DBS Control utility. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

A NONTEMPORAL privilege is a table and view level privilege and can also be granted at the database or user level. This privilege is not an automatic privilege and must be explicitly granted by user DBC or another user who has sufficient privileges.

The NONTEMPORAL privilege is granted implicitly to DBC with GRANT OPTION. The system does not grant the NONTEMPORAL privilege implicitly to any other users or databases.

The NONTEMPORAL privilege follows the system convention that it can be granted by an owner on any owned objects. Note that an owner must explicitly grant the NONTEMPORAL privilege to itself in order to use the NONTEMPORAL qualifier on owned objects.

For example, assume user john123 has created a table, MyTemporalTable, and is logged in to the database. In order to perform nontemporal operations to this table, john123 would first need to send the following request:

```
GRANT NONTEMPORAL ON mytemporaltable TO john123;
```

The privilege is recorded as 'NT' in the AccessRight column in the DBC.AccessRights table.

Any use of the NONTEMPORAL privilege, whether successful or denied by the system, is automatically logged in the access logging tables. Nontemporal logging requires no explicit BEGIN LOGGING statement, and cannot be disabled.

For statements that include the NONTEMPORAL prefix, the system checks the NONTEMPORAL privilege in addition to those that the statement normally requires. For example, to execute NONTEMPORAL DELETE requires both NONTEMPORAL and DELETE privileges.

## Related Information

| For more information on... | See...                                                            |
|----------------------------|-------------------------------------------------------------------|
| GRANT (regular form)       | <i>Teradata Vantage™ - SQL Data Control Language</i> , B035-1149. |
| REVOKE (temporal form)     | <a href="#">REVOKE (Temporal Form)</a> .                          |
| DBS Control utility        | <i>Teradata Vantage™ - Database Utilities</i> , B035-1102.        |
| DELETE (temporal form)     | <a href="#">DELETE (Temporal Form)</a>                            |
| INSERT (temporal form)     | <a href="#">INSERT/INSERT SELECT (Temporal Forms)</a>             |
| Nontemporal operations     | <a href="#">Nontemporal Operations</a>                            |
| UPDATE (temporal form)     | <a href="#">UPDATE (Temporal Form)</a>                            |

## REVOKE (Temporal Form)

Revokes the NONTEMPORAL and other explicit privileges on a database, user, table, or view.

## REVOKE Syntax (Temporal Form)

### Note:

*Teradata Vantage™ - Temporal Table Support* describes syntax that is especially relevant to temporal tables. Syntax that is not required, or that is not otherwise specific to temporal tables is generally not shown in this document. For additional syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 , *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 , and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

```
REVOKE [GRANT OPTION FOR] privilege_spec ON item
 { TO | FROM } user_spec [...] [;]
```

### *privilege\_spec*

```
{ ALL [PRIVILEGES] |
 [ALL BUT] { NONTEMPORAL | privilege } [,...] |
 NONTEMPORAL |
 privilege
}
```

### *item*

```
{ database_name |
 user_name |
 [database_name. | user_name.] { table_name | view_name }
}
```

### *user\_spec*

```
[ALL] user_name
```

## REVOKE Syntax Elements (Temporal Form)

### GRANT OPTION FOR

Specifies that only the GRANT authority is removed from the specified privileges for the specified grantees for the corresponding explicit privileges they have.

**ALL [PRIVILEGES]**

Specifies to revoke from the specified user all explicitly granted privileges that can be granted on the specified object, and that are held, either implicitly or explicitly, WITH GRANT OPTION by the user executing the REVOKE.

REVOKE ALL includes the NONTEMPORAL privilege if the `IncIntForGrntOrRevokAll` field of the DBS Control utility is set to TRUE. Otherwise, REVOKE ALL excludes the NONTEMPORAL privilege.

**ALL BUT**

Specifies to revoke all explicit database privileges from the specified user, except those listed, that can be granted on the specified object and that are held, either implicitly or explicitly, WITH GRANT OPTION by the user performing the REVOKE statement.

**NONTEMPORAL**

Specifies that the user cannot use the NONTEMPORAL prefix to perform nontemporal operations on *table\_name* or *view\_name* or on any transaction-time tables, bitemporal tables, or updatable views contained in *database\_name* or *user\_name*.

For details on the NONTEMPORAL privilege, see [Usage Notes](#).

***privilege***

Specifies a privilege other than NONTEMPORAL.

For details, see the REVOKE statement in *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

**ON *database\_name***

Specifies the name of a database that contains or may contain transaction-time tables, bitemporal tables, or both.

**ON *user\_name***

Specifies the name of a user that contains or may contain transaction-time tables, bitemporal tables, or both.

**ON *table\_name***

Specifies the name of a bitemporal or transaction-time table.

**ON *view\_name***

Specifies the name of an updatable view created on a bitemporal table or transaction-time table.

**TO [ALL] *username***

**FROM [ALL] *username***

Specifies the name of an existing database or user that identifies the recipient.

If you specify ALL, then the object privileges are revoked from the named database or user and from every database or user owned by that database or user.

## Related Information

| For more information on... | See...                                                            |
|----------------------------|-------------------------------------------------------------------|
| REVOKE (regular form)      | <i>Teradata Vantage™ - SQL Data Control Language</i> , B035-1149. |
| GRANT (temporal form)      | <a href="#">GRANT (Temporal Form)</a> .                           |
| DBS Control utility        | <i>Teradata Vantage™ - Database Utilities</i> , B035-1102.        |
| DELETE (temporal form)     | <a href="#">DELETE (Temporal Form)</a> .                          |
| INSERT (temporal form)     | <a href="#">INSERT/INSERT SELECT (Temporal Forms)</a>             |
| Nontemporal operations     | <a href="#">Nontemporal Operations</a>                            |
| UPDATE (temporal form)     | <a href="#">UPDATE (Temporal Form)</a>                            |

# Administration

This section provides information about administration of temporal table support.

## System Clocks

Temporal table support depends on all system nodes running network time protocol (NTP). NTP keeps the system node clocks synchronized within 100 milliseconds. If NTP is unavailable on the system, or is not running on any system node, temporal table support is disabled.

---

**Note:**

NTP is not required on SMP systems.

---

To install the teradata-ntp package:

1. Go to <https://support.teradata.com>.
2. Log in.

To configure NTP, see Knowledge Article KAP1A9C72, also available at <https://support.teradata.com>.

Vantage can manage the small differences between node clocks due to the minute drift that NTP allows. In the unlikely event that an update is made to an existing row such that the beginning of a time period is after the end time, the transaction is automatically aborted.

Teradata recommends synchronizing the system to an external master time source, such as a Web-based or government sponsored standard time service.

### NOTICE

Do not make manual changes to individual node clocks. Circumventing NTP, as by the use of operating system commands to directly change the current time on a node, will compromise temporal table support, and could result in incorrect data or a loss of data.

When new nodes are added to the system, the database administrator must manually set the initial time on those nodes. The time must closely match the time on the other nodes in the system. If a node must be replaced, the initial time set on the replacement must be greater (later) than the time the previous node went down.

## Nontemporal Operations

To allow operations that are otherwise not allowed on tables with transaction time, statements can specify the NONTEMPORAL prefix. Such operations on tables with transaction time require the NONTEMPORAL privilege.

Here is an example that grants the NONTEMPORAL privilege to user simon573 on a transaction-time table named Policy\_Types:

```
GRANT NONTEMPORAL ON Policy_Types TO simon573;
```

To disable all nontemporal operations, use the EnabNonTempoOp field of the DBS Control utility.

For details on the DBS Control utility, see *Teradata Vantage™ - Database Utilities*, B035-1102.

## Capacity Planning for Temporal Tables

Temporal tables typically contain more rows than otherwise equivalent nontemporal tables. This is due to the way rows are automatically added to temporal tables as a result of most kinds of modifications. Furthermore, under normal conditions, “deleted” rows are not truly physically deleted from temporal tables that have a transaction time dimension.

For the same reasons, temporal tables can grow faster than nontemporal tables, depending on how frequently they are modified, and on the nature of those modifications. Tables with transaction-time columns grow monotonically, never shrinking, because rows are never physically deleted from these tables.

The following tables show how temporal tables that have a transaction-time column can grow depending on the nature and frequency of table modifications. Valid-time tables are likely to experience less growth, because rows in valid-time tables can be physically deleted from the tables.

Use these examples to estimate annual growth of temporal tables for capacity planning.

Table and Row Size Calculation forms are available in *Teradata Vantage™ - Database Design*, B035-1094 for nontemporal tables.

- Sequenced modifications are typically historical in nature.
- To reflect conservative estimates, the table size increase calculation is based on the maximum number of rows that can be produced by each type of modification.

### Example: Capacity planning for lightly modified temporal tables

|                                                     | Transaction-time Table | Bitemporal Table   | Bitemporal Table       |
|-----------------------------------------------------|------------------------|--------------------|------------------------|
| Table Size Before Modifications (rows)              | 100                    | 100                | 100                    |
| Modification Type                                   | Current                | Current            | Sequenced (historical) |
| Modifications per Year (percent of rows)            | 10% (0.19% weekly)     | 10% (0.19% weekly) | 10% (0.19% weekly)     |
| Number of Additional Rows Produced per Modification | 1                      | 1 or 2             | 1, 2, or 3             |

|                                               | Transaction-time Table | Bitemporal Table | Bitemporal Table |
|-----------------------------------------------|------------------------|------------------|------------------|
| Largest Table Size After Modifications (rows) | 110                    | 120              | 130              |
| Annual Increase in Table Size                 | 10%                    | 20%              | 30%              |

## Example: Capacity planning for moderately modified temporal tables

|                                                     | Transaction-time Table | Bitemporal Table   | Bitemporal Table       |
|-----------------------------------------------------|------------------------|--------------------|------------------------|
| Table Size Before Modifications (rows)              | 100                    | 100                | 100                    |
| Modification Type                                   | Current                | Current            | Sequenced (historical) |
| Modifications per Year (percent of rows)            | 30% (0.58% weekly)     | 30% (0.58% weekly) | 30% (0.58% weekly)     |
| Number of Additional Rows Produced per Modification | 1                      | 1 or 2             | 1, 2, or 3             |
| Largest Table Size After Modifications (rows)       | 130                    | 160                | 190                    |
| Annual Increase in Table Size                       | 30%                    | 60%                | 90%                    |

## Example: Capacity planning for heavily modified temporal tables

|                                                     | Transaction-time Table | Bitemporal Table   | Bitemporal Table       |
|-----------------------------------------------------|------------------------|--------------------|------------------------|
| Table Size Before Modifications (rows)              | 100                    | 100                | 100                    |
| Modification Type                                   | Current                | Current            | Sequenced (historical) |
| Modifications per Year (percent of rows)            | 50% (0.96% weekly)     | 50% (0.96% weekly) | 50% (0.96% weekly)     |
| Number of Additional Rows Produced per Modification | 1                      | 1 or 2             | 1, 2, or 3             |
| Largest Table Size After Modifications (rows)       | 150                    | 200                | 250                    |
| Annual Increase in Table Size                       | 50%                    | 100%               | 150%                   |

## Archiving Temporal Tables

The Archive/Recovery utility can be used to archive and restore all types of temporal tables. Archive operations on temporal tables use the same syntax as nontemporal tables. For temporal tables, the ARCHIVE, RESTORE, and COPY commands can operate on:

- Entire temporal tables

Archiving an entire temporal table saves all rows to the archive, including history, current, future, open, and closed rows.

- Specified partitions of a temporal table

An archive operation can be limited to specified partitions of row-partitioned temporal tables with primary indexes, provided those tables are not also column partitioned. For example, a bitemporal table that is partitioned using the recommended partitioning expression has rows separated into the following partitions:

- open rows with valid-time periods that are current and future
- open rows with valid-time periods that are history
- closed rows

The archive can be limited to store only the current and history open rows from the first partition.

Use the following guidelines when archiving temporal tables that have been partitioned into current and history rows:

- History rows are automatically formed in partitions containing current and future rows, and in partitions containing open rows when current, open rows are modified. The ALTER TABLE TO CURRENT statement repartitions the table, moving history rows out of the current partition.

If archiving the current and future rows, ensure the current partition includes only current and future rows by issuing an ALTER TABLE TO CURRENT statement on the table immediately prior to the ARCHIVE operation.

If restoring only current and future rows to an existing temporal table, issue an ALTER TABLE TO CURRENT statement on the table immediately prior to the restore operation.

- Archive the complete partition that isolates the open current and future rows, or archive the entire table. Do not archive a history partition alone, or a subset of the partition for open current and future rows.
- RESTORE the entire temporal archive. Never restore only a portion of the archive.
- If only the current partition is restored for a temporal table, the existing history partition for that table is deleted, and a new history is begun starting from the time of the restore. This loses historical information from the existing table.

Teradata recommends a dual archive strategy for temporal tables. Save entire temporal tables in one archive, and the current temporal partitions in a separate archive. The archive containing entire tables can be used to restore temporal tables including history information. The archive containing current partitions can be used for disaster recovery, when restoring history rows is not desired.

## Related Information

| For information on...        | See...                                                                                                                                                                                                             |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Archive/Recovery Utility     | <i>Teradata® Archive/Recovery Utility Reference</i> , B035-2412                                                                                                                                                    |
| Partitioning temporal tables | <a href="#">Row Partitioning Temporal Tables</a>                                                                                                                                                                   |
| ALTER TABLE TO CURRENT       | <ul style="list-style-type: none"> <li>• <a href="#">Partitioning Expressions for Temporal Tables</a></li> <li>• <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144</li> </ul> |

# How to Read Syntax

This document uses the following syntax conventions.

| Syntax Convention | Meaning                                                                                                                                                                                                                                                                                                                                                                  |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| KEYWORD           | Keyword. Spell exactly as shown.<br>Many environments are case-insensitive. Syntax shows keywords in uppercase unless operating system restrictions require them to be lowercase or mixed-case.                                                                                                                                                                          |
| <i>variable</i>   | Variable. Replace with actual value.                                                                                                                                                                                                                                                                                                                                     |
| <i>number</i>     | String of one or more digits. Do not use commas in numbers with more than three digits.<br>Example: 10045                                                                                                                                                                                                                                                                |
| [ x ]             | x is optional.                                                                                                                                                                                                                                                                                                                                                           |
| [ x   y ]         | You can specify x, y, or nothing.                                                                                                                                                                                                                                                                                                                                        |
| { x   y }         | You must specify either x or y.                                                                                                                                                                                                                                                                                                                                          |
| x [...]           | You can repeat x, separating occurrences with spaces.<br>Example: x x x<br>See note after table.                                                                                                                                                                                                                                                                         |
| x [, ...]         | You can repeat x, separating occurrences with commas.<br>Example: x, x, x<br>See note after table.                                                                                                                                                                                                                                                                       |
| x [delimiter...]  | You can repeat x, separating occurrences with specified delimiter.<br>Examples:<br><ul style="list-style-type: none"> <li>If <i>delimiter</i> is semicolon:<br/>x; x; x</li> <li>If <i>delimiter</i> is { ,   OR }, you can do either of the following: <ul style="list-style-type: none"> <li>x, x, x</li> <li>x OR x OR x</li> </ul> </li> </ul> See note after table. |

---

**Note:**

You can repeat only the immediately preceding item. For example, if the syntax is:

```
KEYWORD x [...]
```

You can repeat x. Do not repeat KEYWORD.

If there is no white space between x and the delimiter, the repeatable item is x and the delimiter. For example, if the syntax is:

```
[x, [...]] y
```

- You can omit x: y
  - You can specify x once: x, y
  - You can repeat x and the delimiter: x, x, x, y
-

## Examples

This section provides examples of various operations that involve temporal tables.

Some of the examples use the following tables:

```
CREATE MULTISET TABLE Policy(
 Policy_ID INTEGER,
 Customer_ID INTEGER,
 Policy_Type CHAR(2) NOT NULL,
 Policy_Details CHAR(40),
 Policy_Term PERIOD(DATE) NOT NULL AS VALIDTIME
)
PRIMARY INDEX(Policy_ID);

CREATE MULTISET TABLE Policy_Types (
 Policy_Name VARCHAR(20),
 Policy_Type CHAR(2) NOT NULL PRIMARY KEY,
 Policy_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
 AS TRANSACTIONTIME
)
PRIMARY INDEX (Policy_Name);

CREATE MULTISET TABLE Policy_History(
 Policy_ID INTEGER,
 Customer_ID INTEGER,
 Policy_Type CHAR(2) NOT NULL,
 Policy_Details CHAR(40),
 Validity PERIOD(DATE) NOT NULL AS VALIDTIME,
 Policy_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
 AS TRANSACTIONTIME
)
PRIMARY INDEX(Policy_ID);
```

Policy is a valid-time table, Policy\_Types is a transaction-time table, and Policy\_History is a bitemporal table. These tables are simplified forms of tables that might be used by an insurance application.

## Creating Temporal Tables

For examples of creating temporal tables, see [Creating Temporal Tables](#).

## Querying Temporal Tables

The following examples demonstrate the basic kinds of temporal queries.

### Example: Current Query on a Valid-Time Table

To query a valid-time table for the rows that are valid at the current time (rows that overlap with current time), use the `CURRENT VALIDTIME` temporal qualifier in the `SELECT` statement. For example:

```
CURRENT VALIDTIME
SELECT *
FROM Policy
WHERE Policy_Type = 'AU';
```

The result is a nontemporal result set. (The result set does not include the valid-time column.)

| Policy_ID | Customer_ID | Policy_Type | Policy_Details    |
|-----------|-------------|-------------|-------------------|
| -----     | -----       | -----       | -----             |
| 541077    | 766492008   | AU          | STD-CH-344-YXY-00 |
| 541145    | 616035020   | AU          | STD-CH-348-YXN-00 |
| 541008    | 246824626   | AU          | STD-CH-345-NXY-00 |

### Example: Current Query on a Transaction-Time Table

To query a transaction-time table for the current rows, use the `CURRENT TRANSACTIONTIME` temporal qualifier in the `SELECT` statement. For example:

```
CURRENT TRANSACTIONTIME
SELECT *
FROM Policy_Types;
```

The result is a nontemporal result set. (The result set does not include the transaction-time column.)

| Policy_Name        | Policy_Type |
|--------------------|-------------|
| -----              | -----       |
| Premium Automobile | AP          |
| Basic Homeowner    | HM          |
| Basic Automobile   | AU          |
| Premium Homeowner  | HP          |

## Example: Sequenced Query on a Valid-Time Table

To query a valid-time table for the rows that are valid at a specific time period, use the SEQUENCED VALIDTIME temporal qualifier in the SELECT statement. For example:

```
SEQUENCED VALIDTIME PERIOD '(2009-01-01, 2009-12-31)'
SELECT Policy_ID, Customer_ID
FROM Policy
WHERE Policy_Type = 'AU';
```

The result set is a temporal table that includes rows that are valid for a period of applicability specified by PERIOD '(2009-01-01, 2009-12-31)':

| Policy_ID | Customer_ID | VALIDTIME                |
|-----------|-------------|--------------------------|
| -----     | -----       | -----                    |
| 541077    | 766492008   | ('09/12/21', '09/12/31') |
| 541145    | 616035020   | ('09/12/03', '09/12/31') |
| 541008    | 246824626   | ('09/10/01', '09/12/31') |

### Note:

The valid-time column for the result set, VALIDTIME, which is automatically appended to the results of a sequenced valid-time query, is different from the valid-time column (Policy\_Term) of the temporal table that was queried. The valid time for the results of the query is the intersection of the PA of the query and the original valid-time periods of the rows.

## Example: Nonsequenced Query on a Valid-Time Table

To query a valid-time table such that no special temporal treatment is given to the valid-time column, use the NONSEQUENCED VALIDTIME temporal qualifier in the SELECT statement. For example:

```
NONSEQUENCED VALIDTIME SELECT * FROM Policy;
```

| Policy_ID | Customer_ID | Policy_Type | Policy_Details    | Policy_Term              |
|-----------|-------------|-------------|-------------------|--------------------------|
| -----     | -----       | -----       |                   |                          |
| 541008    | 246824626   | AP          | STD-CH-345-NXY-00 | ('09/10/01', '99/12/31') |
| 540944    | 123344567   | HM          | STD-PL-332-YXY-00 | ('07/02/03', '99/12/31') |
| 497201    | 304779902   | AU          | STD-CH-524-WXY-00 | ('05/02/14', '06/02/14') |
| 541145    | 616035020   | B           | STD-CH-348-YXN-00 | ('09/12/03', '10/12/01') |
| 497201    | 304779902   | AU          | STD-CH-524-WXY-01 | ('06/02/14', '07/02/14') |

|        |           |    |                   |                          |
|--------|-----------|----|-------------------|--------------------------|
| 497201 | 304779902 | AU | STD-CH-524-WXY-02 | ('07/02/14', '99/12/31') |
| 541077 | 766492008 | HP | STD-CH-344-YXY-00 | ('09/12/21', '99/12/31') |

The valid-time column, `Policy_Term`, is treated as a regular column with a period data type, and appears in the result set. All rows in the table are returned, even if they are history rows.

## Example: Nonsequenced Query on a Transaction-Time Table

To query a transaction-time table such that no special semantics are placed on the transaction-time column, use the `NONSEQUENCED TRANSACTIONTIME` temporal qualifier in the `SELECT` statement. For example:

```
NONSEQUENCED TRANSACTIONTIME SELECT * FROM Policy_Types;
```

| Policy_Name        | Policy_Type | Policy_Duration                                                          |
|--------------------|-------------|--------------------------------------------------------------------------|
| Premium Automobile | AP          | ('2012-06-19 20:04:18.470000-07:00', '9999-12-31 23:59:59.999999+00:00') |
| Basic Homeowner    | HM          | ('2012-06-19 20:04:32.410000-07:00', '9999-12-31 23:59:59.999999+00:00') |
| Business           | B           | ('2012-06-19 20:04:32.680000-07:00', '9999-12-31 23:59:59.999999+00:00') |
| Basic Automobile   | AU          | ('2012-06-19 20:04:32.470000-07:00', '9999-12-31 23:59:59.999999+00:00') |
| Major Medical      | M           | ('2012-06-19 20:04:32.550000-07:00', '2012-06-19 20:19:54.100000-07:00') |
| Premium Homeowner  | HP          | ('2012-06-19 20:04:32.610000-07:00', '9999-12-31 23:59:59.999999+00:00') |

Notice that the Major Medical policy type is a closed row. It shows an end transaction time prior to 9999-12-31. This indicates that the row was logically deleted from the table at the ending transaction time, on 2012-06-19. However, because the row is in a table with a transaction-time dimension, the physical row persists in the table as a closed row.

## Example: As Of Query on a Valid-Time Table

To get a snapshot of a valid-time table where the valid-time period in the result rows overlap a specific time, use the `VALIDTIME AS OF` temporal qualifier in the `SELECT` statement. For example:

```
VALIDTIME AS OF DATE '2005-03-14' SELECT * FROM Policy;
```

| Policy_ID | Customer_ID | Policy_Type | Policy_Details    |
|-----------|-------------|-------------|-------------------|
| 497201    | 304779902   | AU          | STD-CH-524-WXY-00 |

`AS OF` also accepts business calendar functions as date input. This request returns all policies that were valid on January 1, 2011. For more information on the business calendar functions, see *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211.

```
VALIDTIME AS OF TD_YEAR_BEGIN(DATE '2011-05-24') SELECT * FROM Policy;
```

| Policy_ID | Customer_ID | Policy_Type | Policy_Details    |
|-----------|-------------|-------------|-------------------|
| 541008    | 246824626   | AP          | STD-CH-345-NXY-00 |
| 540944    | 123344567   | HM          | STD-PL-332-YXY-00 |

|        |           |    |                   |
|--------|-----------|----|-------------------|
| 497201 | 304779902 | AU | STD-CH-524-WXY-02 |
| 541077 | 766492008 | HP | STD-CH-344_YXY-00 |

## Example: As Of Query on a Transaction-Time Table

To get a snapshot of a transaction-time table where the transaction-time period in the result rows overlap a specific time, use the TRANSACTIONTIME AS OF temporal qualifier in the SELECT statement.

For example:

```
TRANSACTIONTIME AS OF DATE '2012-06-20' SELECT * FROM Policy_Types;
```

| Policy_Name        | Policy_Type |
|--------------------|-------------|
| -----              | -----       |
| Premium Automobile | AP          |
| Basic Homeowner    | HM          |
| Business           | B           |
| Basic Automobile   | AU          |
| Premium Homeowner  | HP          |

Notice that the Major Medical policy type that was logically deleted from the table on 2012-06-19 does not appear in the result set, though it did appear in Example 5 when the NONSEQUENCED TRANSACTIONTIME qualifier to SELECT was used.

## Modifying Temporal Tables

The following examples demonstrate basic modifications to temporal tables.

### Example: Current Valid-Time Insert into a Valid-Time Table

To perform a current valid-time insert into a valid-time table, use the CURRENT VALIDTIME qualifier.

Consider the following valid-time table:

```
CREATE MULTISET TABLE Policy(
 Policy_ID INTEGER,
 Customer_ID INTEGER,
 Policy_Type CHAR(2) NOT NULL,
 Policy_Details CHAR(40),
 Validity PERIOD(DATE) NOT NULL AS VALIDTIME
)
PRIMARY INDEX(Policy_ID);
```

The following statement performs a current valid-time insert into the Policy table. Because the INSERT uses a positional assignment list (where no column names are provided), no value for the valid-time column can be specified. The system timestamps the value of the valid-time column.

```
CURRENT VALIDTIME INSERT INTO Policy
VALUES (541077, 766492008, 'AU', 'STD-CH-344-YXY-00');
```

The following statement also performs a current valid-time insert into the Policy table. Because the INSERT uses a named list, a value for the valid-time column can be specified.

```
CURRENT VALIDTIME INSERT INTO Policy
(Policy_ID, Customer_ID, Policy_Type, Policy_Details, Validity)
VALUES (541145, 616035020, 'AU', 'STD-CH-348-YXN-01',
PERIOD ('(2009-12-03, 2010-12-01)'));
```

## Example: Sequenced Valid-Time Insert into a Valid-Time Table

Use a sequenced valid-time insert to insert history, current, or future rows into a valid-time table. A sequenced valid-time insert is similar to a conventional insert where the valid-time column is treated as any other column in the table.

Consider the same valid-time table as in the previous example. The following statements perform sequenced valid-time inserts into the Policy table.

```
SEQUENCED VALIDTIME INSERT INTO Policy
VALUES (232540, 909234455, 'BM', 'STD-CH-344-YYY-00',
PERIOD (DATE '1999-01-01', DATE '1999-12-31'));

SEQUENCED VALIDTIME INSERT INTO Policy
(Policy_ID, Customer_ID, Policy_Type, Policy_Details, Validity)
VALUES (944540, 344567123, 'BM', 'STD-PL-332-YXY-01',
PERIOD (DATE '2007-02-03', DATE '2008-02-02'));
```

## Example: Nonsequenced Valid-Time Insert into a Valid-Time Table

A nonsequenced valid-time insert is similar to a conventional insert where the valid-time column is treated as any other column in the table.

Consider the same valid-time table as in the previous examples. The following statements perform nonsequenced valid-time inserts into the Policy table.

```
NONSEQUENCED VALIDTIME INSERT INTO Policy
VALUES (540232, 455909234, 'AU', 'STD-CH-344-YYY-00',
```

```

 PERIOD (DATE '2009-01-01', DATE '2009-12-31'));

NONSEQUENCED VALIDTIME INSERT INTO Policy
 (Policy_ID, Customer_ID, Policy_Type, Policy_Details, Validity)
 VALUES (540944, 123344567, 'AU', 'STD-PL-332-YXY-01',
 PERIOD (DATE '2007-02-03', DATE '2008-02-02'));

```

## Example: Current Valid-Time Insert into a Bitemporal Table

To insert data into a bitemporal table that is open in the transaction-time dimension and current in the valid-time dimension, use the **CURRENT VALIDTIME** qualifier.

Consider the following bitemporal table:

```

CREATE MULTISET TABLE Policy_History(
 Policy_ID INTEGER,
 Customer_ID INTEGER,
 Policy_Type CHAR(2) NOT NULL,
 Policy_Details CHAR(40),
 Validity PERIOD(DATE) NOT NULL AS VALIDTIME,
 Policy_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
 AS TRANSACTIONTIME)
PRIMARY INDEX(Policy_ID);

```

The following statement performs a current valid-time insert into the `Policy_History` table. Because the **INSERT** uses a positional assignment list (where no column names are provided), no value for the valid-time column can be specified. Because the system inserts the value for the transaction-time column, no value for the transaction-time column can be specified.

```

CURRENT VALIDTIME INSERT INTO Policy_History
 VALUES (541077, 766492008, 'AU', 'STD-CH-344-YXY-00');

```

The following statement also performs a current valid-time insert into the `Policy_History` table. Because the **INSERT** uses a named list, a value for the valid-time column can be specified. Because the system inserts the value for the transaction-time column, no value for the transaction-time column can be specified.

```

CURRENT VALIDTIME INSERT INTO Policy_History
 (Policy_ID, Customer_ID, Policy_Type, Policy_Details, Validity)
 VALUES (541145, 616035020, 'AU', 'STD-CH-348-YXN-01',
 PERIOD '(2009-12-03, 2010-12-01)');

```

## Example: Sequenced Valid-Time Insert into a Bitemporal Table

A sequenced valid-time insert is similar to a conventional insert, where the valid-time column is treated as any other column in the table. Use a sequenced valid-time insert to insert rows that are history, current, or future in the valid-time dimension.

All such insertions are open in the transaction-time dimension. Because the system automatically inserts the value for the transaction-time column, the INSERT statement cannot specify a value for the transaction-time column.

Consider the following bitemporal table:

```
CREATE MULTISET TABLE Policy_History(
 Policy_ID INTEGER,
 Customer_ID INTEGER,
 Policy_Type CHAR(2) NOT NULL,
 Policy_Details CHAR(40),
 Validity PERIOD(DATE) NOT NULL AS VALIDTIME,
 Policy_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
 AS TRANSACTIONTIME)
PRIMARY INDEX(Policy_ID);
```

The following statements perform sequenced valid-time inserts that are open in the transaction-time dimension into the Policy\_History table.

```
SEQUENCED VALIDTIME INSERT INTO Policy_History
 VALUES (232540, 909234455, 'BM', 'STD-CH-344-YYY-00',
 PERIOD (DATE '1999-01-01', DATE '1999-12-31'));

SEQUENCED VALIDTIME INSERT INTO Policy_History
 (Policy_ID, Customer_ID, Policy_Type, Policy_Details, Validity)
 VALUES (944540, 344567123, 'BM', 'STD-PL-332-YXY-01',
 PERIOD (DATE '2007-02-03', DATE '2008-02-02'));
```

## Example: Nonsequenced Valid-Time Insert into a Bitemporal Table

A nonsequenced valid-time insert is similar to a conventional insert where the valid-time column is treated as any other column in the table. Because the system automatically inserts the value for the transaction-time column, the INSERT statement cannot specify a value for the transaction-time column.

Consider the following bitemporal table:

```
CREATE MULTISET TABLE Policy_History(
 Policy_ID INTEGER,
 Customer_ID INTEGER,
 Policy_Type CHAR(2) NOT NULL,
 Policy_Details CHAR(40),
 Validity PERIOD(DATE) NOT NULL AS VALIDTIME,
 Policy_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
 AS TRANSACTIONTIME)
PRIMARY INDEX(Policy_ID);
```

The following statements perform nonsequenced valid-time inserts that are open in the transaction-time dimension into the Policy\_History table.

```
NONSEQUENCED VALIDTIME INSERT INTO Policy_History
 VALUES (540232, 450909234, 'AU', 'STD-CH-344-YYY-00',
 PERIOD (DATE '2009-11-01', UNTIL_CHANGED));

NONSEQUENCED VALIDTIME INSERT INTO Policy_History
 (Policy_ID, Customer_ID, Policy_Type, Policy_Details, Validity)
 VALUES (540944, 120344567, 'AU', 'STD-PL-332-YXY-01',
 PERIOD (DATE '2010-02-03', DATE '2011-02-02'));
```

## Example: Nontemporal Insert into a Bitemporal Table

A nontemporal insert in to a bitemporal table is similar to a conventional insert, where the valid-time and transaction-time columns are treated as any other column in the table. You can use a nontemporal insert to insert closed or open rows.

To perform a nontemporal insert, you must have the NONTEMPORAL privilege on the target table.

Consider the following bitemporal table:

```
CREATE MULTISET TABLE Policy_History(
 Policy_ID INTEGER,
 Customer_ID INTEGER,
 Policy_Type CHAR(2) NOT NULL,
 Policy_Details CHAR(40),
 Validity PERIOD(DATE) NOT NULL AS VALIDTIME,
 Policy_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
 AS TRANSACTIONTIME)
PRIMARY INDEX(Policy_ID);
```

The following nontemporal INSERT statements insert rows into Policy\_History, explicitly specifying values for the valid-time and transaction-time columns:

```

NONTEMPORAL INSERT INTO Policy_History
VALUES (411458, 160350204, 'AU', 'STD-CH-340-YXN-01',
PERIOD '(2009-12-03, 2010-12-01)',
PERIOD (TIMESTAMP '2004-01-01 00:00:00.000000', UNTIL_CLOSED));

NONTEMPORAL INSERT INTO Policy_History
VALUES (114583, 603502048, 'AU', 'STD-CH-920-YXD-01',
PERIOD '(2009-12-08, 2010-12-07)',
PERIOD (TIMESTAMP '2004-01-01 00:00:00.000000', UNTIL_CLOSED));

```

## Example: Current Insert into a Transaction-Time Table

The following INSERT statement inserts an open row into the Policy\_Types table. Because the system automatically inserts the value for the transaction-time column, no value for the transaction-time column can be specified.

```

INSERT INTO Policy_Types
VALUES ('Basic Motorcycle', 'BM');

```

## Example: Nontemporal Insert into a Transaction-Time Table

A nontemporal insert is similar to a conventional insert, where the transaction-time column is treated as any other column in the table. You can use a nontemporal insert to insert closed or open rows.

### Note:

To perform a nontemporal insert, you must have the NONTEMPORAL privilege on the target table.

Consider the following transaction-time table:

```

CREATE MULTISET TABLE Policy_Types (
 Policy_Name VARCHAR(20),
 Policy_Type CHAR(2) NOT NULL PRIMARY KEY,
 Policy_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
 AS TRANSACTIONTIME
)
PRIMARY INDEX (Policy_Name);

```

The following nontemporal INSERT statements insert rows into Policy\_Types, explicitly specifying values for the transaction-time column:

```

NONTEMPORAL INSERT INTO Policy_Types
VALUES ('Premium Automobile', 'AP',
PERIOD (TIMESTAMP '2004-01-01 00:00:00.000000', UNTIL_CLOSED));

NONTEMPORAL INSERT INTO Policy_Types
(Policy_Name, Policy_Type, Policy_Duration)
VALUES ('Basic Homeowner', 'HM',
PERIOD (TIMESTAMP '2004-01-01 00:00:00.000000', UNTIL_CLOSED));

```

## Example: Update a Row in a Valid-Time Table

Assume an insurance customer has a basic homeowner's policy begin on February 3, 2007, which is valid and renewed automatically unless the policy holder requests a change. The full information, including the period of validity for the customer's policy (in the Policy\_Term column) is shown by the following query:

```

NONSEQUENCED VALIDTIME SELECT * FROM Policy WHERE Policy_ID=540944;
Policy_ID Customer_ID Policy_Type Policy_Details Policy_Term

540944 123344567 HM STD-PL-332-YXY-01 ('07/02/03', '99/12/31')

```

Now assume that the customer requests after a year that the policy be upgraded to a premium homeowner's policy. The update statement specifies a period of applicability for the change, which would be from the day of the change, February 3, 2008, indefinitely again, until the customer requests further changes or cancels the policy. The following statement would make the required update to the customer policy:

```

SEQUENCED VALIDTIME PERIOD '(2008-02-03, UNTIL_CHANGED)'
UPDATE POLICY SET Policy_Type='HP'
WHERE Policy_ID=540944;

```

The result will be two rows in the table for this Policy\_ID:

- a history row showing the initial policy with the Policy\_Term column (the valid-time column) showing an ending date corresponding to when the change was made
- a row reflecting the current policy, with valid time starting on the date the change was made

```

NONSEQUENCED VALIDTIME SELECT * FROM Policy where Policy_ID=540944;
Policy_ID Customer_ID Policy_Type Policy_Details Policy_Term

540944 123344567 HP STD-PL-332-YXY-01 ('08/02/03', '99/12/31')
540944 123344567 HM STD-PL-332-YXY-01 ('07/02/03', '08/02/03')

```

For a nontemporal table, using nontemporal semantics, the Policy\_Type value in the row would have simply been replaced, with no history row left in the table to show how the policy had existed prior to the change.

## Example: Current Delete from a Valid-Time Table

To perform a current delete, use the CURRENT VALIDTIME qualifier in the DELETE statement.

For a table with valid time, current rows qualify for deletion. Depending on the period of validity of a qualified row and whether the table also supports transaction time, the delete operation may physically delete a row, logically delete a row, modify the period of validity for a row, or logically delete a row and create a new row. Consider the following data in the Policy table:

```

NONSEQUENCED VALIDTIME
SELECT Policy_ID, Customer_ID, Validity
FROM Policy
WHERE Policy_Type = 'AU';

```

| Policy_ID | Customer_ID | Validity                 |
|-----------|-------------|--------------------------|
| 497201    | 304779902   | ('05/02/14', '06/02/13') |
| 540944    | 123344567   | ('07/02/03', '08/02/02') |
| 541077    | 766492008   | ('09/12/21', '99/12/31') |
| 541145    | 616035020   | ('09/12/03', '10/12/01') |
| 541008    | 246824626   | ('09/10/01', '99/12/31') |

Suppose the value of TEMPORAL\_DATE is the following:

```
SELECT TEMPORAL_DATE;
```

| Temporal Date |
|---------------|
| 09/12/21      |

The following current DELETE statement physically deletes the qualified row from the table because the beginning bound of the period of validity is equal to TEMPORAL\_DATE:

```

CURRENT VALIDTIME DELETE
FROM Policy
WHERE Policy_ID = 541077;

```

## Example: Current Delete from a Valid-Time Table

The following current DELETE statement modifies the period of validity for the qualified row from the table because the beginning bound of the period of validity is less than TEMPORAL\_DATE. The row becomes a history row.

```

CURRENT VALIDTIME DELETE
FROM Policy
WHERE Policy_ID = 541145;

NONSEQUENCED VALIDTIME
SELECT Policy_ID, Customer_ID, Validity
FROM Policy
WHERE Policy_Type = 'AU';

```

| Policy_ID | Customer_ID | Validity                 |
|-----------|-------------|--------------------------|
| -----     | -----       | -----                    |
| 497201    | 304779902   | ('05/02/14', '06/02/13') |
| 540944    | 123344567   | ('07/02/03', '08/02/02') |
| 541145    | 616035020   | ('09/12/03', '09/12/21') |
| 541008    | 246824626   | ('09/10/01', '99/12/31') |

## Example: Current Modifications Do Not Apply to Future Rows

This example demonstrates that current data modifications do not apply to future rows. Assume the following tables describe a company's employees and departments:

```
CREATE MULTISET TABLE employee ,NO FALLBACK ,
 NO BEFORE JOURNAL,
 NO AFTER JOURNAL,
 CHECKSUM = DEFAULT
(
 eid INTEGER NOT NULL,
 ename VARCHAR(50) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
 bdate DATE FORMAT 'yyyy/mm/dd',
 job_duration PERIOD(DATE) NOT NULL AS VALIDTIME,
 deptid INTEGER,
 mid INTEGER)

PRIMARY INDEX (eid);

CREATE MULTISET TABLE dept ,NO FALLBACK ,
 NO BEFORE JOURNAL,
 NO AFTER JOURNAL,
 CHECKSUM = DEFAULT
(
 deptid INTEGER NOT NULL,
 deptname VARCHAR(100) CHARACTER SET LATIN NOT CASESPECIFIC)
UNIQUE PRIMARY INDEX (deptid);
```

Assume the company management must cut down the staff of the SUPPORT department. Employees who have been with the company for less than three months must be discharged. The following query removes employees with job durations of less than three months:

```
CURRENT VALIDTIME
DELETE employee
FROM dept
WHERE dept.deptname = 'SUPPORT' AND
```

```
dept.deptid = employee.deptid AND
BEGIN(job_duration) > CURRENT_DATE - interval '3' month;
```

Now assume that the system includes employee entries for new employees who have not yet started working at the company. The DELETE statement above will not remove these future employees. To remove them together with the current employees, the following statement could be used:

```
BT;
/* delete currently employees in department with less than 3 months
work*/
CURRENT VALIDTIME
DELETE employee
FROM dept
WHERE dept.deptname = 'SUPPORT' AND
 dept.deptid = employee.deptid AND
 BEGIN(job_duration) > current_date - interval '3' month;

/* delete all future employees */
SEQUENCED VALIDTIME
DELETE employee
FROM dept
WHERE dept.deptname = 'SUPPORT' AND
 dept.deptid = employee.deptid AND
 BEGIN(job_duration) > TEMPORAL_DATE;

ET;
```

Alternatively, the following SQL would accomplish the same:

```
REPLACE VIEW v1 AS
NONSEQUENCED VALIDTIME
SELECT employee.eid, dept.deptid
FROM employee, dept
WHERE dept.deptname = 'SUPPORT' AND
 dept.deptid = employee.deptid AND
 BEGIN(job_duration) > CURRENT_DATE - interval '3' month
 AND job_duration OVERLAPS PERIOD(TEMPORAL_DATE, UNTIL_CHANGED);
SEQUENCED VALIDTIME PERIOD(TEMPORAL_DATE, UNTIL_CHANGED)
DELETE employee FROM v1 WHERE v1.eid = employee.eid AND
 v1.deptid = employee.deptid;
```

## Example: Sequenced Delete from a Valid-Time Table

To perform a sequenced delete, use the VALIDTIME or SEQUENCED VALIDTIME qualifier in the DELETE statement.

For a table with valid time, any row with a period of validity that overlaps with the period of applicability qualifies for deletion. The delete operation may physically delete a row, logically delete a row, modify the period of validity for a row, or delete a row and create a new row. Consider the following data in the Policy table:

```

NONSEQUENCED VALIDTIME
SELECT Policy_ID, Customer_ID, Validity
FROM Policy
WHERE Policy_Type = 'AU';

```

| Policy_ID | Customer_ID | Validity                 |
|-----------|-------------|--------------------------|
| 497201    | 304779902   | ('05/02/14', '06/02/13') |
| 540944    | 123344567   | ('07/02/03', '08/02/02') |
| 541077    | 766492008   | ('09/12/21', '99/12/31') |
| 541145    | 616035020   | ('09/12/03', '10/12/01') |
| 541008    | 246824626   | ('09/10/01', '99/12/31') |

The following sequenced DELETE statement physically deletes one row from the table. The period of validity for policy 540944 (PERIOD '(2007-02-03, 2008-02-02)') is fully contained within the period of applicability of the sequenced delete statement (PERIOD '(2007-01-01, 2008-03-01)'):

```

SEQUENCED VALIDTIME PERIOD '(2007-01-01, 2008-03-01)' DELETE
FROM Policy;

```

## Example: Sequenced Delete from a Valid-Time Table

If the period of applicability of the sequenced delete in the last example had been (PERIOD '(2005-05-01, 2005-06-01)'), such that it was smaller than, and fully contained within, the period of validity of policy 497201, the policy row would be split into two rows, preserving the validity for periods that were not deleted:

```

SEQUENCED VALIDTIME PERIOD '(2005-05-01, 2005-06-01)' DELETE
FROM Policy;

NONSEQUENCED VALIDTIME
SELECT Policy_ID, Customer_ID, Validity
FROM Policy
WHERE Policy_Type = 'AU';

```

| Policy_ID | Customer_ID | Validity                 |
|-----------|-------------|--------------------------|
| -----     | -----       | -----                    |
| 497201    | 304779902   | ('05/02/14', '05/05/01') |
| 497201    | 304779902   | ('05/06/01', '06/02/13') |
| 541145    | 616035020   | ('09/12/03', '10/12/01') |
| 541077    | 766492008   | ('09/12/21', '99/12/31') |
| 541008    | 246824626   | ('09/10/01', '99/12/31') |

## Example: Sequenced Delete from a Valid-Time Table

If the period of applicability of the sequenced delete in the last example had been (PERIOD '(2005-05-01, 2005-06-01)'), such that it was smaller than, and fully contained within, the period of validity of policy 497201, the policy row would be split into two rows, preserving the validity for periods that were not deleted:

```
SEQUENCED VALIDTIME PERIOD '(2005-05-01, 2005-06-01)' DELETE
FROM Policy;
```

```
NONSEQUENCED VALIDTIME
SELECT Policy_ID, Customer_ID, Validity
FROM Policy
WHERE Policy_Type = 'AU';
```

| Policy_ID | Customer_ID | Validity                 |
|-----------|-------------|--------------------------|
| -----     | -----       | -----                    |
| 497201    | 304779902   | ('05/02/14', '05/05/01') |
| 497201    | 304779902   | ('05/06/01', '06/02/13') |
| 541145    | 616035020   | ('09/12/03', '10/12/01') |
| 541077    | 766492008   | ('09/12/21', '99/12/31') |
| 541008    | 246824626   | ('09/10/01', '99/12/31') |

## Example: Nonsequenced Delete from a Valid-Time Table

A nonsequenced delete applies no special temporal logic to the delete operation or row selection, and operates on a valid-time table as a conventional delete would operate on a nontemporal table:

Start from the same valid-time table that was used for the sequenced delete examples:

| Policy_ID | Customer_ID | Validity                 |
|-----------|-------------|--------------------------|
| -----     | -----       | -----                    |
| 497201    | 304779902   | ('05/02/14', '06/02/13') |
| 540944    | 123344567   | ('07/02/03', '08/02/02') |
| 541077    | 766492008   | ('09/12/21', '99/12/31') |

|        |           |                          |
|--------|-----------|--------------------------|
| 541145 | 616035020 | ('09/12/03', '10/12/01') |
| 541008 | 246824626 | ('09/10/01', '99/12/31') |

Each of the following nonsequenced DELETE statements physically deletes one row from the table:

```

NONSEQUENCED VALIDTIME DELETE
FROM Policy
WHERE Customer_ID = 304779902;

```

```

NONSEQUENCED VALIDTIME DELETE
FROM Policy
WHERE BEGIN(Validity) = DATE '2007-02-03';

```

## Example: Current or Sequenced Delete from a Bitemporal Table

The syntax of these operations is identical to the same kinds of deletions performed on valid-time tables:

- To perform a current delete, use the CURRENT VALIDTIME qualifier in the DELETE statement.
- To perform a sequenced delete, use the SEQUENCED VALIDTIME qualifier in the DELETE statement. (Using VALIDTIME alone as the qualifier is equivalent.)

There are two important ways that these kinds of deletions on bitemporal tables differ from those on valid-time tables:

- Current and sequenced deletions on bitemporal tables affect only rows that are open in the transaction-time dimension.
- Because rows are physically removed from bitemporal tables only when the NONTEMPORAL qualifier is used, rows deleted in SEQUENCED VALIDTIME are only deleted logically. The ending bound of their transaction-time period is changed from the value of UNTIL\_CLOSED to the date or timestamp of the deletion, and the row becomes closed in the transaction-time dimension. The logically deleted row becomes a history row.

The valid-time period remains unchanged for the logically deleted row. The deleted state of the row is reflected in the ending bound of the transaction time. However, similar to a SEQUENCED VALIDTIME DELETE on a valid-time table, if the period of validity of the original row extended beyond the period of applicability of the sequenced delete new rows are created that reflect the time periods for which the information was not deleted. The new rows have appropriately modified valid-time periods. These new rows are open in the transaction-time dimension, because their time periods were not included in the period of applicability of the deletion.

## Example: Nontemporal Delete from a Bitemporal Table

Performing a nontemporal delete on a bitemporal table physically deletes the specified rows. Because a nontemporal delete can be used to remove history rows from the table, the NONTEMPORAL privilege is required to perform nontemporal operations on temporal tables that have transaction time. Nontemporal deletes should be used only if absolutely necessary, and only by appropriately authorized personnel.

## Example: Merging Nontemporal Table Data into a Row-Partitioned Bitemporal Table

You can use the temporal form of the MERGE statement to merge data from a nontemporal table into a primary-indexed temporal table. Suppose you have the following nontemporal table called `Policy_Changes`:

```
CREATE TABLE Policy_Changes(
 Policy_ID INTEGER,
 Customer_ID INTEGER,
 Policy_Type CHAR(2) NOT NULL,
 Policy_Details CHAR(40)
);
```

Suppose you also have the following bitemporal table called `Policy` that is row partitioned according to the partitioning guidelines for a bitemporal table:

```
CREATE MULTISET TABLE Policy(
 Policy_ID INTEGER,
 Customer_ID INTEGER,
 Policy_Type CHAR(2) NOT NULL,
 Policy_Details CHAR(40),
 Validity PERIOD(DATE) AS VALIDTIME,
 Policy_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
 AS TRANSACTIONTIME
)
PRIMARY INDEX(Policy_ID)
PARTITION BY
 CASE_N((END(Validity) IS NULL OR
 END(Validity) >= CURRENT_DATE AT '-12:59') AND
 END(Policy_Duration) >= CURRENT_TIMESTAMP,
 END(Validity) < CURRENT_DATE AT '-12:59' AND
 END(Policy_Duration) >= CURRENT_TIMESTAMP,
 END(Policy_Duration) < CURRENT_TIMESTAMP);
```

The following statement performs a sequenced merge in the valid-time dimension into the `Policy` table from the `Policy_Changes` table where the period of applicability is December 1, 2009 to December 7, 2009.

The matching condition is applied on open rows of the `Policy` table where the period of validity overlaps the period of applicability. If the matching condition is satisfied, a sequenced update is performed; if the matching condition is not satisfied, a sequenced insert is performed.

```
SEQUENCED VALIDTIME
MERGE INTO Policy USING (
```

```

NONSEQUENCED VALIDTIME PERIOD (DATE'2009-12-01', DATE'2009-12-07')
SELECT
 source.Policy_ID,
 source.Customer_ID,
 source.Policy_Type,
 source.Policy_Details,
 target.Validity AS vt,
 END(target.Policy_Duration) AS ett
FROM Policy_Changes source LEFT OUTER JOIN Policy target
ON source.Policy_ID = target.Policy_ID
WHERE (vt IS NULL OR
 ((BEGIN(vt) < DATE '2009-12-07') AND
 (END(vt) > DATE '2009-12-01') AND
 (ett = TIMESTAMP '9999-12-31 23:59:59.999999')))
) AS merge_source (
 PID,
 CID,
 PType,
 PDetails,
 j,
 k
)
ON (Policy_ID = merge_source.PID) AND
 END(Validity) = END(j) AND END(Policy_Duration) = k
WHEN MATCHED THEN
 UPDATE SET Policy_Details = merge_source.PDetails
WHEN NOT MATCHED THEN
 INSERT (
 merge_source.PID,
 merge_source.CID,
 merge_source.PType,
 merge_source.PDetails,
 PERIOD(TEMPORAL_DATE, UNTIL_CHANGED)
);

```

## Example: Dropping a Valid-Time Column

To drop a valid-time column from a valid-time table, use the ALTER TABLE statement.

Consider the following valid-time table:

```

CREATE MULTISET TABLE Customer (
 Customer_Name VARCHAR(40),
 Customer_ID INTEGER,
 Customer_Address VARCHAR(80),
 Customer_Phone VARCHAR(12),
 Customer_Validity PERIOD(DATE) NOT NULL AS VALIDTIME
)
PRIMARY INDEX (Customer_ID);

```

The following statement drops the Customer\_Validity column:

```

ALTER TABLE Customer DROP Customer_Validity;

```

To drop a valid-time column from a bitemporal table, use the ALTER TABLE statement and specify the NONTEMPORAL qualifier. Dropping any type of column from a bitemporal table requires the NONTEMPORAL privilege on the table, and the NONTEMPORAL qualifier to ALTER TABLE must be used.

Consider the following bitemporal table:

```
CREATE MULTISET TABLE Customer (
 Customer_Name VARCHAR(40),
 Customer_ID INTEGER,
 Customer_Address VARCHAR(80),
 Customer_Phone VARCHAR(12),
 Customer_Validity PERIOD(DATE) NOT NULL AS VALIDTIME,
 Customer_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
 AS TRANSACTIONTIME
)
PRIMARY INDEX (Customer_ID);
```

The following statement drops the Customer\_Validity column:

```
NONTEMPORAL ALTER TABLE Customer DROP Customer_Validity;
```

When a valid-time column is dropped from a bitemporal table, all rows that are no longer valid (all history rows in the valid-time dimension) are physically deleted from the table.

## Example: Dropping a Transaction-Time Column

Dropping any type of column from a transaction-time or bitemporal table requires the NONTEMPORAL privilege on the table, and the NONTEMPORAL qualifier to ALTER TABLE must be used.

Consider the following transaction-time table:

```
CREATE MULTISET TABLE Customer (
 Customer_Name VARCHAR(40),
 Customer_ID INTEGER,
 Customer_Address VARCHAR(80),
 Customer_Phone VARCHAR(12),
 Customer_Duration PERIOD(TIMESTAMP(6) WITH TIME ZONE) NOT NULL
 AS TRANSACTIONTIME
)
PRIMARY INDEX (Customer_ID);
```

Assuming that you have the NONTEMPORAL privilege on the Customer table, the following ALTER TABLE statement drops the Customer\_Duration column:

```
NONTEMPORAL ALTER TABLE Customer DROP Customer_Duration;
```

When a transaction-time column is dropped from a transaction-time or bitemporal table, all closed rows (all history rows in the transaction-time dimension) are physically deleted from the table.

## Views on Temporal Tables

The following examples create views on temporal tables.

### Example: Creating a Sequenced View on a Temporal Table

The following statement creates a sequenced view on the Policy table. The result of the sequenced query is a valid-time table or, in this case, view. The valid-time period for each row in the view is stored in a new column that is automatically appended by the system. The valid-time for each row in the view is the overlap of the valid-time period of the row in the original temporal table with the valid time of the sequenced query. In this case, because a time period is not specified in the sequenced query, the period for the query defaults to (0001-01-01, UNTIL\_CHANGED), and the valid-time periods in the view will match those for the original rows.

Because names for the view columns are not specified in the CREATE VIEW statement, the system assigns the new valid-time column the name VALIDTIME.

```
CREATE VIEW Basic_Auto_Policy_V AS
SEQUENCED VALIDTIME
SELECT Policy_ID, Customer_ID
FROM Policy
WHERE Policy_Type = 'AU';
```

### Example: Creating a Sequenced View on a Temporal Table and Specifying the Valid-Time Column

The following statement creates a similar sequenced view on the Policy table but provides a list of column names that includes the extra column name “Basic\_View\_Validity”, which the system assigns to the new valid-time column that is appended to the view.

```
CREATE VIEW Basic_Auto_Policy_V (
 Policy_ID,
 Customer_ID,
 Basic_View_Validity
) AS
SEQUENCED VALIDTIME
SELECT Policy_ID, Customer_ID
```

```
FROM Policy
WHERE Policy_Type = 'AU';
```

## Example: Creating a Sequenced View on a Temporal Table Using a Business Calendar Function

The following statement employs a business calendar function in an AS OF clause to define a view that shows insurance policies valid at the beginning of the current year.

```
CREATE VIEW ValidePoliciesOnJan1 AS
VALIDTIME AS OF TD_YEAR_BEGIN(CURRENT_DATE) SELECT * FROM Policy;
```

## Session Temporal Qualifier

### Example: Setting the Session Temporal Qualifier to CURRENT VALIDTIME

The following statement sets the session temporal qualifier to current in the valid-time dimension:

```
SET SESSION CURRENT VALIDTIME;
```

### Example: Setting the Session Temporal Qualifier to CURRENT TRANSACTIONTIME

The following statement sets the session temporal qualifier to current in the transaction-time dimension:

```
SET SESSION CURRENT TRANSACTIONTIME;
```

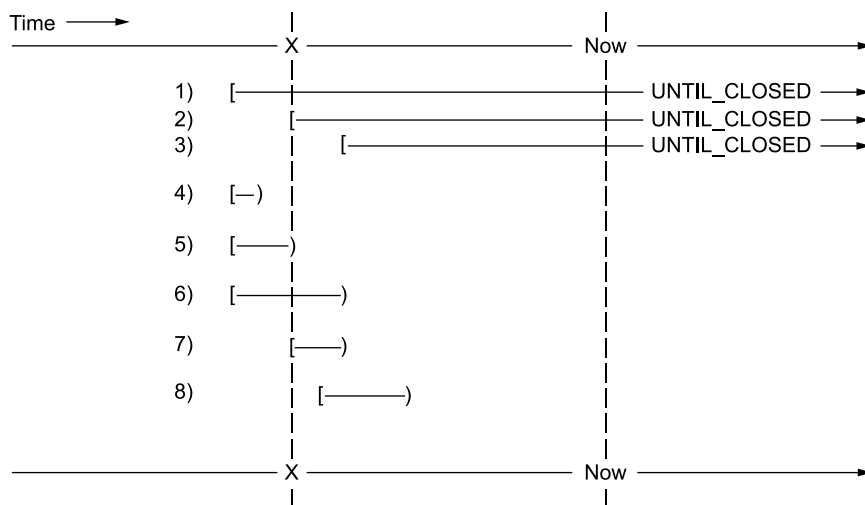
## Restoring a Prior Table State

Tables with transaction time automatically store in the table a snapshot copy of any row that is modified or deleted. This characteristic of these tables can be used to recover a prior state of the table using only SQL. This can be useful to quickly recover from a localized problem, such as if a table or set of tables have become corrupted by a user error, and need to be restored to a consistent state.

Assume a table that has transaction time needs to be restored to the state it was in at a point (time X) prior to the current time. Rows in the table can be classified based on whether they are open or closed, and on their transaction-time column period relation to time X:

| Row | Row State | BEGIN(TT)<br>Beginning Transaction Time | END(TT)<br>End Transaction Time |
|-----|-----------|-----------------------------------------|---------------------------------|
| 1   | Open      | Prior to time X                         | UNTIL_CLOSED                    |
| 2   | Open      | Equal to time X                         | UNTIL_CLOSED                    |
| 3   | Open      | After time X                            | UNTIL_CLOSED                    |
| 4   | Closed    | Prior to time X                         | Prior to time X                 |
| 5   | Closed    | Prior to time X                         | Equal to time X                 |
| 6   | Closed    | Prior to time X                         | After time X                    |
| 7   | Closed    | Equal to time X                         | After time X                    |
| 8   | Closed    | After time X                            | After time X                    |

These rows are represented graphically below.



## Example: Restoring a Temporal Table to a Prior State

To return the table back to the state it was in at time x, use the following plan:

| Row | Row at Time X<br>Compared to Row Now      | Plan                    |
|-----|-------------------------------------------|-------------------------|
| 1   | Row existed at time X and row exists now. | Leave the row as it is. |
| 2   | It existed at time X and row exists now.  | Leave the row as it is. |
| 3   | It did not exist at time X.               | Delete the row.         |

| Row | Row at Time X<br>Compared to Row Now              | Plan                                                         |
|-----|---------------------------------------------------|--------------------------------------------------------------|
| 4   | It was closed at time X, and is still closed now. | Leave the row as it is.                                      |
| 5   | It was closed at time X, and is still closed now. | Leave the row as it is.                                      |
| 6   | It was open at time X, but is closed now.         | Leave BEGIN(TT) as it is.<br>Update END(TT) to UNTIL_CLOSED. |
| 7   | It was open at time X, but row is closed now.     | Leave BEGIN(TT) as it is.<br>Update END(TT) to UNTIL_CLOSED. |
| 8   | It did not yet exist at time X.                   | Delete the row.                                              |

The following SQL will realize the plan for each of the different row states.

```

NONTEMPORAL DELETE tt_table where BEGIN(tt_col) > time X;
NONTEMPORAL UPDATE tt_table
 SET tt_col = PERIOD(BEGIN(tt_col), UNTIL_CLOSED)
 WHERE END(tt_col) IS NOT UNTIL_CLOSED
 AND BEGIN(tt_col) <= time X
 AND END(tt_col) > time X;

```

**Note:**

Because this solution requires NONTEMPORAL SQL, nontemporal operations on temporal tables must be enabled using the DBS Control utility. Additionally, the user executing this SQL must be granted the NONTEMPORAL privilege. For more information on DBS Control see *Teradata Vantage™ - Database Utilities*, B035-1102. For more information on NONTEMPORAL operations, see [Usage Notes](#).

# Potential Concurrency Issues with Current Temporal DML

This section describes a potential concurrency issue that can occur with CURRENT VALIDTIME transactions, and techniques for avoiding these issues.

## Transaction Isolation

Isolation is one of the ACID properties of database transaction processing. Ideally, “concurrent” transactions are serializable, each occurring in isolation from the others, the results of each being independent of any changes resulting from the others. Normally, Vantage guarantees serializable transactions when the transaction isolation level is set to SERIALIZABLE for the transaction or for the session.

However, under some circumstances, CURRENT VALIDTIME selections from temporal tables can violate serializability in temporal transactions, even when the transaction isolation level is set to SERIALIZABLE. This can happen because of the interaction of two characteristics of temporal transactions:

- When rows in temporal tables are changed, they usually generate history rows, which preserve the state of the row as it existed before the change occurred, and which are stored in the same table.
- A CURRENT temporal qualifier selects rows based on the TEMPORAL\_DATE or TEMPORAL\_TIMESTAMP, which is defined as the clock time of the first access to the temporal table, or the first access of the transaction to the TEMPORAL\_DATE or TEMPORAL\_TIMESTAMP function.

This section illustrates the issues, and describes techniques that can be used to ensure serializable temporal transactions.

## Examples

### Note:

Time and date values are exaggerated to make the sequence of events clearer.

Assume the following two tables describe parts and orders:

Parts table:

| Part_ID | Supplier_ID | Price | Discount | Part_Validity            |
|---------|-------------|-------|----------|--------------------------|
| P1      | S1          | \$10  | 10%      | (2008-01-01, 2011-01-01) |

Orders table:

| Order_ID | Part_ID | Quantity | Order_Validity           |
|----------|---------|----------|--------------------------|
| O1       | P1      | 60       | (2008-01-01, 2011-01-01) |

Consider the following two transactions, that are to be applied to the tables:

- **Increase Order**

If the current discount for part P1 is greater than or equal to 10%, double the order quantity.

This transaction modifies the order table based on the parts table:

```
CURRENT VALIDTIME
UPDATE Orders
FROM Parts
SET Quantity= quantity*2
WHERE discount >= 10 AND Orders.part_id = Parts.part_id;
```

- **Reduce Discount**

If the current order quantity for P1 is less than 100, reduce the discount by one half.

This transaction modifies the parts table based on the order table

```
CURRENT VALIDTIME
UPDATE Parts
FROM Orders
SET discount = discount/2.0
WHERE quantity < 100 AND Parts.part_id = Orders.part_id ;
```

Assuming the transactions execute serially, the results will depend on which transaction executes first. This is demonstrated by the first two examples below.

## Example: Increase Order occurs before Reduce Discount

Assume the individual transactions have the following characteristics:

| Times             | Increase Order Transaction | Reduce Discount Transaction |
|-------------------|----------------------------|-----------------------------|
| Begin Time        | 2009-01-02                 | 2009-01-09                  |
| Modification Time | 2009-01-07                 | 2009-01-10                  |
| End Time          | 2009-01-08                 | 2009-01-11                  |

- Begin Time is the value of TEMPORAL\_DATE for the example transactions. This time is used both to qualify rows for participation in the transaction, and to timestamp the modified rows.
- Modification Time is the time when the modification is made by the transaction.
- End Time is when the transaction is committed and completed.

Notice that Increase Order ends before Reduce Discount begins.

The following shows the states of the Orders and Parts tables after transactions have completed.

Orders table state after the Increase Order transaction:

| Order_ID | Part_ID | Quantity | Order_Vaildity           |
|----------|---------|----------|--------------------------|
| O1       | P1      | 60       | (2008-01-01, 2009-01-02) |
| O1       | P1      | 120      | (2009-01-02, 2011-01-01) |

In the original state of the tables, the P1 rows qualified for the CURRENT transaction because their valid times overlapped current time, value of TEMPORAL\_DATE at the time when the transaction started, shown as Begin Time in the table of transaction characteristics. The period (2008-01-01, 2011-01-01) overlaps 2009-01-02.

Additionally, the P1 row in the Orders table was modified, because the P1 row in the Parts table fulfilled the transaction WHERE test (discount >= 10).

The modification of the P1 row in the Orders table leaves a history row showing the state of the row prior to the modification. Notice the end time of the first row and the beginning time of the second row have both been timestamped with TEMPORAL\_DATE at the time of the modification.

Parts Table state after the Reduce Discount transaction:

| Part_ID | Supplier_ID | Price | Discount | Part_Vaildity            |
|---------|-------------|-------|----------|--------------------------|
| P1      | S1          | \$10  | 10%      | (2008-01-01, 2011-01-01) |

The P1 row was not changed, because the current time at the time of the Reduce Discount transaction was 2009-01-09. Using this current time, only the second row in the Orders table qualifies for the transaction, however that row fails the WHERE test (quantity < 100), due to the changes made by the preceding Increase Order transaction.

## Example: Reduce Discount occurs before Increase Order

Assume the individual transactions have the following characteristics:

| Times             | Increase Order Transaction | Reduce Discount Transaction |
|-------------------|----------------------------|-----------------------------|
| Begin Time        | 2009-01-05                 | 2009-01-02                  |
| Modification Time | 2009-01-10                 | 2009-01-03                  |
| End Time          | 2009-01-11                 | 2009-01-04                  |

Notice that Reduce Discount ends before Increase Order begins.

The following shows the states of the Parts and Orders tables after transactions have completed.

Parts Table after the Reduce Discount transaction:

| Part_ID | Supplier_ID | Price | Discount | Part_Vaildity            |
|---------|-------------|-------|----------|--------------------------|
| P1      | S1          | \$10  | 10%      | (2008-01-01, 2009-01-02) |

| Part_ID | Supplier_ID | Price | Discount | Part_Vaildity            |
|---------|-------------|-------|----------|--------------------------|
| P1      | S1          | \$10  | 5%       | (2009-01-02, 2011-01-01) |

In the original state of the tables, the P1 rows qualified for the CURRENT transaction because their valid times overlapped TEMPORAL\_DATE.

Additionally, the P1 row in the Parts table was modified, because the P1 row in the Orders table fulfilled the transaction WHERE test (quantity < 100).

The modification of the P1 row in the Parts table leaves a history row showing the state of the row prior to the modification. Notice the end time of the first row and the beginning time of the second row have both been timestamped with TEMPORAL\_DATE at the time of the modification.

Orders table after Increase Order transaction:

| Order_ID | Part_ID | Quantity | Order_Vaildity           |
|----------|---------|----------|--------------------------|
| O1       | P1      | 60       | (2008-01-01, 2011-01-01) |

The row was not changed, because the current time at the time of the Increase Order transaction was 2009-01-05. Using this current time, only the second row in the Parts table qualifies for the transaction, however that row fails the WHERE test (discount >= 10), due to the changes made by the preceding Reduce Discount transaction.

## Example: Increase Order and Reduce Discount occur concurrently

Assume the individual transactions have the following characteristics:

| Times             | Increase Order Transaction | Reduce Discount Transaction |
|-------------------|----------------------------|-----------------------------|
| Begin Time        | 2009-01-02                 | 2009-01-04                  |
| Modification Time | 2009-01-07                 | 2009-01-05                  |
| End Time          | 2009-01-08                 | 2009-01-06                  |

Notice that Reduce Discount begins and ends during the time Increase Order is running.

The following shows the states of the Parts and Orders tables after transactions have completed.

Parts Table after the Reduce Discount transaction:

| Part_ID | Supplier_ID | Price | Discount | Part_Vaildity            |
|---------|-------------|-------|----------|--------------------------|
| P1      | S1          | \$10  | 10%      | (2008-01-01, 2009-01-04) |
| P1      | S1          | \$10  | 5%       | (2009-01-04, 2011-01-01) |

In the original state of the tables, the P1 rows qualified for the CURRENT transaction because their valid times overlapped TEMPORAL\_DATE.

Additionally, the P1 row in the Parts table was modified, because the P1 row in the Orders table at that current time (2009-01-04) fulfilled the transaction WHERE test (quantity < 100).

The modification of the P1 row in the Parts table leaves a history row showing the state of the row prior to the modification. Notice the end time of the first row and the beginning time of the second row have both been timestamped with TEMPORAL\_DATE at the time of the modification.

Orders table after Increase Order transaction:

| Order_ID | Part_ID | Quantity | Order_Vaildity           |
|----------|---------|----------|--------------------------|
| O1       | P1      | 60       | (2008-01-01, 2009-01-02) |
| O1       | P1      | 120      | (2009-01-02, 2011-01-01) |

The original P1 row in the Orders table qualified for the CURRENT transaction because its valid time (2008-01-01, 2011-01-01) overlapped TEMPORAL\_DATE for the Increase Order transaction (2009-01-02).

Similarly, the original row in the Parts table qualified for the CURRENT transaction because its valid time (2008-01-01, 2009-01-04) overlapped TEMPORAL\_DATE for the Increase Order transaction (2009-01-02). The row also fulfilled the WHERE test (discount >= 10), so the order quantity was increased, resulting in a history row in the Orders table showing the state of the row prior to the modification.

In this case, the final state of the database tables does not match either of the cases where the transactions occurred serially, so the transaction isolation principle of ACID has been violated.

## Recommendations

The following techniques can be used to avoid potential serializability issues with CURRENT temporal transaction concurrency.

- Do not concurrently run multiple applications or transactions that are likely to read or modify the same set of rows using at least one CURRENT VALIDTIME temporal SQL statement. Run these applications and transactions only sequentially, one after the other.
- Apply table level locks preemptively on temporal tables that are to be modified. These locks must be applied at the beginning of the transaction, which requires a BT/ET transaction or an ANSI transaction that uses a "LOCKING TABLE FOR WRITE" qualifier before any non-locking SQL is issued.

### Note:

It is not sufficient for the operation itself to apply a table-level lock, because the timestamp value for qualification may be earlier than the actual acquisition of the lock on the temporal table.

- Use the SEQUENCED VALIDTIME temporal qualifier with an explicit PA in the modification SQL rather than CURRENT VALIDTIME:

```
SEQUENCED VALIDTIME PERIOD (TEMPORAL_DATE/TEMPORAL_TIMESTAMP, UNTIL_CHANGED)
```

Be aware of the following restrictions on this technique:

- A variable PA cannot be specified at a session level. Therefore, the SEQUENCED VALIDTIME qualifier with PA must be mentioned at the statement level. This can require modifications to applications that interact with Vantage.
- Only equality inner joins are supported with the SEQUENCED VALIDTIME qualifier. Therefore applications that use other forms of joins must use one of the other options for avoiding concurrency issues.

## Special Case: Modifying the Same Row

Vantage can detect and abort the special case where concurrent CURRENT temporal transactions attempt to modify the same row of a temporal table at the same time. This capability must be enabled by Teradata Support. If this capability meets the needs of your situation, contact your Teradata Support representative.

## Related Information

| For more information on...            | See...                                                                         |
|---------------------------------------|--------------------------------------------------------------------------------|
| Transaction isolation levels and ACID | <i>Teradata Vantage™ - SQL Request and Transaction Processing</i> , B035-1142. |
| Timestamping temporal transactions    | <a href="#">Timestamping</a> .                                                 |

# Enforcing and Validating Temporal Referential Constraints

Because temporal referential constraints are all “soft RI”, meaning they are not enforced by Vantage, the responsibility for ensuring or validating the referential integrity is yours alone.

From the aspect of integrity assurance, the best way to guarantee the referential integrity of a table without taking advantage of a declarative standard or batch referential constraint is to use a procedural constraint such as a set of triggers to handle inserts, updates, and deletions to the tables in the relationship.

For example, you might want to create DELETE/UPDATE triggers on parent tables, and INSERT/UPDATE triggers on child tables to enforce referential integrity. The following example shows how an UPDATE trigger can be defined to enforce SEQUENCED referential integrity:

```
REPLACE TRIGGER trg_ri_validator
AFTER SEQUENCED VALIDTIME UPDATE OF column_of_interest ON child_table
REFERENCING NEW_TABLE as New1
FOR EACH STATEMENT
BEGIN ATOMIC
(
ABORT 'RI Violation'
FROM
(NONSEQUENCED VALIDTIME
 SELECT foreign_key_column
 FROM child_table
 WHERE foreign_key_column IS NOT NULL
 AND foreign_key_column NOT IN
 (SELECT parent_table.primary_key_column FROM parent_table
 WHERE parent_table.validtime_column
 CONTAINS child_table.validtime_column
 AND child_table.foreign_key_column = parent_table.primary_key_column)
) derived_table_name;
) END;
```

For more information on the CREATE and REPLACE TRIGGER statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

The reasons for preferring declarative constraints over procedural constraints are described briefly in *Teradata Vantage™ - Database Design*, B035-1094. There is the additional likelihood that actively firing triggers will have a greater negative effect on system performance than the simple declarative constraint they are intended to replace.

If you decide not to enforce any form of referential integrity constraint, then you are strongly advised to enforce a set of validation procedures that can detect when and where referential integrity violations occur.

The following are suggested queries that can be used to validate temporal referential constraints in a child table.

## Examples

```

/*****
*
* PURPOSE:
* =====
* Suggested queries to validate a Temporal RI on a child table.
*
* BACKGROUND:
* =====
* Multiple variants of RI can be defined with temporal tables.
*
* When a parent table is temporal, there can be multiple rows in the
* parent that contain a given parent key value with non-overlapping
* temporal column value. Presence of a temporal column causes rows
* to be tracked, so when a non-parent key column is modified
* in the parent, the system tracks history. This occurs in TransactionTime
* dimension. The same occurs for ValidTime dimension when modifications
* are made using CURRENT or SEQUENCED VALIDTIME qualifiers.
*
* Note this example:
*
* create multiset table tpar (
* c1 int,
* c2 int,
* pk int not null sequenced validtime unique
* vt period(timestamp with time zone) as validtime not null,
* tt period(timestamp with time zone) not null as transactiontime
*)
* primary index(c1)
*
* Assume row in parent is
* (1, 1, 1, (t1-until_changed),(t1-until_closed))
* on t3, assume c2 got incremented by 1.
*
* Rows in the parent table then are :
* (1, 1, 1, (t1-until_changed),(t1-t3))
* (1, 1, 1, (t1-t3),(t3-until_closed))
* (1, 2, 1, (t3-until_changed),(t3-until_closed))
*
* As noted, the pk value 1 is valid from t1-until_changed but this value
* is split in the two open rows above. Therefore, when validating whether a
* given child row's time value is present in the parent's VT or TT value,
* the rows must be normalized before comparing.
*
* The queries below use the TD_NORMALIZE_MEET table function.
*
* DESCRIPTION:
* =====
*
* For each of the variant of RI, a separate query (or set of queries)
* is provided to validate the RI constraint. If query returns rows, then
* it implies that that the child table does not satisfy the constraints.
* Please follow the instructions below to clean up the child table
* FK refers to the columns (excluding the temporal columns or the
* TRC column if the child table does not support ValidTime) on which the
* RI constraint is defined.
*

```

```
*
* 1) If FK is not present in the parent, then either update
* the child to point to correct FK column value or delete
* the row in the child
* 2) If FK is present in the parent, but the child's
* date/time column (date/timestamp or VT or TT column)
* does not satisfy the corresponding temporal RI
* property in the parent's VT or TT, then fix the
* portions of the child column to satisfy the RI
* constraint or delete the child row
* 3) If parent and child table's VT column's data types are
* different(ex Period(date), period(timestamp) etc.),
* Then use the cast function in the queries
* appropriately.
* For Ex: In the below queries the contains condition
* will be written by using cast function as follows.
* (cast(tblf.vtp as period(date)) contains t2chld.vt)
*
*
* GLOSSARY
* =====
* NT - Nontemporal
* VT - Validtime
* TT - Transactiontime
* BiT - BiTemporal
* TRC - Temporal Relationship Constraints
* CRI - Current Referential Integrity
* SRI - Sequenced Referential Integrity
* NRI - Nonsequenced Referential Integrity
*
*
*****/
/*****
/***** NT child -- NT parent *****/
/*****
/* RI definition.
 foreign key(fk) references with no check option tpar(pk)
*/
SELECT fk
FROM tchld
WHERE fk NOT IN (
SELECT pk
FROM tpar)
AND
fk IS NOT NULL;
/*****
/***** NT child -- VT parent (TRC)*****/
/* The parent's VT must contain the child dt to unitl_changed *****/
/* when pk-fk match */
/*****
/* RI definition.
 foreign key(fk,dt) references with no check option tpar(pk,vtp)
*/
drop table nm_par;
CREATE MULTISSET TABLE nm_par ,NO FALLBACK ,
 NO BEFORE JOURNAL,
 NO AFTER JOURNAL,
 CHECKSUM = DEFAULT,
 DEFAULT MERGEBLOCKRATIO
(
 pk INTEGER,
 vtp PERIOD(DATE) as VALIDTIME)
```

```

PRIMARY INDEX (pk);
INSERT INTO nm_par (pk,vtp)
WITH SUBTBL(x,d) AS
(NONSEQUENCED VALIDTIME
SELECT pk,vt
FROM tpar)
SELECT *
FROM TABLE (
 TD_SYSNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x), SUBTBL.d)
 RETURNS (pk1 INTEGER, vtp1 PERIOD(DATE))
 HASH BY x LOCAL
 ORDER BY x,d) AS TBLF(pk,vtp);
SELECT *
FROM (
CURRENT VALIDTIME SELECT DISTINCT fk,dt
FROM nm_par LEFT OUTER JOIN tchld
ON fk=pk WHERE
 NOT(vtp CONTAINS period(dt, UNTIL_CHANGED))
 AND vtp IS NOT NULL
UNION

SELECT * FROM (NONSEQUENCED VALIDTIME
SELECT fk,tchld.dt
FROM tchld
WHERE fk NOT IN (
SELECT pk
FROM tpar)
AND
fk IS NOT NULL)DT)dt(x,y);
/*****
/***** NT child -- TT parent *****/
/***** child row must exist in the open rows of parent *****/
/*****/
/* RI definition.
 foreign key(fk) references with no check option tpar(pk)
*/
CURRENT TRANSACTIONTIME
SELECT fk
FROM tchld
WHERE fk NOT IN
(
SELECT pk
FROM tpar)
AND fk IS NOT NULL ;
/*****
/***** NT child -- BiT parent (TRC) *****/
/* The parent's VT must contain the child dt to unitl_changed */
/* when pk-fk match */
/*****/
/* RI definition.
 foreign key(fk,dt) references with no check option tpar(pk,vt)
*/
drop table nm_par;
CREATE MULTISET TABLE nm_par ,NO FALLBACK ,
 NO BEFORE JOURNAL,
 NO AFTER JOURNAL,
 CHECKSUM = DEFAULT,
 DEFAULT MERGEBLOCKRATIO
 (
 pk INTEGER,
 vtp PERIOD(DATE) as VALIDTIME)
PRIMARY INDEX (pk);
INSERT INTO nm_par (pk,vtp)
WITH SUBTBL(x,d) AS

```

```

(NONSEQUENCED VALIDTIME
AND CURRENT TRANSACTIONTIME
SELECT pk,vt
FROM tpar)
SELECT *
FROM TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x), SUBTBL.d)
 RETURNS (pk1 INTEGER, vtp1 PERIOD(DATE))
 HASH BY x LOCAL
 ORDER BY x,d) AS TBLF(pk,vtp);
SELECT *
FROM (
CURRENT VALIDTIME SELECT DISTINCT fk,dt
FROM nm_par LEFT OUTER JOIN tchld
ON fk=pk WHERE
 NOT(vtp CONTAINS period(dt, UNTIL_CHANGED))
 AND vtp IS NOT NULL
UNION

SELECT * FROM (NONSEQUENCED VALIDTIME
AND CURRENT TRANSACTIONTIME
SELECT fk,tchld.dt
FROM tchld
WHERE fk NOT IN (
SELECT pk
FROM tpar)
AND
fk IS NOT NULL)DT)dt(x,y);
/*****
/***** VT child -- NT parent *****/
/***** NRI in VT dimension *****/
/*****/
/* RI definition.
 nonsequenced validtime foreign key(fk) references with no check
 option tpar(pk)
*/
NONSEQUENCED VALIDTIME
SELECT fk,vt
FROM tchld
WHERE fk NOT IN
(
SELECT pk
FROM tpar)
AND fk IS NOT NULL ;
/*****
/***** VT child -- VT parent *****/
/*****/
/* CRI in VT dimension */
/* RI definition.
 current validtime foreign key(fk) references with no check option
 tpar(pk)
*/

WITH SUBTBL(x,d) AS
 (NONSEQUENCED VALIDTIME
SELECT pk,vt
FROM tpar
WHERE END(tpar.vt) >= temporal_date)
SELECT x (title 'Foreign Key Column'),y (title ' ValidTime Column')
FROM (
NONSEQUENCED VALIDTIME
SELECT fk (title 'Foreign Key Column'),vt (title ' ValidTime Column')
FROM TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x), SUBTBL.d)

```

```

RETURNS (pk1 INTEGER, vtp1 PERIOD(DATE))
 HASH BY x LOCAL
ORDER BY x,d) AS TBLF(pk,vtp)
, tchld
WHERE pk=fk
 AND
NOT (tblf.vtp CONTAINS period(
CASE WHEN
begin (tchld.vt) > temporal_date THEN
begin (tchld.vt)
ELSE temporal_date
END
END (tchld.vt)))
 AND END(tchld.vt) >= temporal_date
 AND END(tblf.vtp) >= temporal_date
 AND tchld.vt IS NOT NULL
UNION
SELECT * FROM (NONSEQUENCED VALIDTIME
SELECT fk,tchld.vt
FROM tchld
WHERE fk NOT IN
(
SELECT pk
FROM tpar
WHERE
END (tpar.vt) >= temporal_date
)
 AND fk IS NOT NULL

 AND END(tchld.vt) >= temporal_date)DT)dt(x,y);

/* SRI in VT dimension */
/* RI definition.
 sequenced validtime foreign key(fk) references with no check option
 tpar(pk)
*/

WITH SUBTBL(x,d) AS
 (NONSEQUENCED VALIDTIME
SELECT pk,vt
FROM tpar)
SELECT x (title 'Foreign Key Column'),y (title ' ValidTime Column')
FROM (
NONSEQUENCED VALIDTIME
SELECT fk,tchld.vt
FROM TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x), SUBTBL.d)
RETURNS (pk1 INTEGER, vtp1 PERIOD(DATE))
 HASH BY x LOCAL
ORDER BY x,d) AS TBLF(pk,vtp)
, tchld
WHERE pk=fk
 AND
NOT (tblf.vtp CONTAINS tchld.vt)
 AND tchld.vt IS NOT NULL
UNION
SELECT * FROM (NONSEQUENCED VALIDTIME
SELECT fk,tchld.vt
FROM tchld
WHERE fk NOT IN
(
SELECT pk
FROM tpar)

```

```

 AND fk IS NOT NULL)DT)dt(x,y);
/*****
/***** VT child -- Bit parent *****/
/*****
/* CRI in VT dimension */
/* RI definition.
 current validtime foreign key(fk) references with no check option
 tpar(pk)
*/

WITH SUBTBL(x,d) AS
 (NONSEQUENCED VALIDTIME
 AND CURRENT TRANSACTIONTIME
 SELECT pk,vt
 FROM tpar
 WHERE END(tpar.vt) >= temporal_date)
SELECT x (title 'Foreign Key Column'),y (title ' ValidTime Column')
FROM (
 NONSEQUENCED VALIDTIME
 SELECT fk, tchld.vt
 FROM TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x), SUBTBL.d)
 RETURNS (pk1 INTEGER, vtp1 PERIOD(DATE))
 HASH BY x LOCAL
 ORDER BY x,d) AS TBLF(pk,vtp)
 , tchld
 WHERE pk=fk
 AND
 NOT (tblf.vtp CONTAINS period(
 CASE WHEN
 begin (tchld.vt) > temporal_date THEN
 begin (tchld.vt)
 ELSE temporal_date
 END
 END (tchld.vt)))
 AND END(tchld.vt) >= temporal_date
 AND END(tblf.vtp) >= temporal_date
 AND tchld.vt IS NOT NULL
 UNION
 SELECT * FROM (NONSEQUENCED VALIDTIME
 SELECT fk,tchld.vt
 FROM tchld
 WHERE fk NOT IN
 (
 SELECT pk
 FROM tpar
 WHERE
 END (tpar.vt) >= temporal_date
)
 AND fk IS NOT NULL
 AND END(tchld.vt) >= temporal_date)DT)dt(x,y);

/* SRI in VT dimension */
/* RI definition.
 sequenced validtime foreign key(fk) references with no check option
 tpar(pk)
*/
WITH SUBTBL(x,d) AS
 (NONSEQUENCED VALIDTIME
 AND CURRENT TRANSACTIONTIME
 SELECT pk,vt
 FROM tpar)
SELECT x (title 'Foreign Key Column'),y (title ' ValidTime Column')
FROM (

```

```

NONSEQUENCED VALIDTIME
SELECT fk,tchld.vt
FROM TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x), SUBTBL.d)
 RETURNS (pk1 INTEGER, vtp1 PERIOD(DATE))
 HASH BY x LOCAL
 ORDER BY x,d) AS TBLF(pk,vtp)
, tchld
WHERE pk=fk
AND
NOT (tblf.vtp CONTAINS tchld.vt)
AND tchld.vt IS NOT NULL
UNION
SELECT * FROM (NONSEQUENCED VALIDTIME
SELECT fk,tchld.vt
FROM tchld
WHERE fk NOT IN
(
 SELECT pk
 FROM tpar)
 AND fk IS NOT NULL)DT)dt(x,y);
/*****
/***** TT child -- NT parent *****/
/***** NRI in TT dimension *****/
/*****/
/* RI definition.
 nonsequenced transactiontime foreign key(fk) references
 with no check option tpar(pk)
*/
NONSEQUENCED TRANSACTIONTIME
SELECT fk (title 'Foreign Key Column'),tt (title 'TransactionTime Column')
FROM tchld
WHERE fk NOT IN
(
 SELECT pk
 FROM tpar)
 AND fk IS NOT NULL;
/*****
/***** TT child -- VT parent (TRC) *****/
/**** NONSEQUENCED TRANSACTIONTIME RI with TRC on parent't VT *****/
/* The parent's VT must contain the child dt to until_changed */
/* when pk-fk match */
/*****/
/* RI definition.
 nonsequenced transactiontime foreign key(fk,dt) references
 with no check option tpar(pk,vt)
*/

drop table nm_par;
CREATE MULTISET TABLE nm_par ,NO FALLBACK ,
 NO BEFORE JOURNAL,
 NO AFTER JOURNAL,
 CHECKSUM = DEFAULT,
 DEFAULT MERGEBLOCKRATIO
 (
 pk INTEGER,
 vtp PERIOD(DATE) as VALIDTIME)
PRIMARY INDEX (pk);
INSERT INTO nm_par (pk,vtp)
WITH SUBTBL(x,d) AS
(NONSEQUENCED VALIDTIME
SELECT pk,vt
FROM tpar)
SELECT *

```

```

FROM TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x), SUBTBL.d)
 RETURNS (pk1 INTEGER, vtp1 PERIOD(DATE))
 HASH BY x LOCAL
 ORDER BY x,d) AS TBLF(pk,vtp);
SELECT *
FROM (
 CURRENT VALIDTIME SELECT DISTINCT fk,dt
 FROM nm_par LEFT OUTER JOIN tchld
 ON fk=pk WHERE
 NOT(vtp CONTAINS period(dt, UNTIL_CHANGED))
 AND vtp IS NOT NULL
 UNION

 SELECT * FROM (NONSEQUENCED TRANSACTIONTIME
 SELECT fk,tchld.dt
 FROM tchld
 WHERE fk NOT IN (
 SELECT pk
 FROM tpar)
 AND
 fk IS NOT NULL)DT)dt(x,y);
/*****
/***** TT child -- TT parent *****/
/*****
/* CRI in TT dimension */
/* RI definition.
 current transactiontime foreign key(fk) references
 with no check option tpar(pk)
*/
/* CURRENT rows in child should have equivalent PK in parent's open rows
 for each matching fk-pk rows.
*/
CURRENT TRANSACTIONTIME
SELECT fk (title 'Foreign Key Column'),tt (title 'TransactionTime Column')
FROM tchld
WHERE fk NOT IN (
 SELECT pk
 FROM tpar)
 AND
 fk IS NOT NULL;

/* SRI in TT dimension */
/* RI definition.
 sequenced transactiontime foreign key(fk) references
 with no check option tpar(pk)
*/
/* child row's TT must be contained in the normalized TT of the parent
 for each of the matching fk-pk rows.
*/

WITH SUBTBL(x,d) AS
 (NONSEQUENCED TRANSACTIONTIME
 SELECT pk,tt
 FROM tpar)
SELECT *
FROM (
 NONSEQUENCED TRANSACTIONTIME
 SELECT fk,tchld.tt
 FROM TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x), SUBTBL.d)

 RETURNS (pk1 INTEGER, ttp1 PERIOD(timestamp
 with time zone))

```

```

 HASH BY x LOCAL
ORDER BY x,d) AS TBLF(pk,ttp)
, tchld
WHERE pk=fk
AND
NOT (tblf.ttp CONTAINS tchld.tt)
AND tchld.tt IS NOT NULL
UNION
SELECT * FROM (NONSEQUENCED TRANSACTIONTIME
SELECT fk,tchld.tt
FROM tchld
WHERE fk NOT IN
(
SELECT pk
FROM tpar)
AND fk IS NOT NULL)DT)dt(x,y);
/*****
/***** TT child -- BiT parent(TRC) *****/
/*****
/* CURRENT TRANSACTIONTIME RI AND TRC on parent */
/* RI definition.
current transactiontime foreign key(fk,dt) references
with no check option tpar(pk,vtp)
*/
drop table nm_par;
CREATE MULTISET TABLE nm_par ,NO FALLBACK ,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT,
DEFAULT MERGEBLOCKRATIO
(
pk INTEGER,
vtp PERIOD(DATE) as VALIDTIME)
PRIMARY INDEX (pk);
INSERT INTO nm_par (pk,vtp)
WITH SUBTBL(x,d) AS
(NONSEQUENCED VALIDTIME
AND CURRENT TRANSACTIONTIME
SELECT pk,vtp
FROM tpar)
SELECT *
FROM TABLE (
TD_SYSNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x), SUBTBL.d)
RETURNS (pk1 INTEGER, vtp1 PERIOD(DATE))
HASH BY x LOCAL
ORDER BY x,d) AS TBLF(pk,vtp);
SELECT *
FROM (
CURRENT VALIDTIME SELECT DISTINCT fk,dt
FROM nm_par LEFT OUTER JOIN tchld
ON fk=pk WHERE
NOT(vtp CONTAINS period(dt, UNTIL_CHANGED))
AND vtp IS NOT NULL
UNION
SELECT * FROM(NONSEQUENCED VALIDTIME
AND CURRENT TRANSACTIONTIME
SELECT fk,tchld.dt
FROM tchld
WHERE fk NOT IN (
SELECT pk
FROM tpar)
AND
fk IS NOT NULL)DT)dt(x,y);

```

```

/* SEQUENCED TRANSACTIONTIME RI AND TRC on parent */
/* RI definition.
 sequenced transactiontime foreign key(fk,dt) references
 with no check option tpar(pk,vt)
*/
/* child row's TT must be contained in the normalized TT of the parent
 for each of the matching fk-pk rows */
/* normalize the VT on open rows in TT AND PK column -
 with this result, normalize in TT diemntion on matching PK AND VT
*/
drop table tmptbl;
CREATE MULTISET TABLE tmptbl
AS (
WITH SUBTBL(x,d,t) AS
 (NONSEQUENCED VALIDTIME
 AND CURRENT TRANSACTIONTIME
 SELECT pk,vt, tt
 FROM tpar)
SELECT *
FROM (TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x, SUBTBL.t),
 SUBTBL.d)
 RETURNS (pk1 INTEGER, ttp1 PERIOD(timestamp
with time zone), vtp1 period(date))
 HASH BY x, t LOCAL
 ORDER BY x,t,d)AS TBLF(pk,tt, vt))
) with data ;
drop table tmptbl1;
CREATE MULTISET TABLE tmptbl1
AS (
NONSEQUENCED VALIDTIME
 AND NONSEQUENCED TRANSACTIONTIME
 SELECT pk, vt, tt
 FROM tpar
 WHERE END(tt) is not until_closed
 UNION
 SELECT pk, vt, tt
 FROM tmptbl
) with data ;
WITH SUBTBL(x,v, d) AS
(
 SEL * FROM tmptbl1
)
SELECT *
FROM (
NONSEQUENCED TRANSACTIONTIME
 SELECT fk, tchld.dt, tchld.tt
 FROM TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x, SUBTBL.v),
 SUBTBL.d)
 RETURNS (pk1 INTEGER, vtp1 period(date), ttp1 PERIOD(timestamp
with time zone))
 HASH BY x, v LOCAL
 ORDER BY x,v,d)AS TBLF(pk,vtp, ttp)
 , tchld
 WHERE pk=fk
 AND
 ((NOT (tblf.ttp CONTAINS tchld.tt))
 OR (NOT (tblf.vtp CONTAINS period(tchld.dt, UNTIL_CHANGED)))
)
)
 UNION
 SELECT * FROM (NONSEQUENCED TRANSACTIONTIME
 SELECT fk,tchld.dt, tchld.tt
 FROM tchld

```

```

WHERE fk NOT IN
(
 SELECT pk
 FROM tpar)
 AND fk IS NOT NULL)DT)dt order by 1, 2 ;
/*****
/***** BiT child -- NT parent *****/
/*****
/* NRI in both the dimensions */
/* RI definition.
 nonsequenced validtime and nonsequenced transactiontime foreign
 key(fk) references with no check option tpar(pk)
*/

NONSEQUENCED VALIDTIME
 AND NONSEQUENCED TRANSACTIONTIME
SELECT fk,vt,tt
FROM tchld
WHERE fk NOT IN
(
 SELECT pk
 FROM tpar)
 AND fk IS NOT NULL ;
/*****
/***** BiT child -- VT parent *****/
/*****
/* CRI in VT dimension NRI in TT dimension */
/* RI definition.
 current validtime and nonsequenced transactiontime foreign key(fk)
 references with no check option tpar(pk)
*/

WITH SUBTBL(x,d) AS
 (NONSEQUENCED VALIDTIME
 SELECT pk,vt
 FROM tpar
 WHERE END(tpar.vt) >= temporal_date)
SELECT *
FROM (
 NONSEQUENCED VALIDTIME
 AND NONSEQUENCED TRANSACTIONTIME
 SELECT fk, tchld.vt
 FROM TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x), SUBTBL.d)
 RETURNS (pk1 INTEGER, vtp1 PERIOD(DATE))
 HASH BY x LOCAL
 ORDER BY x,d) AS TBLF(pk,vtp)
 , tchld
 WHERE pk=fk
 AND
 NOT (tblf.vtp CONTAINS period(
 CASE WHEN
 begin (tchld.vt) > temporal_date THEN
 begin (tchld.vt)
 ELSE temporal_date
 END ,
 END (tchld.vt)))
 AND END(tchld.vt) >= temporal_date
 AND END(tblf.vtp) >= temporal_date
 AND tchld.vt IS NOT NULL
 UNION
 SELECT * FROM (NONSEQUENCED VALIDTIME
 AND NONSEQUENCED TRANSACTIONTIME
 SELECT fk,tchld.vt
 FROM tchld

```

```

WHERE fk NOT IN
(
SELECT pk
FROM tpar
WHERE
END (tpar.vt) >= temporal_date
)
AND fk IS NOT NULL
AND END(tchld.vt) >= temporal_date)DT)dt(x,y);
/* SRI in VT dimension NRI in TT dimension */
/* RI definition.
 sequenced validtime and nonsequenced transactiontime foreign key(fk)
 references with no check option tpar(pk)
*/

WITH SUBTBL(x,d) AS
 (NONSEQUENCED VALIDTIME
SELECT pk,vt
FROM tpar)
SELECT *
FROM (
NONSEQUENCED VALIDTIME
AND NONSEQUENCED TRANSACTIONTIME
SELECT fk,tchld.vt
FROM TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x), SUBTBL.d)

RETURNS (pk1 INTEGER, vtp1 PERIOD(DATE))
 HASH BY x LOCAL
ORDER BY x,d) AS TBLF(pk,vtp)
, tchld
WHERE pk=fk
AND
NOT (tblf.vtp CONTAINS tchld.vt)
AND tchld.vt IS NOT NULL
UNION
SELECT * FROM (NONSEQUENCED VALIDTIME
AND NONSEQUENCED TRANSACTIONTIME
SELECT fk,tchld.vt
FROM tchld
WHERE fk NOT IN
(
SELECT pk
FROM tpar)
AND fk IS NOT NULL)DT)dt(x,y) order by 1, 2 ;
/***** BiT child -- TT parent *****/
/* NRI in VT dimension CRI in TT dimension */
/* RI definition.
 nonsequenced validtime and current transactiontime foreign key(fk)
 references with no check option tpar(pk)
*/
/* CURRENT rows in child should have equivalent PK in parent's open rows
 for each matching fk-pk rows */
NONSEQUENCED VALIDTIME
AND CURRENT TRANSACTIONTIME
SELECT fk,vt,tt
FROM tchld
WHERE fk NOT IN (
SELECT pk
FROM tpar)
AND
fk IS NOT NULL;

```

```

/* NRI in VT dimension SRI in TT dimension */
/* RI definition.
 nonsequenced validtime and sequenced transactiontime foreign key(fk)
 references with no check option tpar(pk)
*/
/* child row's TT must be contained in the normalized TT of the parent
 for each of the matching fk-pk rows */
WITH SUBTBL(x,d) AS
 (NONSEQUENCED TRANSACTIONTIME
 SELECT pk,tt
 FROM tpar)
SELECT *
FROM (
 NONSEQUENCED VALIDTIME
 AND NONSEQUENCED TRANSACTIONTIME
 SELECT DISTINCT fk,tchld.tt
 FROM TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x), SUBTBL.d)
 RETURNS (pk1 INTEGER, ttp1 PERIOD(timestamp
 with time zone))
 HASH BY x LOCAL
 ORDER BY x,d) AS TBLF(pk,ttp)
 , tchld
 WHERE pk=fk
 AND
 NOT (tblf.ttp CONTAINS tchld.tt)
 AND tchld.tt IS NOT NULL
 UNION
 SELECT * FROM (NONSEQUENCED VALIDTIME
 AND NONSEQUENCED TRANSACTIONTIME
 SELECT fk,tchld.tt
 FROM tchld
 WHERE fk NOT IN
 (
 SELECT pk
 FROM tpar)
 AND fk IS NOT NULL)DT)dt(x,y);
/*****
/***** BiT child -- BiT parent *****/
/*****
/* CRI in VT dimension, CRI TT dimension */
/* RI definition.
 current validtime and current transactiontime foreign key(fk)
 references with no check option tpar(pk)
*/

WITH SUBTBL(x,d) AS
 (NONSEQUENCED VALIDTIME
 AND CURRENT TRANSACTIONTIME
 SELECT pk,vt
 FROM tpar
 WHERE END(tpar.vt) >= temporal_date)
SELECT *
FROM (
 NONSEQUENCED VALIDTIME
 AND CURRENT TRANSACTIONTIME
 SELECT fk, tchld.vt
 FROM TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x), SUBTBL.d)
 RETURNS (pk1 INTEGER, vtp1 PERIOD(DATE))
 HASH BY x LOCAL
 ORDER BY x,d) AS TBLF(pk,vtp)
 , tchld

```

```

WHERE pk=fk
AND
NOT (tblf.vtp CONTAINS period(
CASE WHEN
begin (tchld.vt) > temporal_date THEN
begin (tchld.vt)
ELSE temporal_date
END
END (tchld.vt)))
AND END(tchld.vt) >= temporal_date
AND END(tblf.vtp) >= temporal_date
AND tchld.vt IS NOT NULL
UNION
SELECT * FROM (NONSEQUENCED VALIDTIME
AND CURRENT TRANSACTIONTIME
SELECT fk,tchld.vt
FROM tchld
WHERE fk NOT IN
(
SELECT pk
FROM tpar
WHERE
END (tpar.vt) >= temporal_date
)
AND fk IS NOT NULL
AND END(tchld.vt) >= temporal_date)DT)dt(x,y);

/* SRI in VT dimension, CRI TT dimension */
/* RI definition.
sequenced validtime and current transactiontime foreign key(fk)
references with no check option tpar(pk)
*/
WITH SUBTBL(x,d) AS
(NONSEQUENCED VALIDTIME
AND CURRENT TRANSACTIONTIME
SELECT pk,vt
FROM tpar)
SELECT *
FROM (
NONSEQUENCED VALIDTIME
AND CURRENT TRANSACTIONTIME
SELECT fk,tchld.vt
FROM TABLE (
TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x), SUBTBL.d)
RETURNS (pk1 INTEGER, vtp1 PERIOD(DATE))
HASH BY x LOCAL
ORDER BY x,d) AS TBLF(pk,vtp)
, tchld
WHERE pk=fk
AND
NOT (tblf.vtp CONTAINS tchld.vt)
AND tchld.vt IS NOT NULL
UNION
SELECT * FROM (NONSEQUENCED VALIDTIME
AND CURRENT TRANSACTIONTIME
SELECT fk,tchld.vt
FROM tchld
WHERE fk NOT IN
(
SELECT pk
FROM tpar)
AND fk IS NOT NULL)DT)dt(x,y);
/*****
* CRI in VT dimension, SRI TT dimension

```

```

*****/
/* RI definition.
 current validtime and sequenced transactiontime foreign key(fk)
 references with no check option tpar(pk)
*/
drop table tmpTbl;
CREATE MULTISET TABLE tmpTbl
AS (
WITH SUBTBL(x,d,t) AS
 (NONSEQUENCED VALIDTIME
 AND CURRENT TRANSACTIONTIME
 SELECT pk,vt, tt
 FROM tpar)
SELECT *
FROM (TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x, SUBTBL.t),
 SUBTBL.d)
 RETURNS (pk1 INTEGER, ttp1 PERIOD(timestamp
with time zone), vtp1 period(date))
 HASH BY x, t LOCAL
 ORDER BY x,t,d)AS TBLF(pk,tt, vt))
) with data ;
drop table tmpTbl1;
CREATE MULTISET TABLE tmpTbl1
AS (
NONSEQUENCED VALIDTIME
 AND NONSEQUENCED TRANSACTIONTIME
 SELECT pk, vt, tt
 FROM tpar
 WHERE END(tt) is not until_closed
 UNION
 SELECT pk, vt, tt
 FROM tmpTbl
) with data ;
WITH SUBTBL(x,v, d) AS
(
 SEL * FROM tmpTbl1
)
SELECT *
FROM (
NONSEQUENCED VALIDTIME AND NONSEQUENCED TRANSACTIONTIME
 SELECT DISTINCT fk, tchld.vt, tchld.tt
 FROM TABLE (
 TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x, SUBTBL.v),
 SUBTBL.d)
 RETURNS (pk1 INTEGER, vtp1 period(date), ttp1 PERIOD(timestamp
with time zone))
 HASH BY x, v LOCAL
 ORDER BY x,v,d)AS TBLF(pk,vtp, ttp)
 , tchld
 WHERE pk=fk
 AND
 ((NOT (tblf.ttp CONTAINS tchld.tt))
 OR (NOT (tblf.vtp CONTAINS PERIOD(
CASE WHEN
begin (tchld.vt) > temporal_date THEN
begin (tchld.vt)
ELSE temporal_date
END
END (tchld.vt))))
)
 AND END(tchld.vt) >= TEMPORAL_DATE
 UNION
 SELECT * FROM (NONSEQUENCED VALIDTIME AND NONSEQUENCED TRANSACTIONTIME

```

```

SELECT fk,tchld.vt, tchld.tt
FROM tchld
WHERE fk NOT IN
(
SELECT pk
FROM tpar)
AND fk IS NOT NULL)DT)dt order by 1, 2 ;

/*****
* SRI in VT dimension, SRI TT dimension
*****/
/* RI definition.
sequenced validtime and sequenced transactiontime foreign key(fk)
references with no check option tpar(pk)
*/
drop table tmpTbl;
CREATE MULTISET TABLE tmpTbl
AS (
WITH SUBTBL(x,d,t) AS
(NONSEQUENCED VALIDTIME
AND CURRENT TRANSACTIONTIME
SELECT pk,vt, tt
FROM tpar)
SELECT *
FROM (TABLE (
TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x, SUBTBL.t),
SUBTBL.d)
RETURNS (pk1 INTEGER, ttp1 PERIOD(timestamp
with time zone), vtp1 period(date))
HASH BY x, t LOCAL
ORDER BY x,t,d)AS TBLF(pk,tt, vt))
) with data ;
drop table tmpTbl1;
CREATE MULTISET TABLE tmpTbl1
AS (
NONSEQUENCED VALIDTIME
AND NONSEQUENCED TRANSACTIONTIME
SELECT pk, vt, tt
FROM tpar
WHERE END(tt) is not until_closed
UNION
SELECT pk, vt, tt
FROM tmpTbl
) with data ;
WITH SUBTBL(x,v, d) AS
(
SEL * FROM tmpTbl1
)
SELECT *
FROM (
NONSEQUENCED VALIDTIME AND NONSEQUENCED TRANSACTIONTIME
SELECT DISTINCT fk, tchld.vt, tchld.tt
FROM TABLE (
TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(SUBTBL.x, SUBTBL.v),
SUBTBL.d)
RETURNS (pk1 INTEGER, vtp1 period(date), ttp1 PERIOD(timestamp
with time zone))
HASH BY x, v LOCAL
ORDER BY x,v,d)AS TBLF(pk,vtp, ttp)
, tchld
WHERE pk=fk
AND
((NOT (tblf.ttp CONTAINS tchld.tt))
OR (NOT (tblf.vtp CONTAINS tchld.vt)))

```

```
)
UNION
SELECT * FROM (NONSEQUENCED VALIDTIME AND NONSEQUENCED TRANSACTIONTIME
SELECT fk,tchld.vt, tchld.tt
FROM tchld
WHERE fk NOT IN
(
SELECT pk
FROM tpar)
AND fk IS NOT NULL)DT)dt order by 1, 2 ;
```

# ANSI Temporal Tables

This section describes the major differences between Teradata's proprietary implementation of temporal tables and syntax, which is described in this document, and Teradata's implementation of ANSI/ISO compliant temporal tables, which is described in *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186. It also describes three methods that can be used to convert transaction-time tables into ANSI-compliant system-versioned system-time tables.

## ANSI Temporal and Teradata Temporal

Teradata introduced support for creating and manipulating temporal tables before an ANSI/ISO standard had been developed. Consequently, the original Teradata temporal tables and SQL syntax do not conform to the ANSI standard. When ANSI/ISO standards for temporal tables were approved, Teradata developed new, ANSI-compliant temporal tables and SQL syntax.

Both the ANSI compliant and original non-ANSI compliant versions of temporal tables are available in Vantage.

Use the following comparison to help determine which version of temporal tables best meets your requirements.

### Note:

Most temporal qualifiers and query syntax that is used with Teradata's proprietary temporal tables can be used also on ANSI temporal tables.

| ANSI Temporal Tables and Syntax                                                                                                                                                           | Teradata Temporal Tables and Syntax                                                                                                                                                                                                     |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ANSI/ISO compliant, with minor variations for valid-time (application-time) table definitions and Teradata extensions to allow temporal queries of valid-time tables.                     | Not ANSI/ISO compliant.                                                                                                                                                                                                                 |
| Temporal columns of temporal tables are derived dynamically from physical DateTime columns that store the beginning and ending bound values of the derived periods.                       | Temporal columns may be derived periods or may use Teradata Period data types, that allow a column to represent a duration. (Use of Period data types is allowed, but not recommended.)                                                 |
| Start and end columns that constitute temporal derived period columns are always implicitly projected in SELECT * queries.                                                                | Temporal columns, or start and end columns that constitute temporal derived period columns may or may not be projected, depending on the temporal query qualifier or the temporal qualifier that is set as the default for the session. |
| In a system-time table, the component begin and end timestamp columns of the SYSTEM_TIME derived period column can only be modified if system versioning is first removed from the table. | In a transaction-time table (analogous to an ANSI system-time table), the special NONTEMPORAL privilege and qualifier allow modification of transaction-time column values.                                                             |

| ANSI Temporal Tables and Syntax                                                                                                                    | Teradata Temporal Tables and Syntax                                                                                                    |
|----------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| Removing system versioning from a system-time table physically deletes all closed rows from the table, and renders the table a non-temporal table. |                                                                                                                                        |
| Default SELECT behavior is to qualify all rows of valid-time tables. Default behavior cannot be changed with session qualifiers.                   | Default SELECT behavior is to qualify only current rows of valid-time tables. Default behavior can be changed with session qualifiers. |

## Converting Transaction-Time Tables to ANSI System-Versioned System-Time Tables

Teradata valid-time tables qualify as ANSI application-time temporal tables, if they are defined using a valid-time derived period column and have no temporal constraints. These tables are ANSI compliant without modifications.

Teradata transaction-time tables are analogous to ANSI system-versioned system-time temporal tables, but must be converted to system-versioned system-time tables in order to be used with ANSI-compliant temporal SQL. For sites that have implemented transaction-time tables, this section provides three methods to convert transaction-time tables to system-time tables. Teradata recommends contacting your Teradata representative for help with this process.

Although Teradata's original temporal SQL that is used to qualify temporal queries and modifications does operate on Teradata's ANSI temporal tables, it is not ANSI-compliant SQL.

### Note:

In order to use ANSI temporal tables on systems that used temporal tables prior to Teradata Database 15.0, the session temporal qualifier must be set to ANSIQUALIFIER. This is normally set appropriately by Teradata personnel. ANSIQUALIFIER changes the SQL behavior with respect to default temporal qualifiers such that unqualified queries and modifications of valid-time tables act as nonsequenced.

You can check the session temporal qualifier setting by looking at the Temporal Qualifier field of the output of the HELP SESSION statement. For more information on HELP SESSION, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

For more information on session temporal qualifiers, see [SET SESSION \(Session Temporal Qualifiers\)](#).

## Method 1: Alter Existing Transaction-Time Table

This method:

- requires that the executor have the NONTEMPORAL privilege in the database, and that the database be enabled to recognize that privilege. Read more about the NONTEMPORAL privilege in *Teradata Vantage™ - Temporal Table Support*, B035-1182.

- cannot be used if the transaction-time table is row partitioned on the beginning or ending bound of the transaction-time.
  - is not recommended for large, column-partitioned tables because for these tables the update operation in Step 4 can be very resource-intensive and time-consuming.
1. Note all the constraints on the transaction-time table.
  2. Drop all the constraints from the transaction-time table.
  3. Use `NONTEMPORAL ALTER TABLE` to add two new columns of type `TIMESTAMP(6) WITH TIME ZONE`. For the purposes of this procedure, assume the columns are named `sys_start` and `sys_end`. These will hold the beginning and ending bound values of the new `SYSTEM_TIME` derived period column.
  4. Use `NONTEMPORAL UPDATE` to populate the new columns with the start and end values of the existing transaction-time columns or derived period column.
  5. Use `NONTEMPORAL ALTER TABLE` to drop the existing transaction-time column. Use the `WITHOUT DELETE` option to preserve the historical closed rows, which would otherwise be deleted automatically when you drop the transaction-time column:

```
ALTER TABLE transaction_time_table_name
 DROP transaction_time_column WITHOUT DELETE
```

6. Use `ALTER TABLE` to create the `SYSTEM_TIME` derived period column and to add attributes to the set the `sys_start` and `sys_end` columns in the same `ALTER TABLE` statement:

```
ALTER TABLE transaction_time_table_name
 ADD PERIOD FOR SYSTEM_TIME(sys_start,sys_end)
 ADD sys_start TIMESTAMP(6) WITH TIME ZONE NOT NULL
 GENERATED ALWAYS AS ROW START
 add sys_end TIMESTAMP(6) WITH TIME ZONE NOT NULL
 GENERATED ALWAYS AS ROW END;
```

7. Add system versioning to make the new table an ANSI system-time temporal table:

```
ALTER TABLE transaction_time_table_name
 ADD SYSTEM VERSIONING;
```

8. Recreate all the constraints that were dropped in step 2. Note that ANSI constraints behave as `NONSEQUENCED` constraints.

## Method 2: INSERT ... SELECT to New Table When Transaction-Time Column is Derived Period

This method:

- requires that the executor have the NONTEMPORAL privilege in the database, and that the database be enabled to recognize that privilege. For more information on the NONTEMPORAL privilege, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.
  - cannot be used if the transaction-time table is row partitioned on the beginning or ending bound of the transaction-time.
1. Note all the constraints on the transaction-time table.
  2. Drop all the constraints from the transaction-time table.
  3. Use NONTEMPORAL ALTER TABLE to drop the existing transaction-time column. Use the WITHOUT DELETE option to preserve the historical closed rows, which would otherwise be deleted automatically when you drop the transaction-time column:

```
ALTER TABLE transaction_time_table_name
 DROP transaction_time_column WITHOUT DELETE
```

4. Use ALTER TABLE to create the SYSTEM\_TIME derived period column and to add attributes to the set the sys\_start and sys\_end columns in the same ALTER TABLE statement:

```
ALTER TABLE new_table_name
 ADD PERIOD FOR SYSTEM_TIME(sys_start,sys_end)
 ADD sys_start TIMESTAMP(6) WITH TIME ZONE NOT NULL
 GENERATED ALWAYS AS ROW START

 ADD sys_end TIMESTAMP(6) WITH TIME ZONE NOT NULL
 GENERATED ALWAYS AS ROW END;
```

5. Add system versioning to make the new table an ANSI system-time temporal table:

```
ALTER TABLE new_table_name
 ADD SYSTEM VERSIONING;
```

6. Recreate all the constraints that were dropped in step 2. Note that ANSI constraints behave as NONSEQUENCED constraints.

## Method 3: INSERT ... SELECT to New Table When Transaction-Time Column is Period Data Type

This method:

- does not require the NONTEMPORAL privilege.
  - can be used on transaction-time tables that are row-partitioned on the beginning or ending bound of the transaction-time period.
1. Create a new table with columns that match the non-transaction-time columns of the existing table. Add two new TIMESTAMP(6) WITH TIME ZONE columns that will hold the beginning and ending bound

values for the ANSI system-time derived period column. For the purposes of this procedure, assume the columns are named `sys_start` and `sys_end`.

2. Use a `NONSEQUENCED INSERT ... SELECT` to copy the rows of the transaction-time table into the new table.
3. Use `ALTER TABLE` to create the `SYSTEM_TIME` derived period column and to add attributes to the set the `sys_start` and `sys_end` columns in the same `ALTER TABLE` statement:

```
ALTER TABLE new_table_name
 ADD PERIOD FOR SYSTEM_TIME(sys_start,sys_end)
 ADD sys_start TIMESTAMP(6) WITH TIME ZONE NOT NULL
 GENERATED ALWAYS AS ROW START
 add sys_end TIMESTAMP(6) WITH TIME ZONE NOT NULL
 GENERATED ALWAYS AS ROW END;
```

4. Note the constraints on the transaction-time table.
5. Drop the transaction-time table.
6. Rename the new table as the old table.
7. Add system versioning to make the new table an ANSI system-time temporal table:

```
ALTER TABLE new_table_name
 ADD SYSTEM VERSIONING;
```

8. Recreate all the constraints that were dropped in step 2. Note that ANSI constraints behave as `NONSEQUENCED` constraints.

# Additional Information

## Teradata Links

| Link                                                                                                    | Description                                                                                                                                                                                                                                                |
|---------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="https://docs.teradata.com/">https://docs.teradata.com/</a>                                     | Search Teradata Documentation, customize content to your needs, and download PDFs.<br>Customers: Log in to access Orange Books.                                                                                                                            |
| <a href="https://support.teradata.com">https://support.teradata.com</a>                                 | One-stop source for Teradata community support, software downloads, and product information.<br>Log in for customer access to: <ul style="list-style-type: none"><li>• Community support</li><li>• Software updates</li><li>• Knowledge articles</li></ul> |
| <a href="https://www.teradata.com/University/Overview">https://www.teradata.com/University/Overview</a> | Teradata education network                                                                                                                                                                                                                                 |
| <a href="https://support.teradata.com/community">https://support.teradata.com/community</a>             | Link to Teradata community                                                                                                                                                                                                                                 |