

Teradata Vantage™ - ANSI Temporal Table Support

Release 17.10

July 2021

Copyright and Trademarks

Copyright © 2014 - 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 ANSI Temporal Table Support	5
Changes and Additions	5
Temporal Database	5
The Need to Represent Time	5
Temporal Columns	7
Temporal Queries and Modifications	8
Chapter 2: ANSI Temporal Concepts	9
Derived Period Columns	9
System Time and Valid Time	9
UNTIL_CLOSED and UNTIL_CHANGED	12
Temporal Row Types	12
Inserting Rows into Temporal Tables	13
Querying Temporal Tables	14
Modifying Temporal Tables	14
Timestamping	17
ANSI Temporal and Teradata Temporal	17
Exceptions to the ANSI Standard	18
Chapter 3: Working With ANSI System-Time Tables	20
Creating ANSI System-Time Tables	20
Bulk Loading Data into Temporal Tables	30
Querying ANSI System-Time Tables	31
Modifying Rows in ANSI System-Time Tables	36
HELP and SHOW Statements	40
Cursors and ANSI System-Time Table Queries	41
Chapter 4: Working With ANSI Valid-Time Tables	42
Creating ANSI Valid-Time Tables	42
Querying ANSI Valid-Time Tables	56
Modifying Rows in ANSI Valid-Time Tables	61
HELP and SHOW Statements	69
Cursors and ANSI Valid-Time Table Queries	70
Chapter 5: Working With ANSI Bitemporal Tables	71
Creating ANSI Bitemporal Tables	71
Querying ANSI Bitemporal Tables	81
Modifying Rows in ANSI Bitemporal Tables	85

Chapter 6: Administration	94
System Clocks	94
Capacity Planning for Temporal Tables	94
Archiving Temporal Tables	96
Appendix A: How to Read Syntax	98
Appendix B: Converting Transaction-Time Tables into ANSI System-Time Tables	100
Appendix C: Additional Information	104

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

For information on Teradata's proprietary temporal tables, which have different capabilities than ANSI-compliant temporal tables, see *Teradata Vantage™ - Temporal Table Support*, B035-1182. For more information on the differences between the two temporal paradigms, see [ANSI Temporal and Teradata Temporal](#).

Note that most temporal qualifiers and query syntax that is used with Teradata's proprietary temporal tables can be used also on ANSI temporal tables.

Changes and Additions

Date	Description
July 2021	Minor edits.

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.

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, though 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 policies, represented by rows in the Policy table, became valid. One approach is to add a DATE type column to the Policy table called Start_Date. If the application also needs to know when policies are no longer valid, a corresponding End_Date column can be added.

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);
```

However, this introduces several complications. 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 on the original row now needs to be updated to reflect when the policy conditions were changed, rather than when the policy becomes invalid.

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 explicitly changing the Start_Date and End_Date columns. Queries become more complicated.

The mere presence of one or more DateTime type columns 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.

Vantage provides built-in support for ANSI-compatible temporal tables. Temporal variations of standard SQL statements let you create, alter, 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 columns and temporal statements facilitate creating applications that can represent information that changes over time.

Temporal Columns

Teradata provides temporal table support at the column level using *derived period columns*. A period is an anchored duration that represents a set of contiguous time granules within the duration. It has a beginning and ending bound, defined by two DateTime type columns in the table.

In a temporal table, the values from the beginning and ending bound columns are combined in a third, derived column that represents the period of time, or duration they delimit.

Now the application for the insurance company can create the Policy table with a derived period column to record the period during which each policy (row) is valid.

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

The derived period column is used internally. The value for each row is created and maintained automatically by Vantage based on the table definition. Data is never inserted explicitly for the derived period column, and that column is never itself returned or queried.

```
INSERT INTO Policy
(Policy_ID, Customer_ID, Policy_Type, Policy_Details,
 Policy_Begin, Policy_End)
VALUES (541008, 246824626, 'AU', 'STD-CH-345-NXY-00',
        DATE '2009-10-01', DATE '2010-10-01');
```

```
SELECT * FROM Policy;
```

```
Policy_ID  Customer_ID  Policy_Type  Policy_Details
Policy_Begin  Policy_End
```

```
-----
-----
```

```

541008      246824626  AU          STD-CH-345-NXY-00
2009/10/01  2010/10/01

```

Temporal Queries and Modifications

Vantage supports temporal SQL that allows you to qualify queries based on the durations associated with the rows:

- AS OF queries consider only those rows that are in effect as of a given point in time.
- BETWEEN *time1* AND *time2* queries consider only rows with durations that overlap or immediately succeed a given time period.
- FROM *time1* TO *time2* queries are similar to BETWEEN ... AND queries, but they do not include rows where the duration succeeds but does not overlap the given time period.
- CONTAINED IN(*time1*, *time2*) queries qualify rows with durations that lie within the specified time period.

For example:

```

SELECT Customer_ID, Policy_type
FROM Policy
FOR VALIDTIME AS OF DATE'2009-11-24';

```

```

Customer_ID  Policy_Type
-----
246824626   AU

```

Temporal DELETE and UPDATE modifications can include a FOR PORTION OF qualifier that specifies when during the duration of a row a change or deletion is in effect, so such changes need not apply to the whole row in an all-or-nothing fashion.

Note:

Teradata provides rich support for Period data types, including operators, functions, and predicates, which could be used to return similar results. However, Period data types, period functions (BEGIN, END, LAST and INTERVAL), and the period predicate operator MEETS, are not ANSI standard SQL.

Related Information

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

ANSI Temporal Concepts

This section defines important concepts and terms related to Teradata ANSI temporal table support.

Temporal tables store and maintain information with respect to time. Using temporal tables, Vantage can process statements and queries that include time-based reasoning. ANSI temporal tables include one or two special derived columns that represent different time dimensions. These dimensions are used for different purposes.

Derived Period Columns

A derived period column is derived from DateTime data in other table columns. Derived period columns can be specified in a table definition and are maintained dynamically by the database.

ANSI temporal tables can have one or two derived period columns to represent different kinds of time. These derived period columns combine the values from two regular DateTime type columns, and represent the period or duration bounded by the values in the two component columns. Special SQL DDL syntax for CREATE and ALTER TABLE statements allows specification of these derived period columns to create temporal tables.

Note:

The period that a derived period column represents starts at the beginning bound and extends up to, but does not include, the ending bound value.

Derived period columns function much like native Teradata Period data type columns. Period data types are not ANSI compatible, they are Teradata extensions to the ANSI SQL:2011 standard. Teradata provides rich support for Period data types, including functions, operators, and predicates, many of which can be used on derived temporal columns. Period functions BEGIN, END, LAST, and INTERVAL, and the period predicate operator MEETS are not ANSI standard SQL.

System Time and Valid Time

Static, time-related data can be added to tables by adding columns defined to have DateTime data types such as DATE or TIMESTAMP. Such tables are not considered to be temporal tables because the time values are not automatically managed by the database when changes are made to row data.

Temporal tables include one or two derived period columns that represent temporal dimensions. These two dimensions are independent, and used for different purposes.

System Time

System time is the time period during which the information in a row is or was known to the database. It reflects the database reality, automatically recording when rows have been added, modified, and deleted in tables that include system time.

The system-time period is represented by a derived period column named `SYSTEM_TIME`.

- The beginning bound of the system-time period is the time when the database became aware of a row. It records when every row was added to the table.
- The ending bound of a system-time period reflects when any of the information in an existing row was modified, or when the row was deleted from the table. Rows containing information that is currently in effect have system-time periods with indefinite ending bounds, represented practically as the maximum system timestamp value (9999-12-31 23:59:59.999999+00:00).

You cannot set or modify the values in the component `TIMESTAMP` columns that are the beginning and ending bound values of the derived system-time period column. Vantage maintains these values automatically if the system-time table is created to have system versioning. (This document assumes all system-time tables are created to have system versioning, which is required to enable temporal table behavior.)

Every change to a table that has a system-time dimension is tracked by the database. The physical rows of system-time tables are never deleted, and their substantive data is never modified in these tables:

- When a row is explicitly deleted from a system-time table, the row remains in the table, but the value of the system-time end bound is timestamped to reflect the time of the deletion.
- When a row is modified in a system-time table, the original row with the original values is logically deleted, but the physical row remains in the table. The value of end bound of the system-time period for the original row is timestamped to reflect the time of the modification. A copy of the row having the new, modified values is automatically inserted into the table, and its system-time starting bound is timestamped with the time of the modification.

System-time table rows with end bounds less than the maximum system timestamp value are referred to as “closed,” and can no longer be changed in the database. These rows remain in the table to provide a complete internal history of what happened to the rows in the table. Any prior state of a system-time table can be reproduced. Closed rows are unavailable to most DML modifications, and can only be physically deleted from the table by altering the table definition, whereupon all closed rows are physically deleted from the table permanently.

Use system-versioned system-time tables when historical changes need to be automatically tracked and maintained in the database. For example, system-time tables can be used for some types of regulatory compliance.

For a detailed discussion of system-time tables, see [Working With ANSI System-Time Tables](#).

System Versioning

In order for a system-time table to be considered a temporal table, and to include the automatic timestamping and tracking behaviors characteristic of system-time temporal tables, the table must be

designated to have system versioning in the table definition. It is the system versioning that bestows on the table the temporal capabilities. For purposes of discussion in this documentation, assume that references to "system-time" tables implicitly refer to "system-versioned system-time" tables unless otherwise noted.

Valid Time

Valid time is the time period during which the information in a row is in effect or true for purposes of real-world application. (ANSI calls this period "application time.") 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 represented by a derived period column designated by a VALIDTIME column attribute, though the column may have any name.

- The beginning bound of the valid-time period is the time when the information in the row commences to be true or in effect.
- The ending bound of a valid-time period reflects the time after which the information in the row is no longer considered to be true or valid, such as the end date of a contract.

Consequently, the valid time of a row can span times in the past, present, and future. Rows containing information that is currently valid have valid-time periods that include the current time. If the information in a row is current, but has a finite valid-time period, after sufficient time passes the information ages and becomes no longer valid. At that point the row is considered to be a history row.

When you add a new row to a table with a valid-time dimension, you must specify the time period during which the row information is valid by including values for the beginning and ending bounds of the valid-time period. Rows containing information that is valid indefinitely are represented practically with an ending bound value of the maximum system date or timestamp value.

When you make a time-bounded change to a row in a valid-time table, the database automatically creates new rows and defines their valid-time periods as necessary to delimit in time when the change was valid, but preserves the original state of the information for periods before and after the change. The nature of these automatic changes are determined by how the time period specified for the change relates to the valid-time period of the rows. For more information on this, see [Modifying Temporal Tables](#).

Use valid-time tables when the information in table rows is delimited by time, and for which row information should be maintained, tracked, and manipulated in a time-aware fashion.

Valid-time tables are 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.

For a detailed discussion of valid-time tables, see [Working With ANSI Valid-Time Tables](#).

Bitemporal Tables

System time and valid time are independent time dimensions that are used for different purposes. Bitemporal tables include both dimensions. Changes to bitemporal tables that happen automatically as a result of row modifications are independent for the system-time and valid-time dimensions. These

dimensions must be considered separately when determining what will happen to a row in a bitemporal table as a result of a row modification.

UNTIL_CLOSED and UNTIL_CHANGED

UNTIL_CLOSED and UNTIL_CHANGED are special terms that are understood by Vantage under some circumstances to represent the ending bound of temporal time periods when the duration is indefinite, forever, or for which the end is an unspecified and unknown time in the future. The terms represent “the maximum system timestamp or date value.”

- UNTIL_CLOSED is used with system-time periods, and represents `TIMESTAMP '9999-12-31 23:59:59.999999+00:00'`.
- UNTIL_CHANGED is used with valid-time periods. Its value depends on the data type and precision of the valid-time column. If the type is `DATE`, UNTIL_CHANGED is the value `DATE '9999-12-31'`. If the type is `TIMESTAMP`, UNTIL_CHANGED is the value of `TIMESTAMP '9999-12-31 23:59:59.999999+00:00'`, with precision and time zone matching that of the data type specified for the valid-time start and end columns.

Although these terms do not conform to current ANSI standards, the database supports the use of UNTIL_CHANGED for INSERTs and some queries of ANSI valid-time tables.

These four functions, also extensions to the ANSI standard, can be used in WHERE clauses when querying ANSI temporal tables in the database:

- IS NOT UNTIL_CLOSED
- IS UNTIL_CHANGED
- IS NOT UNTIL_CHANGED

For more information on these functions, see *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211, and the discussion of Period data types in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Temporal Row Types

Rows in temporal tables can be broadly classified according to their temporal time periods.

- In system-time tables, a row can be either “open” or “closed.”
 - Open rows have system-time ending bound values of UNTIL_CLOSED, the maximum possible system timestamp value, indicating they are still active in the database and able to participate in normal SQL operations.
 - Closed rows have system-time ending bound values that are anything less than (prior to) UNTIL_CLOSED. Closed rows cannot be changed. The system-time period end value for these rows indicates when they became inactive in the database. These are rows that have been logically deleted from the table, but remain physically there for record-keeping and history purposes.

Closed rows include rows that have been explicitly deleted from the table and rows with values that have been superseded by row modifications since the original row was added to the table.

Such modifications close the original row, and create a new row in the table to hold the changed information. For more information on how row modifications generate new table rows in temporal tables, see [Modifying Temporal Tables](#).

- In valid-time tables, a row can be either current, future, or history, depending on how the time period during which the row information is valid relates to the current time.
 - Current rows have valid-time periods that overlap the current time. The data in these rows is currently in effect.
 - Future rows have a valid time that commences after the current time. The data in these rows is not yet in effect, but will be when their valid-time period includes the current time unless the information in the row changes before it becomes current. An example would be information describing terms of a contract that does not begin until next month.
 - History rows have valid-time periods that have ended before the current time. The data in these rows was true at some point, but is no longer valid. The information may have become invalid because the original valid-time period has been exceeded, such as for an expired contract, or because of a change to the row data that made the original data no longer untrue, such as a change in terms of an existing valid contract.
 - Rows in valid-time tables can also be discussed as being either valid or invalid at any point in time, depending on whether the valid-time period includes the point in time.
- In bitemporal tables the two temporal dimensions are distinct from each other, so a row is simultaneously open or closed in the system-time dimension, and current, future, or history in the valid-time dimension.

In a sense, the system-time dimension takes precedence. If the valid-time period of a row in a bitemporal table is in the future, but the row is closed in the system-time dimension — indicating the row has been deleted from the table — the row is not considered to be a future row, because it is no longer active in the database. The row has been logically deleted from the database, and serves only to show the state of the row at the time the row was deleted or modified.

Inserting Rows into Temporal Tables

Because temporal tables track information in a time-aware fashion, any new information inserted into temporal tables must be associated with at least one time period: the system time or valid time. Rows in bitemporal tables include both kinds of time periods.

- When you insert a row into a system-time table, the database manages the system-time period for you. The start of the system-time period is automatically set to the time the row was inserted into the table, and the end of the system-time period is set to UNTIL-CLOSED. The row is considered open and actively participates in SQL operations until the row is deleted or until a modification to the row causes the original information to become obsolete.
- When you insert a row into a valid-time table, you must specify the start and end of the valid-time period for the row, the period for which the information in the row is in effect. This could be a period in the past, one that includes the current time, or can be a period in the future. The valid-time period can span from

history to future times. If you are inserting a row containing information that is valid indefinitely, set the end value of the valid-time period to UNTIL_CHANGED.

For more information about inserting rows into temporal tables, see [Modifying Temporal Tables](#).

Querying Temporal Tables

Every row in a temporal table is associated with at least one time dimension by having a corresponding time period. Special temporal SQL qualifiers for SELECT statements (AS OF, BETWEEN ... AND, FROM TO, and CONTAINED IN) allow you to query the data based on these time dimensions. For example:

- For system-time tables, you can query the data that is currently active, as with any table, but you can also query inactive rows to determine when in the past these rows were modified or deleted. Although these rows no longer participate in most SQL operations, they remain in the table, and can be queried using special temporal SQL.
- For valid-time tables, you use temporal SQL to see only rows that are currently valid, or rows that are, were, or will be valid during or within a period of time you specify. You can choose to limit queries to history rows that were valid at earlier times but are not valid now, or future rows that will not become valid until a time in the future.

For more information about querying temporal tables, see [Querying ANSI System-Time Tables](#) and [Querying ANSI Valid-Time Tables](#).

Modifying Temporal Tables

Modifications to temporal tables involve special time-aware handling by the database. Because they are used for different purposes, system-time columns are treated differently than valid-time columns when rows are modified.

System-Time Modifications

Temporal modifications to system-time columns are straightforward and entirely automatic. They are not influenced by the nature of the modification.

- When a row is deleted from a system-time table, the row is not physically deleted from the table. The end of the system-time period of the row is set to the time of the deletion, and the row becomes a closed row. It no longer participates in most SQL operations on the table, but remains in the table as a historical record of what existed in the past.
- When a row is modified in a system-time table, the end of the system-time period of the row is set to the time of the modification, closing the row, and a copy of the row with the modified data is inserted into the table. The system-time period of the new row begins at the time of the modification, and ends at UNTIL_CLOSED. The new row is active in the database and can participate in SQL operations.

For more information on modifying ANSI system-time tables, see [Modifying Rows in ANSI System-Time Tables](#).

Valid-Time Modifications

Assume a table represents contracts, with each row presenting the terms of an existing contract. If the terms of the contract are changed during the contract period, a regular update to a nontemporal table would result in the table reflecting the new contract terms. But potentially useful information would be lost: the fact that, prior to the modification, the contract terms were different. Valid-time tables let you model this reality using special temporal syntax for modifications, and by special handling of the rows changed by these modifications.

Modifications to valid-time tables are more flexible than those to system-time tables. Although the database again does the temporal bookkeeping for you automatically, the nature of the automatic modifications depends on the relationship between the valid time of the row and the effective time period you specify for the modification.

When you make UPDATE and DELETE modifications to valid-time tables, you can optionally specify a time period for which the modification applies. This is called the period of applicability (PA) of the modification. In this context, the valid-time period of a row is called the period of validity (PV) of the row. The PA of the modification syntax acts as an additional qualifier on the SQL. It is the relationship between the PA and PV that determines which rows can be modified, and whether the modification automatically creates new rows in the table that allow the database to reflect the time-delimited nature of the change.

The simplest case is a change that is effective as of one point in time during the PV of a row, and that lasts for the remaining duration of the PV. A simple change to terms of a contract is an example of this. This is how Vantage handles the change to preserve the temporal nature of change:

- A copy of the row is automatically created and modified to show the new terms. The PV of the new 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, because the new terms are valid for the remaining life of the contract.
- The original row, storing the original terms of the contract is marked as a history row. The ending bound of the valid-time of the old row is set to the time of the modification, because that is when the old terms in that row ceased to be valid.

Modifications to tables that have a valid-time dimension can apply to any period of time, even times that have passed or that are in the future. The changes affect only those rows with PVs that overlap the PA of the modification, and only for as long as the PA and PV coincide.

For example, if the PA of the modification lies within the PV of a row, such as a change to a contract that starts after the contract has begun, but ends before the contract expires, three rows will result from the 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 takes effect.
- The second row has the modified information, and a valid-time period that matches the PA of the modification statement.
- The third row retains the original row information, but has a valid-time period that begins when the modification is no longer valid, and extends through the end time of the original PV of the row.

In this way, valid-time tables keep an automatic history of all changes. Unlike closed rows in system-time tables, history rows in valid-time tables remain accessible to all SQL operations. 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.

Modifications that do not specify a PA behave just as modifications do on nontemporal tables, without affecting the valid time of the modified rows. They affect all rows in the table that meet the query criteria, regardless of the valid time of rows.

For more information about modifying rows in ANSI valid-time tables, see [Modifying Rows in ANSI Valid-Time Tables](#).

Bitemporal Table Modifications

The system-time and valid-time dimensions of bitemporal tables are independent of each other, and are affected just as they are in system-time and valid-time tables, with one important difference. In a bitemporal table only rows that are open in the system-time dimension (those that have a system-time period ending bound of UNTIL_CLOSED) participate in modifications. After a row is closed in system time, it is no longer active in the database.

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

For example, if a row in a bitemporal table is deleted, the ending bound of the system-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 system-time period, is that the row has been deleted.

The valid-time period of the closed 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 it existed at the time of the deletion.

The result of updates to rows in bitemporal tables are more complex, but are completely consistent with the idea of the system-time and valid-time dimensions acting independently.

For example, assume the contract terms 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 updated. Because this is a temporal table, Vantage automatically inserts a copy of the row to store the new terms. The PV 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 system-time period of the new row reflects when the new row was created, and the end of the system-time period is indefinite, set as UNTIL_CLOSED, which it will remain until the newly added row is deleted or modified.

The original row is automatically modified to have the end of the PV reflect the time of the change, when the old contract terms became obsolete. This row becomes a history row in the valid-time dimension. Note that both rows remain open rows in the system-time dimension, and as such, both 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 system-time dimension, however, a copy is made of the original row, reflecting the original PV, but this row is now closed in the system-time dimension as of the time the contract terms changed. No further changes can be made to this row, because it is closed in system time. It provides a permanent “before” snapshot of the original row as it existed in the database before it was changed.

Note that the temporal operations performed on the row automatically by Vantage include independent actions that result from the table having both a system-time dimension and a valid-time dimension.

For more information about modifying ANSI bitemporal tables, see [Modifying Rows in ANSI Bitemporal Tables](#).

Timestamping

An awareness of time is the defining feature of a temporal database. A large part of temporal table handling involves automatic timestamping by the database.

Whenever a row in a system-time table is modified or deleted, or when a row in a valid-time table is modified in a time-bounded fashion, the system automatically timestamps the row and any new rows that are created as a result of the change. These timestamps note the time of the change, and are used to close rows in a system-time tables and modify the PV as appropriate for new rows in valid-time tables.

Timestamping happens at the transaction level, and timestamps are entered automatically by Vantage as the value of the `TEMPORAL_TIMESTAMP` or `TEMPORAL_DATE` built-in database functions at the time of the change. These resolve to 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. `TEMPORAL_TIMESTAMP` and `TEMPORAL_DATE` remain the same throughout the transaction.

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

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. Removing system versioning from a system-time table physically deletes all closed rows from the table, and renders the table a non-temporal 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.
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.

Exceptions to the ANSI Standard

The Teradata implementation of ANSI/ISO Temporal tables and syntax is extended to include the following non-ANSI standard capabilities:

- The ANSI/ISO standard allows for a maximum of only two derived period columns in a temporal table, one for valid time and one for system time. ANSI temporal tables in Vantage can have more than two derived period columns.
- The ANSI/ISO standard does not define a CONTAINED IN qualifier for querying system-time tables. The CONTAINED IN qualifier can be used to query ANSI system-time tables in the database. See [Querying ANSI System-Time Tables](#).
- The ANSI/ISO standard does not define special temporal qualifiers for querying valid-time tables. ANSI temporal valid-time tables in the database can be queried using the same temporal qualifiers as can be used for querying ANSI system-versioned system-time tables (AS OF, BETWEEN ... AND, FROM TO, and CONTAINED IN).
- The ANSI/ISO standard for creating valid-time tables does not require or support the [AS] VALIDTIME column attribute. ANSI temporal valid-time tables in Vantage require [AS] VALIDTIME as part of the column specification in the table definition.

- The ANSI/ISO standard does not support the temporal qualifiers and constraints that are used in the Teradata proprietary implementation of temporal tables that was developed prior to the release of the ANSI/ISO standard for temporal tables. Temporal qualifiers and constraints used on Teradata proprietary temporal tables can be used with the Teradata implementation of ANSI temporal tables. For more information about these constraints and qualifiers, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

Working With ANSI System-Time Tables

System time is the time period during which the information in a row is or was known to the database. It reflects the database reality, automatically recording when rows have been added, modified, and deleted in tables that include system time.

Use system-versioned system-time tables when historical changes need to be automatically tracked and maintained in the database. For example, system-time tables can be used for some types of regulatory compliance.

In order for a system-time table to be considered a temporal table, and to include the automatic timestamping and tracking behaviors characteristic of system-time temporal tables, the table must be designated to have system versioning in the table definition. It is the system versioning that bestows on the table the temporal capabilities. For purposes of discussion in this documentation, assume that references to "system-time" tables implicitly refer to "system-versioned system-time" tables unless otherwise noted.

Note:

This material covers only the syntax, rules, and other details that apply to ANSI temporal tables. The syntax diagrams presented are extracts of the full diagrams, and focus on the temporal syntax.

Most of the existing rules and options that apply to conventional, nontemporal versions of the SQL statements discussed here also apply to the temporal statements. The nontemporal rules and options are not repeated here. For more information on conventional, nontemporal SQL DDL and DML statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 .

Creating ANSI System-Time Tables

ANSI supports special CREATE TABLE and ALTER TABLE syntax for creating system-time tables. Creating temporal tables includes:

- Defining the beginning and ending bound columns for the system-time period
- Defining the SYSTEM_TIME derived period column
- Enabling system versioning for the table

CREATE TABLE (ANSI System-Time Table Form)

Create a new ANSI system-time table.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

CREATE TABLE Syntax (ANSI System-Time Table Form)

```
CREATE MULTISET TABLE table_name (
  sys_start TIMESTAMP(6) WITH TIME ZONE NOT NULL GENERATED ALWAYS AS ROW START,
  sys_end TIMESTAMP(6) WITH TIME ZONE NOT NULL GENERATED ALWAYS AS ROW END,
  PERIOD FOR SYSTEM_TIME ( sys_start, sys_end )
) WITH SYSTEM VERSIONING [;]
```

CREATE TABLE Syntax Elements (ANSI System-Time Table Form)

table_name

The name of the system-time table. May include database qualifier.

sys_start

The name of the column that will store the beginning bound of the system-time period.

GENERATED ALWAYS AS ROW START

Required attribute for column that defines the beginning bound of system-time period.

sys_end

The name of the column that will store the ending bound of the system-time period.

GENERATED ALWAYS AS ROW END

Required attribute for column that defines the ending bound of system-time period.

PERIOD FOR SYSTEM_TIME

Creates the system-time derived period column.

SYSTEM VERSIONING

Required attribute for system-time temporal tables. Must be the last clause in the CREATE TABLE statement.

Usage Notes

General

- The `sys_start` column must be defined as NOT NULL and GENERATED ALWAYS AS ROW START. The `sys_end` column must be defined as NOT NULL and GENERATED ALWAYS AS ROW END.
- The GENERATED ALWAYS AS ROW START or END attributes cannot be dropped from the definitions of the columns that constitute the `SYSTEM_TIME` derived period column.
- Component columns of a system-time derived period column cannot be part of the primary index.
- To function as a temporal table, the system-time table must be defined WITH SYSTEM VERSIONING. (Valid-time tables do not have system versioning.)
- A table can have only one system-time period definition.
- A system-time table cannot be a queue, error, global temporary, global temporary trace, or volatile table.
- CHECK constraints on tables with system time cannot include the start or end columns of the system-time period.
- The start and end columns of the system-time period cannot be part of a primary or foreign key of a temporal referential constraint.
- System-time tables cannot act as source tables in CREATE TABLE AS statements.
- Statistics cannot be collected on the system-time derived period column, but they can be collected on the component start and end time columns.
- Algorithmic compression (ALC) is not allowed on DateTime columns that act as the beginning and ending bound values of a temporal derived period column.

Row Partitioning ANSI System-Time Tables

Temporal tables should be row partitioned to improve query performance. Partitioning can logically group the table rows into open and closed rows. Queries of open rows are directed automatically to the partition containing the 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.

Example: Row Partitioning an ANSI System-Time Table

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

```
CREATE MULTISET TABLE employee_systime (
    eid INTEGER NOT NULL,
```

```

    ename VARCHAR(10) NOT NULL,
    deptno INTEGER NOT NULL,
    sys_start TIMESTAMP WITH TIME ZONE NOT NULL
                        GENERATED ALWAYS AS ROW START,
    sys_end TIMESTAMP WITH TIME ZONE NOT NULL
                        GENERATED ALWAYS AS ROW END,
    PERIOD FOR SYSTEM_TIME(sys_start, sys_end)
) PRIMARY INDEX(eid) WITH SYSTEM VERSIONING
PARTITION BY
    CASE_N (END(SYSTEM_TIME) >= CURRENT_TIMESTAMP, NO CASE);

```

Note:

The partitioning expression could have used `sys_end` instead of `END(SYSTEM_TIME)`.

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.

Example: Creating a System-Time Table

The following example creates a system-time table and includes the system versioning clause, which is required to make the table a temporal table.

```

CREATE MULTISET TABLE employee_system_time (
    eid INTEGER NOT NULL,
    name VARCHAR(10) NOT NULL,
    deptno INTEGER NOT NULL,
    sys_start TIMESTAMP(6) WITH TIME ZONE NOT NULL
                        GENERATED ALWAYS AS ROW START,
    sys_end TIMESTAMP(6) WITH TIME ZONE NOT NULL
                        GENERATED ALWAYS AS ROW END,
    PERIOD FOR SYSTEM_TIME(sys_start, sys_end)
)
PRIMARY INDEX (eid) WITH SYSTEM VERSIONING;

```


ALTER TABLE (ANSI System-Time Table Form)

Temporal syntax for ALTER TABLE allows you to create ANSI temporal system-time tables from existing tables.

Adding system-time to the table involves these ALTER TABLE operations:

- Adding a SYSTEM_TIME derived period column to the table by specifying the columns that will serve as the beginning and ending bounds of the system-time period.
- Adding or altering the columns that will serve as the beginning bound and ending bound of the system-time period. If these columns already exist in the table, special attributes must be added to them.
- Adding system versioning to the table.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

ALTER TABLE Syntax (ANSI System-Time Table Form)

Use the following ALTER TABLE syntax to create a system-time table by adding a system-time derived period column and the component start and end columns in a single ALTER TABLE statement:

```
ALTER TABLE 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 [;]

ALTER TABLE table_name ADD SYSTEM VERSIONING [;]
```

A system-time table is not considered to be a temporal table, and is not afforded any special temporal behaviors until system versioning has been added to the table. Add system versioning to the system-time table using the following syntax in a separate ALTER TABLE statement:

```
ALTER TABLE table_name ADD SYSTEM VERSIONING [;]
```

Note:

The ALTER TABLE statement components must be in the order shown, with the derived period column defined before the component start and end columns. There is no comma between the ADD clauses in this case.

ALTER TABLE Syntax Elements (ANSI System-Time Table Form)

table_name

The name of the system-time table. May include database qualifier.

PERIOD FOR SYSTEM_TIME

Creates the system-time derived period column.

sys_start

The name of the column that will store the beginning bound of the system-time period.

GENERATED ALWAYS AS ROW START

Required attribute for column that defines the beginning bound of system-time period.

sys_end

The name of the column that will store the ending bound of the system-time period.

GENERATED ALWAYS AS ROW END

Required attribute for column that defines the ending bound of system-time period.

SYSTEM VERSIONING

Required attribute for system-time temporal tables.

Usage Notes

General

After a table has been converted to a system-versioned system-time table, most ALTER TABLE operations are not allowed. These tables are typically used for regulatory and compliance purposes, and modifications to the table structure could defeat those purposes. To restore a system-versioned system-time table to a nontemporal table involves dropping the system versioning, after which all normal ALTER TABLE operations are allowed.

Dropping System Versioning and System-Time

System-versioned system-time tables are typically used for regulatory and compliance purposes, and for keeping a table-resident history of database operations on the table data. Consequently, most types of ALTER TABLE changes to these tables are not allowed. However, ALTER TABLE can be used to remove

system versioning. After system versioning has been removed from a temporal table, the table becomes a regular nontemporal table, and all normal ALTER TABLE operations are permitted.

Use the following ALTER TABLE syntax to remove system versioning from a system-time table:

```
ALTER TABLE your_system_time_table DROP SYSTEM VERSIONING;
```

Where *your_system_time_table* is the name of a system-versioned system-time table.

Note:

Dropping the SYSTEM VERSIONING from a system-time table deletes all closed rows from the table, and makes the table a nontemporal table.

To drop the system-time columns, including the derived period column and its component beginning and ending bound TIMESTAMP columns, use the following ALTER TABLE syntax:

```
ALTER TABLE your_system_time_table DROP PERIOD FOR SYSTEM_TIME;
```

Where, again, *your_system_time_table* is the name of a system-versioned system-time table.

Dropping the system-time derived period column will automatically drop the two component columns.

Note:

You must drop the SYSTEM VERSIONING from a system-time table before you can drop the SYSTEM_TIME derived period column and component columns.

Example: ALTER TABLE to Convert a Nontemporal Table to an ANSI System-Time Table

The following SQL creates a regular nontemporal table, and inserts some rows:

```
CREATE MULTISET TABLE employee_systime (
  eid INTEGER NOT NULL,
  ename VARCHAR(10) NOT NULL,
  deptno INTEGER NOT NULL,
  sys_start TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  sys_end TIMESTAMP(6) WITH TIME ZONE NOT NULL
) PRIMARY INDEX(eid);

INSERT INTO employee_systime VALUES
  (1001, 'Sania', 111, TIMESTAMP'2002-01-01 00:00:00.000000-08:00',
    TIMESTAMP'9999-12-31 23:59:59.999999+00:00');
INSERT INTO employee_systime VALUES
```

```

(1002,'Ash',333,TIMESTAMP'2003-07-01 12:11:00.000000-08:00',
      TIMESTAMP'9999-12-31 23:59:59.999999+00:00');
INSERT INTO employee_systime VALUES
(1003,'SRK',111,TIMESTAMP'2004-02-10 00:00:00.000000-08:00',
      TIMESTAMP'2006-03-01 00:00:00.000000-08:00');
INSERT INTO employee_systime VALUES
(1004,'Fred',222, TIMESTAMP'2002-07-01 12:00:00.350000-08:00',
      TIMESTAMP'2005-05-01 12:00:00.350000-08:00');
INSERT INTO employee_systime VALUES
(1005,'Alice',222,TIMESTAMP'2004-12-01 00:12:23.120000-08:00',
      TIMESTAMP'2005-05-01 12:00:00.450000-08:00');
INSERT INTO employee_systime VALUES
(1004,'Fred',555, TIMESTAMP'2005-05-01 12:00:00.350000-08:00',
      TIMESTAMP'9999-12-31 23:59:59.999999+00:00');
INSERT INTO employee_systime VALUES
(1005,'Alice',555,TIMESTAMP'2005-05-01 12:00:00.450000-08:00',
      TIMESTAMP'9999-12-31 23:59:59.999999+00:00');

```

An unqualified SELECT on the table returns all rows, regardless of whether the row is open or closed in system time, because this is not yet a temporal table:

```
SELECT * FROM employee_systime;
```

eid	ename	deptno	sys_start	sys_end
1002	Ash	333	2003-07-01 12:11:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1005	Alice	222	2004-12-01 00:12:23.120000-08:00	2005-05-01 12:00:00.450000-08:00
1004	Fred	222	2002-07-01 12:00:00.350000-08:00	2005-05-01 12:00:00.350000-08:00
1005	Alice	555	2005-05-01 12:00:00.450000-08:00	9999-12-31 23:59:59.999999+00:00
1004	Fred	555	2005-05-01 12:00:00.350000-08:00	9999-12-31 23:59:59.999999+00:00
1001	Sania	111	2002-01-01 00:00:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1003	SRK	111	2004-02-10 00:00:00.000000-08:00	2006-03-01 00:00:00.000000-08:00

Two ALTER TABLE statements can change the table into a system-time temporal table:

```

ALTER TABLE employee_systime
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;
ALTER TABLE employee_systime
ADD SYSTEM VERSIONING;

```

Now an unqualified SELECT will show only the rows that are open in system time:

```
SELECT * FROM employee_systime;
```

eid	ename	deptno	sys_start		sys_end	
1002	Ash	333	2003-07-01	12:11:00.000000+00:00	9999-12-31	23:59:59.999999+00:00
1001	Sania	111	2002-01-01	00:00:00.000000+00:00	9999-12-31	23:59:59.999999+00:00
1001	Fred	222	2002-07-01	12:00:00.350000+00:00	9999-12-31	23:59:59.999999+00:00
1003	Alice	222	2004-12-01	00:12:23.120000+00:00	9999-12-31	23:59:59.999999+00:00

Special temporal qualifiers allow you to display closed rows from system-time tables. For more information see [Querying ANSI System-Time Tables](#).

INSERT (ANSI System-Time Table Form)

Add new rows to ANSI system-time tables.

Syntax

There is no special temporal syntax for inserting rows into temporal tables. Use the standard SQL INSERT statement. However, note the following:

- You must include values for the beginning and ending bound columns that constitute the system-time derived period column.
- Values entered for the system-time columns will be automatically replaced by the database, so can be any values of any type. The value for the beginning system-time column will be replaced by the value of the `TEMPORAL_TIMESTAMP` function at the time of the insertion, and the ending system-time value will be replaced automatically with the maximum system timestamp value, `9999-12-31 12:59:59.999999+00:00`.
- As for any type of derived period column in the database, you cannot insert Period type values for the derived period column itself.

Example: Inserting Rows into an ANSI System-Time Table

```
INSERT INTO employee_system_time VALUES
(1001, 'Sania', 111, TIMESTAMP'2002-01-01 00:00:00.000000+00:00',
TIMESTAMP'2002-07-01 12:00:00.350000+00:00');
INSERT INTO employee_systime VALUES
(1001, 'Fred', 222, TIMESTAMP'2002-07-01
12:00:00.350000+00:00', UNTIL_CLOSED);
INSERT INTO employee_systime VALUES (1002, 'Ash', 333, 123, 456);
INSERT INTO employee_systime VALUES (1003, 'SRK', 111, , 'nothing');
INSERT INTO employee_systime VALUES (1003, 'Alice', 222, 'Wonder', NULL);
```

Note:

Values for the start and end columns that constitute the system-time period must be provided in the INSERT statement, but they will be automatically replaced by the database. The start time value will be replaced by the value of the `TEMPORAL_TIMESTAMP` function at the time of the insertion. The end time value will be replaced by the maximum system `TIMESTAMP(6) WITH TIME ZONE` value, `9999-12-31 23:59:59.999999+00:00`.

Bulk Loading Data into Temporal Tables

The database supports three 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.

If the FastLoad script includes a CHECKPOINT specification, restarts during loading can change the system-time values for rows that are inserted after the restart.

In this case, create a nontemporal table, load the data then use ALTER TABLE to add the SYSTEM_TIME derived period column and system-versioning.

- 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 and ALTER TABLE to convert the tables to system-versioned system-time 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.

Querying ANSI System-Time Tables

Special temporal syntax in the FROM clause of SELECT statements allows you to qualify queries by the time period to which they apply. Rows are only returned if they meet the temporal qualifications. These temporal qualifiers take precedence over other criteria that may be specified in a WHERE clause, which can be used to further restrict the rows returned by Vantage.

FROM Clause (ANSI System-Time Table Form)

Uses system-time dimension criteria to determine which rows in ANSI system-time tables are subject to a SELECT query.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

FROM Clause Syntax (ANSI System-Time Table Form)

```
FROM system_time_table [
  FOR SYSTEM TIME {
    AS OF point_in_time |
    BETWEEN point_in_time_1 AND point_in_time_2 |
    FROM point_in_time_1 TO point_in_time_2 |
    CONTAINED IN ( point_in_time_1, point_in_time_2 )
  }
]
```

FROM Clause Syntax Elements (ANSI System-Time Table Form)

system_time_table

The system-time table being queried. The table must be a system-versioned table to be subject to temporal syntax.

AS OF

Qualifies rows in system-time tables that were open at a given point in time. Use the AS OF qualifier to query the table as it existed at the AS OF time.

Note:

Although the returned rows were open (active in the database) at the time specified by the query, they may have been closed before the query is submitted. Such rows will show a timestamp for the ending bound that is prior to 9999-12-31 23:59:59.999999+00:00. Closed rows in the results indicate the row was modified or deleted after the AS OF time.

point_in_time_1

point_in_time_2

A timestamp expression that can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a DATE or TIMESTAMP[(n)] [WITH TIME ZONE] value.

The expression can be any expression, including parameterized values and built-in functions such as CURRENT DATE, CURRENT_TIMESTAMP, TEMPORAL_DATE, or TEMPORAL_TIMESTAMP. The expression cannot reference any columns, but it can be a self-contained noncorrelated scalar subquery.

BETWEEN ... AND

Qualifies all rows with system-time periods that overlap or immediately succeed the period defined by *point_in_time_1* and *point_in_time_2*.

Note:

This is not the commonly understood meaning of "between" because BETWEEN will qualify rows with system-time periods that begin before by *point_in_time_1*, and rows that start immediately after *point_in_time_2* and extend beyond that. For a qualifier that reflects the commonly understood meaning of BETWEEN, use the CONTAINED IN qualifier.

FROM TO

Qualifies all rows with system-time periods that overlap the period defined by *point_in_time_1* and *point_in_time_2*.

CONTAINED IN

Qualifies all rows with system-time periods that are between *point_in_time_1* and *point_in_time_2*. The system-time period starts at or after *point_in_time_1*, and ends before or at *point_in_time_2*.

Note:

CONTAINED IN is a Teradata extension to ANSI.

Usage Notes

- If no temporal qualifiers are used in the FROM clause, the default for system-time tables is to qualify all rows that are currently (at the time of the query) open.
- BETWEEN ... AND, FROM TO, and CONTAINED IN all qualify queries of temporal tables according to a period of time. Any table rows that were active in the database, open in system time, at any time during the specified period (or immediately after it in the case of BETWEEN ... AND) are qualified for the query. The differences in the three temporal qualifiers are subtle, but allow you to define your query to qualify a precise set of rows.

FROM Clause Examples

For the following examples, assume the queries are issued against the following system-versioned system-time table named `employee_systime` that contains a mix of open and closed rows:

eid	ename	deptno	sys_start				sys_end			
1002	Ash	333	2003-07-01	12:11:00.000000	-08:00	9999-12-31	23:59:59.999999	+00:00		
1005	Alice	222	2004-12-01	00:12:23.120000	-08:00	2005-05-01	12:00:00.450000	-08:00		
1004	Fred	222	2002-07-01	12:00:00.350000	-08:00	2005-05-01	12:00:00.350000	-08:00		
1001	Sania	111	2002-01-01	00:00:00.000000	-08:00	9999-12-31	23:59:59.999999	+00:00		
1003	SRK	111	2004-02-10	00:00:00.000000	-08:00	2006-03-01	00:00:00.000000	-08:00		
1004	Fred	555	2005-05-01	12:00:00.350000	-08:00	9999-12-31	23:59:59.999999	+00:00		
1005	Alice	555	2005-05-01	12:00:00.450000	-08:00	9999-12-31	23:59:59.999999	+00:00		

Rows with `sys_end` dates that are not 9999-12-31 23:59:59.999999+00:00 are closed in system time. They have either been logically deleted from the table or modified since they were first added to the table, but they remain in the table as a permanent record of prior states of the table. They can not participate in most SQL operations, such as UPDATES and DELETES, but using temporal qualifiers they can be retrieved from system-time tables.

In this case, the table shows that the department values (column deptno) for Fred and Alice were changed on 2005-05-01, which is the end date of the closed Fred and Alice rows and the start date of the new rows for them that contain the new information.

SRK also shows a sys_end date prior to 9999-12-31 23:59:59.999999+00:00, but because there is no open row remaining in the table for SRK, the sys_end date indicates when the row was (logically) deleted from the table.

A simple SELECT with no temporal qualifiers shows only the open, active rows in a system-time table. Open rows are indicated by system-time periods with ending bounds of 9999-12-31 23:59:59.999999+00:00.

```
SELECT * FROM employee_systime;
```

eid	ename	deptno	sys_start	sys_end
1002	Ash	333	2003-07-01 12:11:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1001	Sania	111	2002-01-01 00:00:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1004	Fred	555	2005-05-01 12:00:00.350000-08:00	9999-12-31 23:59:59.999999+00:00
1005	Alice	555	2005-05-01 12:00:00.450000-08:00	9999-12-31 23:59:59.999999+00:00

This reflects the state of the information today, at the moment the query is processed. Temporal qualifiers allow you to see a snapshot of the table as it existed at any prior time, and return results that were true for former states of the table.

Example: AS OF Query on an ANSI System-Time Table

The following AS OF query retrieves all table rows that were open and active at the point in time specified in the query, regardless of whether these rows are open or closed now.

```
SELECT *
FROM employee_systime
FOR SYSTEM_TIME AS OF TIMESTAMP '2005-01-01 00:00:01.000000-08:00';
```

eid	ename	deptno	sys_start	sys_end
1002	Ash	333	2003-07-01 12:11:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1001	Sania	111	2002-01-01 00:00:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1003	SRK	111	2004-02-10 00:00:00.000000-08:00	2006-03-01 00:00:00.000000-08:00
1004	Fred	222	2002-07-01 12:00:00.350000-08:00	2005-05-01 12:00:00.350000-08:00
1005	Alice	222	2004-12-01 00:12:23.120000-08:00	2005-05-01 12:00:00.450000-08:00

Apart from the sys_end values, the data in the rows reflect the state of the table as it existed on 2005-01-01. At that time Fred and Alice belonged to deptno 222, and SRK was still employed (the SRK row had not yet been deleted).

Because the rows for Fred, Alice, and SRK have sys_end values less than 9999-12-31 23:59:59.999999+00:00, you know that this information from 2005-01-01 is no longer current. One or more changes to these rows occurred after the AS OF date, and the time of the first of those changes is

reflected by the sys_end value. Rows with sys_end dates of 9999-12-31 23:59:59.999999+00:00 have not been changed or deleted since the AS OF date.

A query of the table AS OF 2005-05-02, would reflect a table query that occurred immediately after Fred and Alice had changed departments:

```
SELECT *
FROM employee_systime
FOR SYSTEM_TIME AS OF TIMESTAMP'2005-01-01 00:00:01.000000-08:00';
```

eid	ename	deptno	sys_start	sys_end
1002	Ash	333	2003-07-01 12:11:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1001	Sania	111	2002-01-01 00:00:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1003	SRK	111	2004-02-10 00:00:00.000000-08:00	2006-03-01 00:00:00.000000-08:00
1004	Fred	555	2005-05-01 12:00:00.350000-08:00	9999-12-31 23:59:59.999999+00:00
1005	Alice	555	2005-05-01 12:00:00.450000-08:00	9999-12-31 23:59:59.999999+00:00

Because AS OF queries reflect the state of the table at a given moment in time, they cannot indicate the nature of the changes that occurred to the closed rows that are returned. To determine the nature of changes to rows in a system-time table, you need to see the row as it existed both before and after the change. The remaining temporal query qualifiers, BETWEEN ... AND, FROM TO, and CONTAINED IN, allow you to specify spans of valid time, rather than instances. If, during the specified span, a row has been changed, both the old and new versions of the row are returned, which allows you to determine the nature of the change that caused the row to be closed in system time.

Example: BETWEEN ... AND Query on ANSI System-Time Table

To see the nature of the changes to the Fred and Alice rows, you need a query that will return the rows as they existed both before and after the change. You could submit a query with a BETWEEN ... AND temporal qualifier that spans the time of the change. We know from the AS OF query that the change occurred at 2005-05-01.

```
SELECT *
FROM employee_systime
FOR SYSTEM_TIME BETWEEN TIMESTAMP'2005-04-30 00:00:00.000001-08:00' AND
TIMESTAMP'2005-05-02 00:00:00.000001-08:00'
WHERE ename='Fred' OR ename='Alice'
ORDER BY ename;
```

eid	ename	deptno	sys_start	sys_end
1005	Alice	222	2004-12-01 00:12:23.120000-08:00	2005-05-01 12:00:00.450000-08:00
1005	Alice	555	2005-05-01 12:00:00.450000-08:00	9999-12-31 23:59:59.999999+00:00
1004	Fred	222	2002-07-01 12:00:00.350000-08:00	2005-05-01 12:00:00.350000-08:00
1004	Fred	555	2005-05-01 12:00:00.350000-08:00	9999-12-31 23:59:59.999999+00:00

From the results it is clear that from December 1st, 2004 through May 1st, 2005 Alice was in Department 222. From May 1st, 2005 until now, Alice has been in Department 555. Similarly, Fred started in Department 222 and was there from July 1st, 2002 until May 1st, 2005, after which his department also changed to 555, and remains 555 today. Perhaps all the personnel in Department 222 switched departments in 2005, or possibly the department number was changed.

Example: FROM TO Query on ANSI System-Time Table

Temporal qualifiers that span a period of time can also be used to select all rows in a system-time table, including all open and closed rows, if the specified period is sufficiently broad.

```
SELECT *
FROM employee_systime
FOR SYSTEM_TIME FROM TIMESTAMP'1900-01-01 00:00:00.000001-08:00' TO
CURRENT_TIMESTAMP;
```

eid	ename	deptno	sys_start	sys_end
1002	Ash	333	2003-07-01 12:11:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1005	Alice	222	2004-12-01 00:12:23.120000-08:00	2005-05-01 12:00:00.450000-08:00
1004	Fred	222	2002-07-01 12:00:00.350000-08:00	2005-05-01 12:00:00.350000-08:00
1005	Alice	555	2005-05-01 12:00:00.450000-08:00	9999-12-31 23:59:59.999999+00:00
1004	Fred	555	2005-05-01 12:00:00.350000-08:00	9999-12-31 23:59:59.999999+00:00
1001	Sania	111	2002-01-01 00:00:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1003	SRK	111	2004-02-10 00:00:00.000000-08:00	2006-03-01 00:00:00.000000-08:00

This has returned all rows in the system-time table.

Modifying Rows in ANSI System-Time Tables

When modifications are made to rows in system-versioned system-time tables, Vantage provides automatic handling of the system-time period as needed to maintain the information in a time-aware fashion.

DELETE and UPDATE modifications to system-time tables leave permanent records of the changes in the database. Deleted rows are “closed” in system time by having the end of their system-time period set to the time of the deletion. Modifications to rows close the old row in system time, and automatically create new rows to reflect the new information with system-time periods that begin at the time of the modification.

Closed rows persist in system-time tables until they are explicitly removed and physically deleted from the table. This can only happen if the system versioning is explicitly dropped from the table, whereupon all closed rows are automatically deleted.

The mechanism of new rows being automatically generated by Vantage for modifications to temporal tables is described in more detail in [Modifying Temporal Tables](#).

DELETE (ANSI System-Time Form)

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Deletes one or more rows from system-time tables.

Syntax

There is no special additional temporal syntax for the DELETE statement when used on a system-time table. The syntax is identical to that used for nontemporal tables. That syntax is described fully in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Usage Notes

- Although the syntax for a DELETE statement is identical for system-time and non-temporal tables, the effects are different, because rows are not physically deleted from system-time tables as a result of a DELETE.
- Because rows are not normally physically deleted from system-time tables, the storage required for these tables will not decrease, even when rows are deleted. To physically delete closed rows from system-time tables, drop the system versioning from the table. This automatically physically deletes all closed rows from the table.

For purposes of archiving closed rows prior to deletion, closed rows may be copied to new tables by means of INSERT ... SELECT operations, after which the new tables can be archived.

Example: Deleting Rows from an ANSI System-Time Table

The following example assumes the DELETE operation is applied to the following system-versioned system-time table named `employee_systime`:

eid	ename	deptno	sys_start		sys_end	
1002	Ash	333	2003-07-01	12:11:00.000000-08:00	9999-12-31	23:59:59.999999+00:00
1005	Alice	222	2004-12-01	00:12:23.120000-08:00	2005-05-01	12:00:00.450000-08:00
1004	Fred	222	2002-07-01	12:00:00.350000-08:00	2005-05-01	12:00:00.350000-08:00
1001	Sania	111	2002-01-01	00:00:00.000000-08:00	9999-12-31	23:59:59.999999+00:00
1003	SRK	111	2004-02-10	00:00:00.000000-08:00	2006-03-01	00:00:00.000000-08:00
1004	Fred	555	2005-05-01	12:00:00.350000-08:00	9999-12-31	23:59:59.999999+00:00
1005	Alice	555	2005-05-01	12:00:00.450000-08:00	9999-12-31	23:59:59.999999+00:00

The closed rows have already been deleted from the table, so are not subject to further deletion. Only the open rows in a system-time table can be deleted. The open rows are returned by a SELECT statement that is not temporally qualified:

```
SELECT * FROM employee_systime;
```

eid	ename	deptno	sys_start		sys_end	
1002	Ash	333	2003-07-01	12:11:00.000000-08:00	9999-12-31	23:59:59.999999+00:00
1005	Alice	555	2005-05-01	12:00:00.450000-08:00	9999-12-31	23:59:59.999999+00:00
1004	Fred	555	2005-05-01	12:00:00.350000-08:00	9999-12-31	23:59:59.999999+00:00
1001	Sania	111	2002-01-01	00:00:00.000000-08:00	9999-12-31	23:59:59.999999+00:00

```
DELETE FROM employee_systime WHERE ename='Sania';
```

A simple SELECT shows that the row has been deleted from the table.

```
SELECT * FROM employee_systime;
```

eid	ename	deptno	sys_start		sys_end	
1002	Ash	333	2003-07-01	12:11:00.000000-08:00	9999-12-31	23:59:59.999999+00:00
1005	Alice	555	2005-05-01	12:00:00.450000-08:00	9999-12-31	23:59:59.999999+00:00
1004	Fred	555	2005-05-01	12:00:00.350000-08:00	9999-12-31	23:59:59.999999+00:00

A temporal query that shows all open and closed rows in the table reveals that the Sania row has only been logically deleted, but remains inactive in the table as a closed row.

```
SELECT *
FROM employee_systime
FOR SYSTEM_TIME FROM TIMESTAMP '1900-01-01 00:00:00.000001-08:00' TO
CURRENT_TIMESTAMP;
```

eid	ename	deptno	sys_start		sys_end	
1002	Ash	333	2003-07-01	12:11:00.000000-08:00	9999-12-31	23:59:59.999999+00:00
1005	Alice	222	2004-12-01	00:12:23.120000-08:00	2005-05-01	12:00:00.450000-08:00
1004	Fred	222	2002-07-01	12:00:00.350000-08:00	2005-05-01	12:00:00.350000-08:00
1001	Sania	111	2002-01-01	00:00:00.000000-08:00	2014-02-22	22:32:01.540000-08:00
1003	SRK	111	2004-02-10	00:00:00.000000-08:00	2006-03-01	00:00:00.000000-08:00
1004	Fred	555	2005-05-01	12:00:00.350000-08:00	9999-12-31	23:59:59.999999+00:00
1005	Alice	555	2005-05-01	12:00:00.450000-08:00	9999-12-31	23:59:59.999999+00:00

In this way, a system-time table maintains a complete history of row deletions in the table.

UPDATE (ANSI System-Time Table Form)

Modifies column values in existing rows of system-time tables.

Syntax

There is no special additional temporal syntax for the UPDATE statement when used on a system-time table. The syntax is identical to that used for nontemporal tables. That syntax is described fully in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Usage Notes

Although the syntax for a UPDATE statement is identical for system-time and non-temporal tables, the effects are different. When you change a row in a nontemporal table, the original information in the row no longer exists. The situation can be thought of as if the original row had been deleted and replaced with a new row.

System-time tables make this implicit situation explicit, by literally deleting the original row from the table and inserting a new row into the table to reflect the modified information. Because rows are never physically deleted from system-time tables, the original row is closed in system time, and remains inactive in the table as a snapshot of how the table existed before the change.

Updates do not affect closed rows in system-time tables.

The SET clause cannot include the beginning or ending column components of the SYSTEM_TIME derived period column.

Examples

For the following examples, assume the queries are issued against the following system-versioned system-time table named `employee_systime` that contains a mix of open and closed rows:

eid	ename	deptno	sys_start		sys_end	
1002	Ash	333	2003-07-01	12:11:00.000000-08:00	9999-12-31	23:59:59.999999+00:00
1005	Alice	222	2004-12-01	00:12:23.120000-08:00	2005-05-01	12:00:00.450000-08:00
1004	Fred	222	2002-07-01	12:00:00.350000-08:00	2005-05-01	12:00:00.350000-08:00
1001	Sania	111	2002-01-01	00:00:00.000000-08:00	9999-12-31	23:59:59.999999+00:00
1003	SRK	111	2004-02-10	00:00:00.000000-08:00	2006-03-01	00:00:00.000000-08:00
1004	Fred	555	2005-05-01	12:00:00.350000-08:00	9999-12-31	23:59:59.999999+00:00
1005	Alice	555	2005-05-01	12:00:00.450000-08:00	9999-12-31	23:59:59.999999+00:00

In a table that has system time, only the open rows are subject to UPDATES. A simple SELECT with no temporal qualifiers returns the open table rows:

```
SELECT * FROM employee_systime;
```

eid	ename	deptno	sys_start		sys_end	
1002	Ash	333	2003-07-01	12:11:00.000000-08:00	9999-12-31	23:59:59.999999+00:00
1001	Sania	111	2002-01-01	00:00:00.000000-08:00	9999-12-31	23:59:59.999999+00:00
1004	Fred	555	2005-05-01	12:00:00.350000-08:00	9999-12-31	23:59:59.999999+00:00
1005	Alice	555	2005-05-01	12:00:00.450000-08:00	9999-12-31	23:59:59.999999+00:00

```
UPDATE employee_systime SET deptno=888 WHERE ename='Ash';
```

A simple SELECT shows the row has been changed as expected. Note that the start time for the system-time period records the time of the UPDATE, because it is from this time that the modification is known to the database and effective:

eid	ename	deptno	sys_start		sys_end	
1002	Ash	888	2014-02-23 22:27:56.540000-08:00	9999-12-31 23:59:59.999999+00:00		
1001	Sania	111	2002-01-01 00:00:00.000000-08:00	9999-12-31 23:59:59.999999+00:00		
1004	Fred	555	2005-05-01 12:00:00.350000-08:00	9999-12-31 23:59:59.999999+00:00		
1005	Alice	555	2005-05-01 12:00:00.450000-08:00	9999-12-31 23:59:59.999999+00:00		

A temporal select that shows open and closed rows demonstrates that the original row still exists in the table. It has been closed, and the end boundary of the old row system time has been set to the time of the modification, which is the time when the values in that row ceased to apply or be active in the database.

```
SELECT * FROM employee_systime
FOR SYSTEM TIME BETWEEN TIMESTAMP'1900-01-01 22:14:02.820000-08:00'
and CURRENT_TIMESTAMP
WHERE ename='Ash';
```

eid	ename	deptno	sys_start		sys_end	
1002	Ash	888	2014-02-23 22:27:56.540000-08:00	9999-12-31 23:59:59.999999+00:00		
1002	Ash	333	2003-07-01 12:11:00.000000-08:00	2014-02-23 22:27:56.540000-08:00		

In this way, a system-time table maintains a complete history of changes to the rows in the table.

Defining Triggers for System-Time Tables

Triggers can be defined for INSERT, DELETE, and UPDATE operations on system-time tables. Note the following:

- Actions are triggered only for actions on open rows, that is, rows that have not been logically deleted from the table.
- Defined actions cannot reference the component beginning or ending bound `TIMESTAMP` columns of the `SYSTEM_TIME` derived period column.
- The DELETE or UPDATE operation that triggers an action can be unqualified or can include the temporal `FOR PORTION OF` qualifier.
- Trigger actions on ANSI temporal tables can be qualified with `FOR PORTION OF`.

HELP and SHOW Statements

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

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

(Note that temporal relationship constraints are allowed on ANSI temporal tables, but are not themselves ANSI-compliant.)

- **HELP COLUMN** also includes fields named **Without Overlaps Unique** for ANSI temporal tables that include a valid-time column. The values of this field can be Y or N.
- **HELP CONSTRAINT** has a **Type** field that shows information about named constraints, including temporal constraints. Note that **HELP CONSTRAINT** shows information only about named constraints. Use **SHOW TABLE** to see information about unnamed constraints defined on tables.
- **HELP SESSION** includes a **Temporal Qualifier** field that shows **ANSIQUALIFIER** if the session temporal qualifier is set to **ANSIQUALIFIER**, a requirement for working with ANSI temporal tables, that is normally set automatically.
- **HELP TRIGGER** includes **ValidTime Type** and **TransactionTime Type** fields that display A to indicate trigger has been created on an ANSI temporal table.
- **SHOW TABLE** displays these additional items for ANSI temporal tables:
 - **VALIDTIME** and **SYSTEM_TIME** derived period columns and their component beginning and ending bound **Date/Timestamp** columns.
 - **SYSTEM VERSIONING** for system-versioned system-time tables.
 - temporal **PRIMARY KEY** and **UNIQUE** constraint definitions having **WITHOUT OVERLAPS**.
 - Temporal referential integrity constraints with referencing and referenced period specifications.

Cursors and ANSI System-Time Table 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 system-versioned system-time tables apply to DML associated with a cursor, with some limitations that relate to positioned (updatable) cursors:

- 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** or **DELETE** statement that opens an updatable cursor has the same syntax as described in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146, but for system-versioned system-time tables the **FROM** clause cannot include the special ANSI temporal **FOR SYSTEM TIME** clause.

Related Information

For more information on...	See...
cursors	<i>Teradata Vantage™ - SQL Stored Procedures and Embedded SQL</i> , B035-1148

Working With ANSI Valid-Time Tables

Valid time is the time period during which the information in a row is in effect or true for purposes of real-world application. (ANSI calls this period "application time.") 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.

Use valid-time tables when the information in table rows is delimited by time, and for which row information should be maintained, tracked, and manipulated in a time-aware fashion.

Valid-time tables are 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.

Note:

This material covers only the syntax, rules, and other details that apply to ANSI temporal tables. The syntax diagrams presented are extracts of the full diagrams, and focus on the temporal syntax.

Most of the existing rules and options that apply to conventional, nontemporal versions of the SQL statements discussed here also apply to the temporal statements. The nontemporal rules and options are not repeated here. For more information on conventional, nontemporal SQL DDL and DML statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 .

Creating ANSI Valid-Time Tables

ANSI supports special CREATE TABLE and ALTER TABLE syntax for creating valid-time tables. Creating temporal tables includes:

- Defining the beginning and ending bound columns for the valid-time period
- Defining the VALIDTIME derived period column
- Optional syntax to define primary key, unique, and referential constraints for ANSI valid-time tables.

CREATE TABLE (ANSI Valid-Time Table Form)

Create a new ANSI valid-time table.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

[AS] VALIDTIME is not ANSI compliant, but must be included in valid-time tables defined in the database.

CREATE TABLE Syntax (ANSI Valid-Time Table Form)

```
CREATE MULTISET TABLE table_name (
  valid_start { DATE | TIMESTAMP [ ( precision ) ] [ WITH TIME ZONE ] } NOT NULL,
  valid_end { DATE | TIMESTAMP [ ( precision ) ] [ WITH TIME ZONE ] } NOT NULL,
  PERIOD FOR valid_time_period ( valid_start, valid_end ) [AS] VALIDTIME
) [;]
```

CREATE TABLE Syntax Elements (ANSI Valid-Time Table Form)

table_name

The name of the valid-time table. May include database qualifier.

valid_start

The name of the column that will store the beginning bound of the valid-time period.

valid_end

The name of the column that will store the ending bound of the valid-time period.

precision

The precision of the timestamp value. The default is 6.

valid_time_period

The name of the valid-time derived period column.

VALIDTIME

Required to create valid-time tables in the database.

This clause allows SELECT statements on valid-time tables to use special temporal qualifiers (AS OF, BETWEEN ... AND, FROM TO, CONTAINED IN), which ANSI does not support on valid-time tables. For more information see [Querying ANSI Valid-Time Tables](#).

Note:

[AS] VALIDTIME is a Teradata extension to ANSI.

Usage Notes

General

- Valid time is the Teradata implementation of what ANSI calls “application time.”
- The *valid_start* and *valid_end* columns must be defined as NOT NULL.
- The data types of the *valid_start* and *valid_end* columns must match.
- To function as a temporal table, the valid-time table must be defined AS VALIDTIME. System-time tables do not have valid time.
- A table can have only one valid-time period definition.
- A valid-time table cannot be a queue, error, or global temporary trace table.
- Statistics cannot be collected on the valid-time derived period column, but they can be collected on the component start and end time columns.
- Algorithmic compression (ALC) is not allowed on DateTime columns that act as the beginning and ending bound values of a temporal derived period column.

Row Partitioning ANSI Valid-Time Tables

Temporal tables should be row partitioned to improve query performance. Partitioning can logically group the table rows into current and history rows. Queries of current rows are directed automatically to the partition containing the current 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.

Example: Row Partitioning an ANSI Valid-Time Table

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

```
CREATE MULTISET TABLE employee_vt (
    eid INTEGER NOT NULL,
    ename VARCHAR(5) NOT NULL,
    terms VARCHAR(5),
    job_start DATE NOT NULL,
    job_end DATE NOT NULL,
```

```

        PERIOD FOR job_dur(job_start,job_end) AS VALIDTIME
    ) PRIMARY INDEX(eid)
    PARTITION BY
        CASE_N(
            END(job_dur)>= CURRENT_DATE AT INTERVAL - 12:59' HOUR TO MINUTE,
            NO CASE);

```

Note:

The partitioning expression could have used `job_end` instead of `END(job_dur)`.

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.

Example: Creating an ANSI Valid-Time Table

The following example creates a valid-time table.

```

CREATE MULTISET TABLE employee_valid_time (
    eid INTEGER NOT NULL,
    ename VARCHAR(5) NOT NULL,
    terms VARCHAR(5),
    job_start DATE NOT NULL,
    job_end DATE NOT NULL,
    PERIOD FOR job_dur(job_start,job_end) AS VALIDTIME
)
PRIMARY INDEX (eid);

```

CREATE TABLE AS (ANSI Valid-Time Table Form)

Creates a new temporal table by copying all or a portion of the structure and contents of an existing temporal table.

Note:

System-time tables cannot be source tables for CREATE TABLE AS statements.

Syntax

There is no special additional temporal syntax for the CREATE TABLE AS statement. The syntax is identical to that used for nontemporal tables. That syntax is described fully in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. But note the restrictions listed in the Usage Notes below.

Usage Notes for ANSI Valid-Time Tables

- In order to copy the derived VALIDTIME period definition to the new table, you must use the table-copy form: CREATE TABLE *target_table* AS *source_table* rather than CREATE TABLE *target_table* AS (*subquery*).
- If you specify new column names in the table-copy form of a CREATE TABLE AS statement, you must specify a name for the valid-time derived period column, which is also copied from the source valid-time table.
- You cannot, and need not, explicitly specify the [AS] VALIDTIME column attribute in a CREATE TABLE AS statement, even if you specify new column names for the target table.
- 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 valid-time tables having primary key and unique constraints, the database automatically creates 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 tables, and create unique secondary indexes, rather than SJIs.)

ALTER TABLE (ANSI Valid-Time Table Form)

Temporal syntax for ALTER TABLE allows you to create ANSI temporal valid-time tables from existing tables. It can be used to modify existing tables that may have already been created having DateTime columns that manually track the effective period during which information in the row is effective.

Adding valid time to the table involves these ALTER TABLE operations:

- Adding or altering the two DateTime columns that will serve as the beginning and ending bounds of the valid-time period
- Adding a valid-time derived period column to the table, and designating the derived period column VALIDTIME

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

ALTER TABLE Syntax (ANSI Valid-Time Table Form)

Use the following ALTER TABLE syntax to create a valid-time table by adding a valid-time derived period column:

```
ALTER TABLE table_name
  ADD PERIOD FOR valid_time_period ( valid_start, valid_end ) [AS] VALIDTIME [;]
```

Use the following syntax to drop a valid-time derived period column:

```
ALTER TABLE table_name
  DROP valid_time_period [ WITHOUT DELETE ] [;]
```

ALTER TABLE Syntax Elements (ANSI Valid-Time Table Form)

table_name

The name of the valid-time table. May include database qualifier.

valid_time_period

The name of the valid-time derived period column.

valid_start

The name of the column that will store the beginning bound of the valid-time period.

valid_end

The name of the column that will store the ending bound of the valid-time period.

[AS] VALIDTIME

Required to create valid-time tables in Vantage.

The [AS] VALIDTIME clause allows SELECT statements on valid-time tables to use special temporal qualifiers (AS OF, BETWEEN, FROM TO, CONTAINED IN), which ANSI does not support on valid-time tables. For more information see [Valid-Time Modifications](#).

Note:

[AS] VALIDTIME is a Teradata extension to ANSI.

WITHOUT DELETE

Prevents automatic deletion of history and future rows when the valid-time derived period column is dropped from a table.

Note:

WITHOUT DELETE is a Teradata extension to ANSI.

Usage Notes

- [AS] VALIDTIME is an extension to ANSI. It is required to create valid-time tables in Vantage. The [AS] VALIDTIME clause allows SELECT statements on valid-time tables to use special temporal qualifiers (AS OF, BETWEEN ... AND, FROM TO, CONTAINED IN), which ANSI does not support on valid-time tables. For more information, see [Querying ANSI Valid-Time Tables](#).
- Dropping the valid-time derived period column from a valid-time table deletes all history and future rows from the table unless the WITHOUT DELETE option is used.

Example: ALTER TABLE to Convert a Nontemporal Table to a Valid-Time Table

The following SQL creates a regular nontemporal table, and inserts some rows:

```
CREATE MULTISET TABLE employee_vt (
  eid INTEGER NOT NULL,
  ename VARCHAR(5) NOT NULL,
  terms VARCHAR(5),
  job_start DATE NOT NULL,
```



```
job_end DATE NOT NULL
) PRIMARY INDEX(eid);
```

After rows have been inserted, an unqualified SELECT on the table returns all rows:

```
SELECT * FROM employee_systime;

  eid ename terms  job_start    job_end
---- -
1002 Ash      TA05 2003/01/01  2003/12/31
1005 Alice    TW10 2004/12/01  2005/12/01
1010 Mike     TW07 2015/01/01  2016/12/31
1005 Alice    PW11 2005/12/01  9999/12/31
1001 Sania    TW08 2002/01/01  2006/12/31
1004 Fred     PW12 2001/05/01  9999/12/31
1003 SRK      TM02 2004/02/10  2005/02/10
```

The following ALTER TABLE statement changes the table into a valid-time temporal table:

```
ALTER TABLE employee_vt
ADD PERIOD FOR job_dur(job_start,job_end) AS VALIDTIME;
```

Unlike for system-time tables, an unqualified SELECT of a valid-time table shows all rows in the table, regardless of whether their valid-time period is passed, current, or in the future:

```
SELECT * FROM employee_systime;

  eid ename terms  job_start    job_end
---- -
1002 Ash      TA05 2003/01/01  2003/12/31
1005 Alice    TW10 2004/12/01  2005/12/01
1010 Mike     TW07 2015/01/01  2016/12/31
1005 Alice    PW11 2005/12/01  9999/12/31
1001 Sania    TW08 2002/01/01  2006/12/31
1004 Fred     PW12 2001/05/01  9999/12/31
1003 SRK      TM02 2004/02/10  2005/02/09
```

Automatic temporal behavior is evident in valid-time tables when using temporal qualifiers for SELECT queries, and for UPDATE, and DELETE modifications. For more information, see [Querying ANSI Valid-Time Tables](#) and [Modifying Rows in ANSI Valid-Time Tables](#).

INSERT (ANSI Valid-Time Table Form)

Add new rows to temporal tables.

Syntax

There is no special temporal syntax for inserting rows into temporal tables. Use the standard SQL INSERT statement. However, note the following:

- You must include values for the beginning and ending bound columns that constitute the valid-time derived period column.
- As for any type of derived period column in the database, you cannot insert Period type values for the derived period column itself.

Example: Inserting Rows into a Valid-Time Table

```
INSERT INTO employee_valid_time VALUES
  (1001, 'Sania', 'TW08', DATE '2002-01-01', DATE '2006-12-31');
INSERT INTO employee_vt VALUES
  (1004, 'Fred', 'PW12', DATE '2001-05-01', UNTIL_CHANGED);
INSERT INTO employee_vt VALUES
  (1002, 'Ash', 'TA05', DATE '2003-01-01', DATE '2003-12-31');
INSERT INTO employee_vt VALUES
  (1003, 'SRK', 'TM02', DATE '2004-02-10', DATE '2005-02-10');
INSERT INTO employee_vt VALUES
  (1005, 'Alice', 'TW10', DATE '2004-12-01', DATE '2005-12-01');
INSERT INTO employee_vt VALUES
  (1005, 'Alice', 'PW11', DATE '2005-12-01', UNTIL_CHANGED);
INSERT INTO employee_vt VALUES
  (1010, 'Mike', 'TW07', DATE '2015-01-01', DATE '2016-12-31');
```

Note:

Values for the start and end columns that constitute the valid-time period must be provided in the INSERT statement. UNTIL_CHANGED is a Teradata extension to ANSI that can be used when inserting rows into valid-time tables. It resolves to the maximum system DATE or TIMESTAMP value in accordance with the data type defined for the valid-time period ending bound column.

Usage Notes for Creating ANSI Valid Time Tables

Primary Key and Unique Constraints for ANSI Valid-Time Tables

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Valid-time table definitions can include primary key and unique constraints that prevent rows from having valid-time periods that overlap. A system-defined join index is created automatically for constraint checking during modifications to the table.

Syntax

```
{ PRIMARY KEY | UNIQUE } (
  column_list [, valid_time_period WITHOUT OVERLAPS ]
)
```

column_list

The column name or the comma-separated list of columns that serve as the primary key or uniqueness constraint for the table.

valid_time_period

The name of the valid-time derived period column.

WITHOUT OVERLAPS

Specifies that valid-time periods of the table rows cannot overlap.

Usage Notes

- If *valid_time_period*WITHOUT OVERLAPS is not specified, the constraint acts as a nontemporal constraint and imposes uniqueness on the values disregarding whether valid-time periods overlap.
- If the table includes a SYSTEM_TIME derived period column, the start and end column components of the derived period column cannot be included in the *column_list*.

Example: Temporal Primary Key Constraint on a Valid-Time Table

```
CREATE MULTISET TABLE employee_vt_pk (
  eid INTEGER NOT NULL,
  ename VARCHAR(5) NOT NULL,
  terms VARCHAR(5),
  job_start DATE NOT NULL,
  job_end DATE NOT NULL,
  PERIOD FOR job_dur(job_start,job_end) AS VALIDTIME,
  PRIMARY KEY(deptno, job_dur WITHOUT OVERLAPS)
)
PRIMARY INDEX (eid);
```


Temporal Referential Constraints for ANSI Valid-Time Tables

Temporal 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, which must include the valid-time column as part of the FK, must exist in the corresponding referenced columns (the primary key (PK)) of the parent table. Consequently, the valid-time value of every row in the child table must exist as a valid-time value in the parent table.

These constraints are not enforced by Vantage so are sometimes referred to as "soft referential integrity." 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.

Temporal referential constraints can be defined for valid-time period columns in temporal tables by specifying the valid-time derived period column names with the special PERIOD keyword.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Syntax

```
FOREIGN KEY ( fk_column_list, PERIOD fk_valid_time_period )
REFERENCES WITH NO CHECK OPTION
 ( pk_column_list, PERIOD pk_valid_time_period )
```

fk_column_list

The column name or the comma-separated list of columns that serve as the foreign key for the child table. These columns reference and correspond to the *pk_column_list*. Every value in the columns of the *fk_column_list* columns must exist in the corresponding columns of the *pk_column_list*.

fk_valid_time_period

The name of the valid-time derived period column in the child table.

REFERENCES WITH NO CHECK OPTION

Specifies that this relationship is a referential constraint, and is not enforced by the database.

WITH NO CHECK OPTION is a Teradata extension to ANSI.

table_name

The name of the table referenced by the foreign key.

pk_column_list

The column name or the comma-separated list of columns that serve as the primary key for the parent table.

pk_valid_time_period

The name of the valid-time derived period column in the parent table.

Usage Notes

Because this is a relationship between valid-time columns, the referencing table and referenced table in a PK-FK temporal referential constraint relationship can be either a valid-time or bitemporal table. The table types do not need to match. For more information on bitemporal tables, see [Working With ANSI Bitemporal Tables](#).

Example: Temporal Referential Constraint on an ANSI Valid-Time Table

```
CREATE MULTISET TABLE hire_contracts(
  h_eid INTEGER NOT NULL,
  h_name VARCHAR(5) NOT NULL,
  h_terms VARCHAR(5),
  contract_start DATE NOT NULL,
  contract_end DATE NOT NULL,
  PERIOD FOR hire_period(contract_start,contract_end) AS VALIDTIME,
  PRIMARY KEY(h_eid, hire_period WITHOUT OVERLAPS)
)
PRIMARY INDEX(h_eid);

CREATE MULTISET TABLE employee_vt_ri (
  eid INTEGER NOT NULL,
  ename VARCHAR(5) NOT NULL,
  terms VARCHAR(5),
  job_start DATE NOT NULL,
  job_end DATE NOT NULL,
  PERIOD FOR job_dur(job_start,job_end) AS VALIDTIME,
  UNIQUE(eid,job_dur WITHOUT OVERLAPS),
  FOREIGN KEY(eid,PERIOD job_dur) REFERENCES WITH NO CHECK OPTION
  hire_contracts(h_eid, PERIOD hire_period)
)
PRIMARY INDEX (eid);
```

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)
);
```

Bulk Loading Data into Temporal Tables

Vantage supports two 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.
- Alternatively, 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 and ALTER TABLE to convert the tables into valid-time tables.

Querying ANSI Valid-Time Tables

For valid-time tables, these qualifiers are Teradata extensions to ANSI. ANSI does not support AS OF, BETWEEN ... AND, and FROM TO temporal qualifiers for queries of valid-time tables.

FROM Clause (ANSI Valid-Time Table Form)

Uses system-time dimension criteria to determine which rows in ANSI valid-time tables are subject to a SELECT query.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

ANSI does not support AS OF, BETWEEN ... AND, and FROM TO temporal qualifiers for queries of valid-time tables.

FROM Clause Syntax (ANSI Valid-Time Table Form)

```
FROM valid_time_table [
  [FOR] VALIDTIME {
    AS OF point_in_time |
    BETWEEN point_in_time_1 AND point_in_time_2 |
    FROM point_in_time_1 TO point_in_time_2 |
    CONTAINED IN ( point_in_time_1, point_in_time_2 )
  }
]
```

FROM Clause Syntax Elements (ANSI Valid-Time Table Form)

valid_time_table

The valid-time table being queried.

AS OF

Qualifies rows in valid-time tables with valid-time periods that include the specified point in time. This means the information in the row is, was, or will be valid at the specified point in time.

Note:

Although the returned rows are valid at the time specified by the query, they may not be valid at the time the query is submitted.

point_in_time

point_in_time_1

point_in_time_2

Identifies the point in valid time or delimits the period in valid time for which rows will be returned.

A date or timestamp expression that can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a DATE or TIMESTAMP[(n)] [WITH TIME ZONE] value.

The expression can be any expression, including parameterized values and built-in functions such as CURRENT_DATE, CURRENT_TIMESTAMP, TEMPORAL_DATE, or TEMPORAL_TIMESTAMP. The expression cannot reference any columns, but it can be a self-contained noncorrelated scalar subquery.

BETWEEN ... AND

Qualifies all rows with valid-time periods that overlap or immediately succeed the period defined by *point_in_time_1* and *point_in_time_2*.

Note:

This is not the commonly understood meaning of “between” because BETWEEN will qualify rows with valid-time periods that begin before by *point_in_time_1*, and rows that start immediately after *point_in_time_2* and extend beyond that. For a qualifier that reflects the commonly understood meaning of BETWEEN, use the CONTAINED IN qualifier.

FROM TO

Qualifies all rows with valid-time periods that overlap the period defined by *point_in_time_1* and *point_in_time_2*.

CONTAINED IN

Qualifies all rows with valid-time periods that are between *point_in_time_1* and *point_in_time_2*. The valid-time period starts at or after *point_in_time_1*, and ends before or at *point_in_time_2*.

Note:

CONTAINED IN is a Teradata extension to ANSI.

Usage Notes

- If no temporal qualifiers are used in the FROM clause, the default for valid-time tables is to qualify all rows in the table, regardless of validity.
- If temporal qualifiers are used in the FROM clause, the valid-time start and end columns are not returned in the results.
- BETWEEN ... AND, FROM TO, and CONTAINED IN all qualify queries of temporal tables according to a period of time. Any table rows that were valid at any time during the specified period (or immediately after it in the case of BETWEEN ... AND) are qualified for the query. The differences in the three temporal qualifiers are subtle, but allow you to define your query to qualify a precise set of rows.

FROM Clause Examples (ANSI Valid-Time Table Form)

A simple SELECT on a valid-time table with no temporal qualifiers shows all rows in the table, regardless of the valid-time periods. Assume this valid-time table has a valid-time derived period column, job_dur, that represents the duration from job_start to job_end.

```
SELECT * FROM employee_systime;
```

eid	ename	terms	job_start	job_end
1002	Ash	TA05	2003/01/01	2003/12/31
1005	Alice	TW11	2004/12/01	2005/12/01
1010	Mike	TW07	2015/01/01	2016/12/31
1005	Alice	PW11	2005/12/01	9999/12/31
1001	Sania	TW08	2002/01/01	2006/12/31
1004	Fred	PW12	2001/05/01	9999/12/31
1003	SRK	TM02	2004/02/10	2005/02/09

Rows with job_end dates that are 9999-12-31 are valid indefinitely.

Notice two rows for Alice with non-overlapping dates, indicating that the terms for Alice's job contract changed as of December, 1, 2005. Even though one row ends at 2005-12-01 and the other starts at 2005-12-01, the rows do not "overlap" because the true duration of the period derived from the job_start and job_end columns is considered to be inclusive of the beginning bound and exclusive of the ending bound.

Temporal qualifiers allow you to qualify rows that are, were, or will be valid at any time or during any period you specify.

Example: AS OF Queries on an ANSI Valid-Time Table

The following AS OF query retrieves all table rows that were valid at 2002-01-01.

```
SELECT *
FROM employee_vt
FOR VALIDTIME AS OF DATE'2002-01-01';
```

eid	ename	terms	job_start	job_end
1001	Sania	TW08	2002/01/01	2006/12/31
1004	Fred	PW12	2001/05/01	9999/12/31

Only Sania and Fred had jobs with valid-time periods that include 2002-01-01.

Temporal queries can specify times in the future to return the rows that will be, or will still be valid at a future time:

```
SELECT *
FROM employee_vt
FOR VALIDTIME AS OF DATE'2015-02-01';
```

eid	ename	terms	job_start	job_end
1005	Alice	PW11	2005/12/01	9999/12/31
1010	Mike	TW07	2015/01/01	2016/12/31
1004	Fred	PW12	2001/05/01	9999/12/31

Example: CONTAINED IN Query on an ANSI Valid-Time Table

A CONTAINED IN query will return rows that were valid only within a specified period. Rows with valid-time periods that may start within the specified period, but continue beyond the period are not returned.

```
SELECT *
FROM employee_vt
FOR VALIDTIME CONTAINED IN (DATE'2004-01-01',DATE'2005-12-31');
```

eid	ename	terms	job_start	job_end
-----	-------	-------	-----------	---------

```
1005 Alice TW11 2004/12/01 2005/12/01
1003 SRK TM02 2004/02/10 2005/02/09
```

Example: FROM TO Query on ANSI Valid-Time Table

This query specifies the same period as the CONTAINED IN example, but in the case of a FROM TO qualifier, any rows with valid-time periods that overlap the specified time period will be returned.

```
SELECT *
FROM employee_vt
FOR VALIDTIME FROM DATE'2004-01-01' TO DATE'2005-12-31';
```

eid	ename	terms	job_start	job_end
1005	Alice	TW11	2004/12/01	2005/12/01
1004	Fred	PW12	2001/05/01	9999/12/31
1005	Alice	PW11	2005/12/01	9999/12/31
1003	SRK	TM02	2004/02/10	2005/02/09
1001	Sania	TW08	2002/01/01	2006/12/31

Modifying Rows in ANSI Valid-Time Tables

When modifications are made to rows in temporal tables, Vantage provides automatic handling of the valid-time periods as needed to maintain the information in a time-aware fashion.

DELETE and UPDATE modifications to valid-time tables can use the FOR PORTION OF qualifier to place time bounds on the effectiveness of the modifications. The time during which a change is effective is called the period of applicability (PA) of the modification. Vantage selects the rows that are affected based on the relationship between the PA of the change and the valid-time period of the row, also called the period of validity (PV) of the row because that period defines when the information is considered to be in effect.

The relationship between the PA of a modification and PV of each row determines which rows qualify for the modification. It also determines when the change is effective and whether new rows will be automatically created in the table to account for data that changes during the valid-time period.

The mechanism of new rows being automatically generated by Vantage for modifications to temporal tables is described in more detail in [Modifying Temporal Tables](#).

DELETE (ANSI Valid-Time Table Form)

Deletes one or more rows from valid-time tables with the option of deleting the rows for only a portion of their valid-time periods.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

DELETE Syntax (ANSI Valid-Time Table Form)

```
DELETE FROM valid_time_table
  [ FOR PORTION OF valid_time_period_name
    FROM point_in_time_1 TO point_in_time_2
  ]
  [ WHERE search_condition ] [ ; ]
```

DELETE Syntax Elements (ANSI Valid-Time Table Form)

valid_time_table

The valid-time table from which you are deleting rows.

FOR PORTION OF

Qualifies rows for deletion that have valid-time periods that are within or that overlap the period defined by *point_in_time_1* and *point_in_time_2*.

valid_time_period_name

The name of the valid-time derived period column. This is not “VALIDTIME.” It is the name assigned to the derived period column when the table was defined.

point_in_time_1

point_in_time_2

Delimits the period of applicability of the deletion.

A date or timestamp expression that can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a DATE or TIMESTAMP[(n)] [WITH TIME ZONE] value.

The expression can be any expression, including parameterized values and built-in functions such as CURRENT_DATE, CURRENT_TIMESTAMP, TEMPORAL_DATE, or

TEMPORAL TIMESTAMP. The expression cannot reference any columns, but it can be a self-contained noncorrelated scalar subquery.

WHERE *search_condition*

For information on the this syntax and other nontemporal keywords, clauses, and options for DELETE, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Usage Notes

You cannot use the FOR PORTION OF qualifier within a MERGE INTO statement.

DELETE Examples (ANSI Valid-Time Table Form)

For the following examples, assume the queries are issued against the following valid-time table named `employee_vt` that contains a mix of current, future, and history rows:

eid	ename	terms	job_start	job_end
1002	Ash	TA05	2003/01/01	2003/12/31
1005	Alice	TW11	2004/12/01	2005/12/01
1010	Mike	TW07	2015/01/01	2016/12/31
1005	Alice	PW11	2005/12/01	9999/12/31
1001	Sania	TW08	2002/01/01	2006/12/31
1004	Fred	PW12	2001/05/01	9999/12/31
1003	SRK	TM02	2004/02/10	2005/02/09

Example: Simple DELETE on an ANSI Valid-Time Table

A simple DELETE statement without temporal qualification operates just as a DELETE on a nontemporal table, completely deleting rows that qualify for deletion:

```
DELETE FROM employee_vt WHERE ename='Ash';
```

```
SELECT * FROM employee_vt;
```

eid	ename	terms	job_start	job_end
1005	Alice	TW11	2004/12/01	2005/12/01
1010	Mike	TW07	2015/01/01	2016/12/31
1005	Alice	PW11	2005/12/01	9999/12/31
1001	Sania	TW08	2002/01/01	2006/12/31

```
1004 Fred    PW12  2001/05/01  9999/12/31
1003 SRK     TM02  2004/02/10  2005/02/09
```

Example: Delete from an ANSI Valid-Time Table Where PA of Deletion is Within PV of Row

Temporal tables allow you to “delete” rows for only a portion of their period of validity. The database takes care of adding rows and adjusting valid-time periods to account for the change automatically. The database automatically handles the valid-time modifications for the row, which may involve changing the period bounds and adding new rows to the table.

For example, assume the company grants Fred a year off from his job in 2009. Deleting the Fred row for only that portion of the row period of validity automatically yields two rows for Fred in the table:

```
DELETE FROM employee_vt
FOR PORTION OF job_dur FROM DATE'2009-01-01' TO DATE'2010-01-01'
WHERE ename='Fred';
```

```
SELECT * FROM employee_vt WHERE ename='Fred';
```

```
  eid ename terms  job_start      job_end
----
1004 Fred    PW12  2001/05/01  2009/01/01
1004 Fred    PW12  2010/01/01  9999/12/31
```

Note:

Even though this was a DELETE operation, the net effect was to add a row to the table.

Example: Delete from an ANSI Valid-Time Table where PA of Deletion Overlaps PV of Row

If the PV of the deletion overlaps the PA of a row, only the portion of the row information that is valid during the overlap is deleted, effectively changing the valid-time period for the row:

```
DELETE FROM employee_vt
FOR PORTION OF job_dur FROM DATE'2000-01-01' TO DATE'2002-01-01'
WHERE ename='Fred';
```

```
SELECT * FROM employee_vt WHERE ename='Fred';
```

```
  eid ename terms  job_start      job_end
----
```



```
1004 Fred    PW12 2002/02/01 2009/01/01
1004 Fred    PW12 2010/01/01 9999/12/31
```

```
DELETE FROM employee_vt
FOR PORTION OF job_dur FROM DATE'2008-05-05' TO DATE'2009-05-05'
WHERE ename='Fred';
```

```
SELECT * FROM employee_vt WHERE ename='Fred';
```

```
  eid ename terms  job_start    job_end
-----
1004 Fred    PW12 2002/01/01 2008/05/05
1004 Fred    PW12 2009/05/01 9999/12/31
```

UPDATE (ANSI Valid-Time Table Form)

Modifies one or more existing rows in valid-time tables with the option of modifying the rows for only a portion of their valid-time periods.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

UPDATE Syntax (ANSI Valid-Time Table Form)

```
UPDATE valid_time_table
  [ FOR PORTION OF valid_time_period_name
    FROM point_in_time_1 TO point_in_time_2
  ]
SET set_clause_list
  [ WHERE search_condition ] [ ; ]
```

UPDATE Syntax Elements (ANSI Valid-Time Table Form)

valid_time_table

The valid time table in which you are modifying rows.

FOR PORTION OF

Qualifies rows to be modified that have valid-time periods that are within or that overlap the period defined by *point_in_time_1* and *point_in_time_2*.

valid_time_period_name

The name of the valid-time derived period column. This is not “VALIDTIME.” It is the name assigned to the derived period column when the table was defined.

point_in_time_1

point_in_time_2

Delimits the period of applicability of the modification.

A date or timestamp expression that can be a constant, scalar UDF, scalar subquery, or business calendar function that evaluates to a DATE or TIMESTAMP[(n)] [WITH TIME ZONE] value.

The expression can be any expression, including parameterized values and built-in functions such as CURRENT_DATE, CURRENT_TIMESTAMP, TEMPORAL_DATE, or

TEMPORAL TIMESTAMP. The expression cannot reference any columns, but it can be a self-contained noncorrelated scalar subquery.

SET *set_clause_list*

WHERE *search_condition*

For information on the this syntax and other nontemporal keywords, clauses, and options for UPDATE, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Note:

The SET clause cannot include the start or end column components of the valid time period.

Usage Notes

You cannot use the FOR PORTION OF qualifier within a MERGE INTO statement.

UPDATE Examples (ANSI Valid-Time Table Form)

For the following examples, assume the queries are issued against the following valid-time table named `employee_vt` that contains a mix of current, future, and history rows:

eid	ename	terms	job_start	job_end
1002	Ash	TA05	2003/01/01	2003/12/31
1005	Alice	TW10	2004/12/01	9999/12/31
1010	Mike	TW07	2015/01/01	2016/12/31
1001	Sania	TW08	2002/01/01	2006/12/31
1004	Fred	PW12	2001/05/01	9999/12/31
1003	SRK	TM02	2004/02/10	2005/02/09

Example: Simple UPDATE on an ANSI Valid-Time Table

A simple UPDATE statement without temporal qualification operates just as an UPDATE on a nontemporal table, completely UPDATING rows that qualify for deletion:

```
UPDATE employee_vt SET terms='TA07' WHERE ename='Ash';
```

```
SELECT * FROM employee_vt;
```

eid	ename	terms	job_start	job_end
1002	Ash	TA07	2003/01/01	2003/12/31
1005	Alice	TW10	2004/12/01	9999/12/31
1010	Mike	TW07	2015/01/01	2016/12/31
1001	Sania	TW08	2002/01/01	2006/12/31
1004	Fred	PW12	2001/05/01	9999/12/31
1003	SRK	TM02	2004/02/10	2005/02/09

Example: UPDATE on an ANSI Valid-Time Table Where the PA of Update Overlaps a Portion of the PV of the Row

Temporal tables allow you to modify rows for only a portion of their period of validity. The database takes care of adding rows and adjusting valid-time periods to account for the change automatically. The database automatically handles the valid-time modifications for the row, which may involve changing the period bounds and adding new rows to the table.

For example, assume the company promotes Alice from employment terms TW10 to PW11 after she has been working for a year. Modifying the Alice row for only that portion of the row period of validity automatically yields two rows for Alice in the table:

```
UPDATE employee_vt
FOR PORTION OF job_dur FROM DATE '2005-12-01' TO DATE '9999-12-31'
SET terms='PW11'
WHERE ename='Alice';

SELECT * FROM employee_vt WHERE ename='Alice';
```

eid	ename	terms	job_start	job_end
1005	Alice	PW11	2005/12/01	9999/12/01
1005	Alice	TW10	2004/12/01	2005/12/01

Now the table has two rows for Alice that show how the terms of her employment changed, and the temporal extent of when she worked under each of the terms.

Example: UPDATE on an ANSI Valid-Time Table Where PA of Update is Within PV of Row

If the PV of the update lies within the PA of a row, only the portion of the row information that is valid during the overlap is updated, while the portions of row that existed before and after the change remain in the

table as separate rows. Assume that Fred has to reduce his work for the company for the year of 2005, and agrees to a change in the terms of his contract for that period:

```
UPDATE employee_vt
FOR PORTION OF job_dur FROM DATE '2005-01-01' TO DATE '2006-01-01'
SET terms='TW10'
WHERE ename='Fred';

SELECT * FROM employee_vt WHERE ename='Fred';
```

eid	ename	term	job_start	job_end
1004	Fred	TW10	2005/01/01	2006/01/01
1004	Fred	PW12	2001/05/01	2005/01/01
1004	Fred	PW12	2006/01/01	9999/12/31

Where there had been one row for Fred in the table before the UPDATE, after the UPDATE there are three rows. The three rows track the changing terms of Fred's employment, starting with terms PW12, changing to TW10 for one year beginning in 2005, then returning to PW12 in 2006.

Because this was a valid-time UPDATE executed against a valid-time table, all the work of creating new rows and adjusting the valid-times for each row was handled automatically by the database.

Defining Triggers for Valid-Time Tables

Triggers can be defined for INSERT, DELETE, and UPDATE operations on valid-time tables. Note the following restriction:

- For BEFORE triggers, defined actions cannot reference the component beginning or ending bound DateTime columns of the valid-time derived period column.
- The DELETE or UPDATE operation that triggers an action can be unqualified or can include the temporal FOR PORTION OF qualifier.
- Actions for DELETE and UPDATE triggers on ANSI temporal tables can be qualified with FOR PORTION OF.

HELP and SHOW Statements

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

- HELP COLUMN output includes a Temporal Column field that displays V, S, R, or N, to indicate a valid-time, system-time, temporal relationship constraint, or nontemporal column, respectively. (Note that temporal relationship constraints are allowed on ANSI temporal tables, but are not themselves ANSI-compliant.)
- HELP COLUMN also includes fields named Without Overlaps Unique for ANSI temporal tables that include a valid-time column. The values of this field can be Y or N.

- **HELP CONSTRAINT** has a **Type** field that shows information about named constraints, including temporal constraints. Note that **HELP CONSTRAINT** shows information only about named constraints. Use **SHOW TABLE** to see information about unnamed constraints defined on tables.
- **HELP SESSION** includes a **Temporal Qualifier** field that shows **ANSIQUALIFIER** if the session temporal qualifier is set to **ANSIQUALIFIER**, a requirement for working with ANSI temporal tables, that is normally set automatically.
- **HELP TRIGGER** includes **ValidTime Type** and **TransactionTime Type** fields that display **A** to indicate trigger has been created on an ANSI temporal table.
- **SHOW TABLE** displays these additional items for ANSI temporal tables:
 - **VALIDTIME** and **SYSTEM_TIME** derived period columns and their component beginning and ending bound **Date/Timestamp** columns.
 - **SYSTEM VERSIONING** for system-versioned system-time tables.
 - temporal **PRIMARY KEY** and **UNIQUE** constraint definitions having **WITHOUT OVERLAPS**.
 - Temporal referential integrity constraints with referencing and referenced period specifications.

Cursors and ANSI Valid-Time Table 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.

Cursors are not supported with ANSI valid-time tables, however they are supported on valid-time tables that use the **CURRENT** qualifier of Teradata non-ANSI temporal syntax. For information on using DML with cursors on valid-time tables, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

Related Information

For more information on...	See...
cursors	<i>Teradata Vantage™ - SQL Stored Procedures and Embedded SQL</i> , B035-1148

Working With ANSI Bitemporal Tables

You can combine system time and valid time in a single table to use the features of both:

- Adding a system-time dimension to a table lets you track all changes made to the table, leaving a permanent, persistent history of rows within the table itself.
- Adding a valid-time dimension to a table allows the database to automatically adjust the valid-time period data as modifications to the table are made that affect the effective dates of the information contained in each row.

Because system time and valid time are independent dimensions any change to a bitemporal table affects each dimension independently, and the consequences to the table are a combination of the effects in cases that have been described for individual system-time and valid-time tables in [Working With ANSI System-Time Tables](#) and [Working With ANSI Valid-Time Tables](#)

Creating ANSI Bitemporal Tables

Bitemporal tables include a system-time and valid-time dimension. The syntax for creating bitemporal tables combines syntax for creating both a SYSTEM_TIME and valid-time derived period column.

CREATE TABLE (ANSI Bitemporal Table Form)

Create a new ANSI bitemporal table.

Syntax

There is no special temporal syntax for creating bitemporal tables. It simply combines the forms discussed in [CREATE TABLE \(ANSI System-Time Table Form\)](#) and [CREATE TABLE \(ANSI Valid-Time Table Form\)](#). Note that a bitemporal table also combines the rules, restrictions, and usage notes listed for both system-time and valid-time tables..

For example, a bitemporal table cannot be the source table for a CREATE TABLE AS statement, and statistics cannot be collected on the derived period columns of bitemporal tables.

Note:

All restrictions for system-time and valid-time columns described in [Working With ANSI System-Time Tables](#) and [Working With ANSI Valid-Time Tables](#) apply to bitemporal tables.

Example: Creating an ANSI Bitemporal Table

The following example creates an ANSI bitemporal table.

```
CREATE MULTISET TABLE employee_bitemporal (  
  eid INTEGER NOT NULL,  
  ename VARCHAR(5),  
  deptno INTEGER NOT NULL,  
  terms VARCHAR(5),  
  job_start DATE NOT NULL,  
  job_end DATE NOT NULL,  
  PERIOD FOR job_dur(job_start,job_end) AS VALIDTIME,  
  sys_start TIMESTAMP(6) WITH TIME ZONE NOT NULL  
    GENERATED ALWAYS AS ROW START,  
  sys_end TIMESTAMP(6) WITH TIME ZONE NOT NULL  
    GENERATED ALWAYS AS ROW END,  
  PERIOD FOR SYSTEM_TIME(sys_start,sys_end)  
)  
PRIMARY INDEX (eid) WITH SYSTEM VERSIONING;
```


ALTER TABLE (ANSI Bitemporal Table Form)

Temporal syntax for ALTER TABLE allows you to create ANSI bitemporal tables from existing nontemporal tables, and combine the ALTER TABLE syntaxes for system-time and valid-time tables.

Adding valid time to the table involves these ALTER TABLE operations:

- Adding or altering the two DateTime columns that will serve as the beginning and ending bounds of the valid-time period
- Adding a valid-time derived period column to the table, and designating the derived column VALIDTIME

Adding system time to the table involves these ALTER TABLE operations:

- Adding a SYSTEM_TIME derived period column to the table by specifying the columns that will serve as the beginning and ending bounds of the system-time period.
- Adding or altering the columns that will serve as the beginning bound and ending bound of the system-time period. If these columns already exist in the table, special attributes must be added to them.
- Adding system versioning to the table

Syntax

There is no special syntax for altering tables to bitemporal tables. Use the combined special syntax that is discussed in [ALTER TABLE \(ANSI System-Time Table Form\)](#) and [ALTER TABLE \(ANSI Valid-Time Table Form\)](#).

Note:

The table must be altered to add a valid-time dimension prior to adding the system-time dimension, because ALTER TABLE operations on system-time tables are severely restricted.

Example: ALTER TABLE to Convert a Nontemporal Table to an ANSI Bitemporal Table

The following SQL creates a regular nontemporal table, and inserts some rows:

```
CREATE MULTISET TABLE employee_bitemp (
  eid INTEGER NOT NULL,
  ename VARCHAR(5),
  deptno INTEGER NOT NULL,
  terms VARCHAR(5),
  job_start DATE NOT NULL,
  job_end DATE NOT NULL,
  sys_start TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  sys_end TIMESTAMP(6) WITH TIME ZONE NOT NULL
)
PRIMARY INDEX (eid);

INSERT INTO employee_bitemp VALUES

(1001, 'Sania', 111, 'TW08', DATE '2002-01-01', DATE '2006-12-31',
  TIMESTAMP '2002-01-01 00:00:00.000000-08:00',
  TIMESTAMP '2002-07-01 12:00:00.350000+00:00');
INSERT INTO employee_bitemp VALUES
(1004, 'Fred', 222, 'PW12', DATE '2001-05-01', DATE '9999-12-31',
  TIMESTAMP '2001-05-01 12:00:00.350000-08:00',
  TIMESTAMP '9999-12-31 23:59:59.999999+00:00');
INSERT INTO employee_bitemp VALUES
(1002, 'Ash', 333, 'TA05', DATE '2003-01-01', DATE '2003-12-31',
  TIMESTAMP '2003-01-01 12:11:00.000000-08:00',
  TIMESTAMP '9999-12-31 23:59:59.999999+00:00');
INSERT INTO employee_bitemp VALUES
(1003, 'SRK', 111, 'TM02', DATE '2004-02-10', DATE '2005-02-10',
```

```

    TIMESTAMP'2004-02-10 00:00:00.000000-08:00',
    TIMESTAMP'2004-12-01 00:12:23.120000+00:00');
INSERT INTO employee_bitemp VALUES
(1005,'Alice',222,'TW10',DATE'2004-12-01',DATE'9999-12-31',
TIMESTAMP'2004-12-01 12:00:00.450000-08:00',
TIMESTAMP'9999-12-31 23:59:59.999999+00:00');
INSERT INTO employee_bitemp VALUES
(1010,'Mike',444,'TW07',DATE'2015-01-01',DATE'2016-12-31',
TIMESTAMP'2004-12-01 00:12:23.120000-08:00',
TIMESTAMP'9999-12-31 23:59:59.999999+00:00');

```

After rows have been inserted, an unqualified SELECT on the table returns all rows:

```
SELECT * FROM employee_bitemp;
```

eid	ename	deptno	terms	job_start	job_end	sys_start	sys_end
1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-01-01 12:11:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1005	Alice	222	TW10	2004/12/01	9999/12/31	2004-12-01 12:00:00.450000-08:00	9999-12-31 23:59:59.999999+00:00
1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01 00:12:23.120000-08:00	9999-12-31 23:59:59.999999+00:00
1001	Sania	111	TW08	2002/01/01	2006/12/31	2002-01-01 00:00:00.000000-08:00	2002-07-01 12:00:00.350000+00:00
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01 12:00:00.350000-08:00	9999-12-31 23:59:59.999999+00:00
1003	SRK	111	TM02	2004/02/10	2005/02/10	2004-02-10 00:00:00.000000-08:00	2004-12-01 00:12:23.120000+00:00

The following ALTER TABLE statement adds valid time to the table:

```

ALTER TABLE employee_bitemp
ADD PERIOD FOR job_dur(job_start,job_end) AS VALIDTIME;

```

A simple select from the table still returns all rows, as it would from any valid-time table:

```
SELECT * FROM employee_bitemp;
```

eid	ename	deptno	terms	job_start	job_end	sys_start	sys_end
1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-01-01 12:11:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1005	Alice	222	TW10	2004/12/01	9999/12/31	2004-12-01 12:00:00.450000-08:00	9999-12-31 23:59:59.999999+00:00

1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01	00:12:23.120000-08:00	9999-12-31	23:59:59.999999+00:00
1001	Sania	111	TW08	2002/01/01	2006/12/31	2002-01-01	00:00:00.000000-08:00	2002-07-01	12:00:00.350000+00:00
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01	12:00:00.350000-08:00	9999-12-31	23:59:59.999999+00:00
1003	SRK	111	TM02	2004/02/10	2005/02/10	2004-02-10	00:00:00.000000-08:00	2004-12-01	00:12:23.120000+00:00

The following ALTER TABLE statements add system time to the table:

```
ALTER TABLE employee_bitemp
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;
ALTER TABLE employee_bitemp
ADD SYSTEM VERSIONING;
```

Because the table now includes a system-time dimension, rows that had dates for sys_end that were not 9999-12-31 23:59:59.999999+0:0 are considered closed rows, logically deleted from the database, so now a simple select shows only four rows:

```
SELECT * FROM employee_bitemp;
```

eid	ename	deptno	terms	job_start	job_end	sys_start	sys_end
1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-01-01 12:11:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1005	Alice	222	TW10	2004/12/01	9999/12/31	2004-12-01 12:00:00.450000-08:00	9999-12-31 23:59:59.999999+00:00
1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01 00:12:23.120000-08:00	9999-12-31 23:59:59.999999+00:00
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01 12:00:00.350000-08:00	9999-12-31 23:59:59.999999+00:00

INSERT (ANSI Bitemporal Table Form)

Add new rows to ANSI bitemporal tables.

Syntax

There is no special temporal syntax for inserting rows into bitemporal tables. Use the standard SQL INSERT statement. However, all the restrictions listed for INSERTs into system-time and valid-time tables apply to insertions into bitemporal tables. To review these restrictions, see [INSERT \(ANSI System-Time Table Form\)](#) and [INSERT \(ANSI Valid-Time Table Form\)](#).

Note especially that values must be provided for the component columns of the valid-time and system-time periods, and the values entered for the system-time columns will be replaced with a beginning bound value of TEMPORAL_TIMESTAMP (the timestamp at the time of the insertion) and the ending bound of 9999-12-31 23:59:59.999999+00:00.

Row Partitioning ANSI Bitemporal Tables

Temporal tables should be row partitioned to improve query performance. Partitioning can logically group the table rows into current and history, open and closed rows. For bitemporal tables, queries of current and open rows are directed automatically to the partition containing these 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.

Example: Row Partitioning an ANSI Valid-Time Table

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

```
CREATE MULTISET TABLE employee_bitemp (
  eid INTEGER NOT NULL,
  ename VARCHAR(5),
  deptno INTEGER NOT NULL,
  terms VARCHAR(5),
  job_start DATE NOT NULL,
  job_end DATE NOT NULL,
  PERIOD FOR job_dur (job_start,job_end) AS VALIDTIME,
  sys_start TIMESTAMP(6) WITH TIME ZONE NOT NULL
    GENERATED ALWAYS AS ROW START,
  sys_end TIMESTAMP(6) WITH TIME ZONE NOT NULL
    GENERATED ALWAYS AS ROW END,
  PERIOD FOR SYSTEM_TIME(sys_start,sys_end)
)
```

```

PRIMARY INDEX (eid)
PARTITION BY CASE_N (
  (END(job_dur) >= CURRENT_DATE AT INTERVAL -'12:59' HOUR TO MINUTE)
  AND END(SYSTEM_TIME) >= CURRENT_TIMESTAMP,
  END(job_dur) < CURRENT_DATE AT INTERVAL -'12:59' HOUR TO MINUTE
  AND END(SYSTEM_TIME) >= CURRENT_TIMESTAMP,
  END(SYSTEM_TIME) < CURRENT_TIMESTAMP
)
WITH SYSTEM VERSIONING;

```

Notes

- The partitioning expression could have used `sys_end` instead of `END(SYSTEM_TIME)` and `job_end` instead of `END(job_dur)`.
- `WITH SYSTEM VERSIONING` must be the last clause in the `CREATE TABLE` statement.

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.

Primary Key and Unique Constraints for ANSI Bitemporal Tables

Like valid-time table definitions, bitemporal table definitions can include primary key and unique constraints that prevent rows from having valid-time periods that overlap.

Syntax

The syntax for adding a primary key or unique constraint to bitemporal tables is the same as the syntax used for valid-time tables, described in [Primary Key and Unique Constraints for ANSI Valid-Time Tables](#).

Usage Notes

- The system-time derived period column cannot be included in the columns that constitute the primary key or unique constraint.
- The constraint applies only to rows that are open in the system-time dimension.

Example: Temporal Primary Key Constraint on an ANSI Bitemporal Table

```
CREATE MULTISET TABLE employee_bitemp_pk (
  eid INTEGER NOT NULL,
  ename VARCHAR(5) NOT NULL,
  deptno INTEGER NOT NULL,
  terms VARCHAR(5),
  job_start DATE NOT NULL,
  job_end DATE NOT NULL,
  PERIOD FOR job_dur(job_start,job_end) AS VALIDTIME,
  sys_start TIMESTAMP(6) WITH TIME ZONE NOT NULL
    GENERATED ALWAYS AS ROW START,
  sys_end TIMESTAMP(6) WITH TIME ZONE NOT NULL
    GENERATED ALWAYS AS ROW END,
  PERIOD FOR SYSTEM_TIME(sys_start,sys_end)
  PRIMARY KEY(deptno,job_dur WITHOUT OVERLAPS)
)
PRIMARY INDEX (eid) WITH SYSTEM VERSIONING;
```

Temporal Referential Constraints for ANSI Bitemporal Tables

Temporal 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, which must include the valid-time column as part of the FK, must exist in the corresponding referenced columns (the primary key (PK)) of the parent table. Consequently, the valid-time value of every row in the child table must exist as a valid-time value in the parent table.

These constraints are not enforced by Vantage so are sometimes referred to as "soft referential integrity." 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.

Temporal referential constraints can be defined for valid-time period columns in temporal tables by specifying the valid-time derived period column names with the special PERIOD keyword.

[Temporal Referential Constraints for ANSI Valid-Time Tables](#)

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Usage Notes

- The system-time derived period column cannot be included in the list of referencing or referenced columns.
- The PK-FK constraint applies only to rows that are open in the system-time dimension.
- Because this is a relationship between valid-time columns, the referencing table and referenced table in a PK-FK temporal referential constraint relationship can be either a valid-time or bitemporal table. The table types do not need to match.

Example: Temporal Referential Constraint on an ANSI Bitemporal Table

```
CREATE MULTISET TABLE hire_contracts(
  h_eid INTEGER NOT NULL,
  h_name VARCHAR(5) NOT NULL,
  h_terms VARCHAR(5),
  contract_start DATE NOT NULL,
  contract_end DATE NOT NULL,
  PERIOD FOR hire_period(contract_start,contract_end) AS VALIDTIME,
  PRIMARY KEY(h_eid, hire_period WITHOUT OVERLAPS)

)
PRIMARY INDEX(h_eid);
```



```

CREATE MULTISET TABLE employee_bitemporal_ri (
  eid INTEGER NOT NULL,
  ename VARCHAR(5) NOT NULL,
  deptno INTEGER NOT NULL,
  terms VARCHAR(5),
  job_start DATE NOT NULL,
  job_end DATE NOT NULL,
  PERIOD FOR job_dur(job_start,job_end) AS VALIDTIME,
  UNIQUE(eid,job_dur WITHOUT OVERLAPS),
  sys_start TIMESTAMP(6) WITH TIME ZONE NOT NULL
    GENERATED ALWAYS AS ROW START,
  sys_end TIMESTAMP(6) WITH TIME ZONE NOT NULL
    GENERATED ALWAYS AS ROW END,
  PERIOD FOR SYSTEM_TIME(sys_start,sys_end),
  FOREIGN KEY(eid,PERIOD job_dur) REFERENCES WITH NO CHECK OPTION
  hire_contracts(h_eid, PERIOD hire_period)
)
PRIMARY INDEX (eid) WITH SYSTEM VERSIONING;

```

Bulk Loading Data into Temporal Tables

Vantage supports three 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.

If the FastLoad script includes a CHECKPOINT specification, restarts during loading can change the system-time values for rows that are inserted after the restart.

In this case, create a nontemporal table, load the data then use ALTER TABLE to add the SYSTEM_TIME derived period column and system-versioning.
- 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 and ALTER TABLE to convert the tables to valid-time and system-versioned system-time 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.

Querying ANSI Bitemporal Tables

Special temporal syntax in the FROM clause of SELECT statements allows you to qualify queries by the time period to which they apply. Rows are only returned if they meet the temporal qualifications. These temporal qualifiers take precedence over other criteria that may be specified in a WHERE clause, which can be used to further restrict the rows returned by Vantage.

FROM Clause (ANSI Bitemporal Table Form)

Bitemporal tables support these qualifiers only for system time, as described in [Querying ANSI System-Time Tables](#).

To qualify rows in bitemporal tables by their valid-time periods, use nontemporal SQL in the WHERE clause condition. The SQL can refer to the individual columns that are the beginning and ending bounds of the valid-time derived period column, or you can apply Period data type functions directly to the valid-time derived period column. For more information on Period data type functions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

FROM Clause Examples (ANSI Bitemporal Table Form)

These example use the following ANSI bitemporal table:

eid	ename	deptno	terms	job_start	job_end	sys_start		sys_end	
1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-12-01	12:11:00.000000-08:00	9999-12-31	23:59:59.999999+00:00
1005	Alice	222	TW10	2004/12/01	9999/12/31	2004-12-01	12:00:00.450000-08:00	9999-12-31	23:59:59.999999+00:00
1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01	00:12:23.120000-08:00	9999-12-31	23:59:59.999999+00:00
1001	Sania	111	TW08	2002/01/01	2006/12/31	2002-01-01	00:00:00.000000-08:00	2002-07-01	12:00:00.350000+00:00
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01	12:00:00.350000-08:00	9999-12-31	23:59:59.999999+00:00
1003	SRK	111	TM02	2004/02/10	2005/02/10	2004-02-10	00:00:00.000000-08:00	2004-12-01	00:12:23.120000+00:00

Example: System-Time AS OF Query on an ANSI Bitemporal Table

The following AS OF query retrieves all table rows that were open at a given point in time.

```
SELECT *
FROM employee_bitemp
FOR SYSTEM_TIME AS OF TIMESTAMP'2003-12-02 00:00:01.000000-08:00';
```

eid	ename	deptno	terms	job_start	job_end	sys_start		sys_end	
1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-12-01	12:11:00.000000-08:00	9999-12-31	23:59:59.999999+00:00
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01	12:00:00.350000-08:00	9999-12-31	23:59:59.999999+00:00

Example: System-Time AS OF Query and Valid-Time Current Query on an ANSI Bitemporal Table

The following query further qualifies the query from the last example by adding a valid-time condition to show only rows with information that is currently in effect (job_end is greater than or equal to the current date).

```
SELECT *
FROM employee_bitemp
FOR SYSTEM_TIME AS OF TIMESTAMP'2003-12-02 00:00:01.000000-08:00'
WHERE job_end >= CURRENT_DATE;
```

eid	ename	deptno	terms	job_start	job_end	sys_start		sys_end	
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01	12:00:00.350000-08:00	9999-12-31	23:59:59.999999+00:00

Alternatively, this query could have been used to return the same result:

```
SELECT *
FROM employee_bitemp
FOR SYSTEM_TIME AS OF TIMESTAMP'2003-12-02 00:00:01.000000-08:00'
WHERE END(job_dur) >= CURRENT_DATE;
```

Example: System-Time BETWEEN Query and Valid-Time Non-Temporal Query on an ANSI Bitemporal Table

The following query selects rows that have been deleted from the table (sys_end is something less than the maximum system timestamp value), but that were valid at the beginning of 2006 (job_end is greater than 2006-01-01).

```
SELECT * FROM employee_bitemp
FOR SYSTEM_TIME BETWEEN TIMESTAMP'1900-01-01 00:00:00.000000+00:00' AND
                        CURRENT_TIMESTAMP
WHERE SYS_END < TIMESTAMP'9999-12-31 23:59:59.999999+00:00' AND
      job_end > DATE'2006-01-01';
```

eid	ename	deptno	terms	job_start	job_end	sys_start	sys_end
1001	Sania	111	TW08	2002/01/01	2006/12/31	2002-01-01 00:00:00.000000-08:00	2002-07-01 12:00:00.350000+00:00

Modifying Rows in ANSI Bitemporal Tables

Because system time and valid time are independent dimensions any change to a bitemporal table affects each dimension independently. The behavior of bitemporal tables can be best understood by considering the consequences of changes to each dimension alone.

In the system-time dimension:

- Any change to an existing row marks the original row as a closed row, with the end of the system-time period timestamped to reflect the time of the change.
 - If the change was a deletion, no new rows are added to the table due to the system time.
 - If the change updated the information in the row, Vantage automatically inserts a new row into the table to reflect the new information, and the original row is marked as closed, with system time ending at the time of the change.

Note:

Because the system-time and valid-time dimensions are independent, valid-time temporal changes to a row are treated the same as non-temporal changes, with respect to system time. Such changes cause the original, unchanged row to be closed in system time and stored in the table as a record of how the table existed before the change.

- If a row is inserted into the table, Vantage timestamps the beginning bound of the system-time period with the time of the insertion, and the ending bound is marked as 9999-12-31 12:59:59:999999+00:00.
- In a bitemporal table as in system-time tables, closed rows do not participate in most SQL operations. because these rows are considered to have been deleted from the table.

For the valid-time dimension, the results of changes depends on the relationship between the PA of the change and the PV of the row:

- If the PA overlaps a portion of the PV, Vantage automatically inserts a new row into the table to reflect the new information. Depending on how the PA and PV overlap, the PV of the new row will start or end at the same valid time as the original row, and the other valid-time bound will be automatically timestamped with the date or timestamp of the modification.
- If the PA lies within the PV of the row, two new rows will be generated so that there are rows in the table to reflect the changed row in addition to the original state of the row both before and after the change.
- If the PA completely overlays the PV of the row, the change is similar to a nontemporal modification.

Note:

Only rows that are still open in system time are subject to DELETE and UPDATE modifications in bitemporal tables.

DELETE (ANSI Bitemporal Table Form)

Deletes one or more rows from ANSI bitemporal tables with the option of deleting the rows for only a portion of their valid-time periods.

Syntax

The syntax used for deleting rows in bitemporal tables is the same as the syntax used for valid-time tables, described in [DELETE \(ANSI Valid-Time Table Form\)](#).

DELETE Examples (ANSI Bitemporal Table Form)

Example: Simple DELETE from an ANSI Bitemporal Table

This example uses the following ANSI bitemporal table:

eid	ename	deptno	terms	job_start	job_end	sys_start				sys_end			
1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-12-01	12:11:00.000000	-08:00	9999-12-31	23:59:59.999999	+00:00		
1005	Alice	222	TW10	2004/12/01	9999/12/31	2004-12-01	12:00:00.450000	-08:00	9999-12-31	23:59:59.999999	+00:00		
1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01	00:12:23.120000	-08:00	9999-12-31	23:59:59.999999	+00:00		
1001	Sania	111	TW08	2002/01/01	2006/12/31	2002-01-01	00:00:00.000000	-08:00	2002-07-01	12:00:00.350000	+00:00		
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01	12:00:00.350000	-08:00	9999-12-31	23:59:59.999999	+00:00		
1003	SRK	111	TM02	2004/02/10	2005/02/10	2004-02-10	00:00:00.000000	-08:00	2004-12-01	00:12:23.120000	+00:00		

A simple SELECT shows the open rows that have not been logically deleted from the table. Only these rows are subject to deletion:

```
SELECT * FROM employee_bitemp;
```

eid	ename	deptno	terms	job_start	job_end	sys_start				sys_end			
1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-12-01	12:11:00.000000	-08:00	9999-12-31	23:59:59.999999	+00:00		
1005	Alice	222	TW10	2004/12/01	9999/12/31	2004-12-01	12:00:00.450000	-08:00	9999-12-31	23:59:59.999999	+00:00		
1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01	00:12:23.120000	-08:00	9999-12-31	23:59:59.999999	+00:00		
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01	12:00:00.350000	-08:00	9999-12-31	23:59:59.999999	+00:00		

A simple DELETE that has no temporal qualifications logically deletes (closes) the row in system time. Using a nontemporal query, the row no longer appears in the table:

```
DELETE FROM employee_bitemp WHERE ename='Ash';
```

```
SELECT * FROM employee_bitemp;
```

eid	ename	deptno	terms	job_start	job_end	sys_start				sys_end			
1005	Alice	222	TW10	2004/12/01	9999/12/31	2004-12-01	12:00:00.450000	-08:00	9999-12-31	23:59:59.999999	+00:00		
1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01	00:12:23.120000	-08:00	9999-12-31	23:59:59.999999	+00:00		
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01	12:00:00.350000	-08:00	9999-12-31	23:59:59.999999	+00:00		

Using a temporal query reveals that the row still exists in the table as a closed row. None of the values in the row have been changed except the ending bound of the system time, which indicates the time the row was deleted:

eid	ename	deptno	terms	job_start	job_end	sys_start		sys_end	
1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-12-01	12:11:00.000000-08:00	2014-02-28	19:40:51.250000-08:00
1005	Alice	222	TW10	2004/12/01	9999/12/31	2004-12-01	12:00:00.450000-08:00	9999-12-31	23:59:59.999999+00:00
1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01	00:12:23.120000-08:00	9999-12-31	23:59:59.999999+00:00
1001	Sania	111	TW08	2002/01/01	2006/12/31	2002-01-01	00:00:00.000000-08:00	2002-07-01	12:00:00.350000+00:00
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01	12:00:00.350000-08:00	9999-12-31	23:59:59.999999+00:00
1003	SRK	111	TM02	2004/02/10	2005/02/10	2004-02-10	00:00:00.000000-08:00	2004-12-01	00:12:23.120000+00:00

Example: Valid-Time DELETE from an ANSI Bitemporal Table

```
DELETE FROM employee_bitemp
FOR PORTION OF job_dur FROM DATE'2009-01-01' TO DATE'2010-01-01'
WHERE ename='Fred';
```

eid	ename	deptno	terms	job_start	job_end	sys_start		sys_end	
1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-12-01	12:11:00.000000-08:00	999-12-31	23:59:59.999999+00:00
1005	Alice	222	TW10	2004/12/01	9999/12/31	2004-12-01	12:00:00.450000-08:00	9999-12-31	23:59:59.999999+00:00
1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01	00:12:23.120000-08:00	9999-12-31	23:59:59.999999+00:00
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01	12:00:00.350000-08:00	9999-12-31	23:59:59.999999+00:00

Because the PA of the DELETE statement was within the PV of the affected row, two rows result, just as for a deletion on a valid-time table:

```
SELECT * FROM employee_bitemp;
```

eid	ename	deptno	terms	job_start	job_end	sys_start		sys_end	
1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-12-01	12:11:00.000000-08:00	9999-12-31	23:59:59.999999+00:00
1005	Alice	222	TW10	2004/12/01	9999/12/31	2004-12-01	12:00:00.450000-08:00	9999-12-31	23:59:59.999999+00:00
1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01	00:12:23.120000-08:00	9999-12-31	23:59:59.999999+00:00
1004	Fred	222	PW12	2001/05/01	2009/01/01	2014-02-28	21:06:33.460000-08:00	9999-12-31	23:59:59.999999+00:00
1004	Fred	222	PW12	2010/01/01	9999/12/31	2014-02-28	21:06:33.460000-08:00	9999-12-31	23:59:59.999999+00:00

In system time these rows are both considered to be new rows added to the table, which is reflected by their sys_start values that show the time that the middle portion of the Fred row valid time was deleted. To see all that occurred in system time to the table as a result of the “partial” deletion, use a temporal query that shows all rows, open and closed:


```
SELECT * FROM employee_bitemp
FOR SYSTEM_TIME BETWEEN TIMESTAMP'1900-01-01 00:00:00.000000+00:00' AND CURRENT_TIMESTAMP;
```

eid	ename	deptno	terms	job_start	job_end	sys_start		sys_end	
1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-12-01	12:11:00.000000-08:00	9999-12-31	23:59:59.999999+00:00
1005	Alice	222	TW10	2004/12/01	9999/12/31	2004-12-01	12:00:00.450000-08:00	9999-12-31	23:59:59.999999+00:00
1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01	00:12:23.120000-08:00	9999-12-31	23:59:59.999999+00:00
1001	Sania	111	TW08	2002/01/01	2006/12/31	2002-01-01	00:00:00.000000-08:00	2002-07-01	12:00:00.350000+00:00
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01	12:00:00.350000-08:00	2014-02-28	21:06:33.460000-08:00
1004	Fred	222	PW12	2001/05/01	2009/01/01	2014-02-28	21:06:33.460000-08:00	9999-12-31	23:59:59.999999+00:00
1004	Fred	222	PW12	2010/01/01	9999/12/31	2014-02-28	21:06:33.460000-08:00	9999-12-31	23:59:59.999999+00:00
1003	SRK	111	TM02	2004/02/10	2005/02/10	2004-02-10	00:00:00.000000-08:00	2004-12-01	00:12:23.120000+00:00

Now you can see there are actually three physical Fred rows in the table. The two new rows that were added to account for Fred's time before and after the deletion, and the original Fred row has been logically deleted from the table, as reflected by the sys_end time that shows the time of the change. The valid-time bounds in the deleted row are those of the original Fred row, before the deletion.

UPDATE (ANSI Bitemporal Table Form)

Modifies one or more existing rows in ANSI bitemporal tables with the option of modifying the rows for only a portion of their valid-time periods.

Syntax

The syntax used for updating rows in ANSI bitemporal tables is the same as the syntax used for valid-time tables, described in [UPDATE \(ANSI Valid-Time Table Form\)](#).

Example: Updating a Row in an ANSI Bitemporal Table

This example uses the following ANSI bitemporal table:

eid	ename	deptno	terms	job_start	job_end	sys_start	sys_end
1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-12-01 12:11:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1005	Alice	222	TW10	2004/12/01	9999/12/31	2004-12-01 12:00:00.450000-08:00	9999-12-31 23:59:59.999999+00:00
1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01 00:12:23.120000-08:00	9999-12-31 23:59:59.999999+00:00
1001	Sania	111	TW08	2002/01/01	2006/12/31	2002-01-01 00:00:00.000000-08:00	2002-07-01 12:00:00.350000+00:00
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01 12:00:00.350000-08:00	9999-12-31 23:59:59.999999+00:00
1003	SRK	111	TM02	2004/02/10	2005/02/10	2004-02-10 00:00:00.000000-08:00	2004-12-01 00:12:23.120000+00:00

Only rows that are still open in system time can be DELETED or UPDATED in an ANSI bitemporal table. To see those rows issue a simple SELECT:

```
SELECT * FROM employee_bitemp;
```

eid	ename	deptno	terms	job_start	job_end	sys_start	sys_end
1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-12-01 12:11:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1005	Alice	222	TW10	2004/12/01	9999/12/31	2004-12-01 12:00:00.450000-08:00	9999-12-31 23:59:59.999999+00:00
1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01 00:12:23.120000-08:00	9999-12-31 23:59:59.999999+00:00
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01 12:00:00.350000-08:00	9999-12-31 23:59:59.999999+00:00

Now update if a row is updated for a portion of the valid time, two rows will result:

```
UPDATE employee_bitemp
FOR PORTION OF job_dur FROM DATE '2005-01-01' TO DATE '9999-12-31'
SET terms='PW11'
WHERE ename='Alice';
```

A simple SELECT shows that where there had been one row for Alice before, now there are two, because the terms of her employment contract changed as of 2005.

eid	ename	deptno	terms	job_start	job_end	sys_start	sys_end
-----	-------	--------	-------	-----------	---------	-----------	---------

1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-12-01	12:11:00.000000-08:00	9999-12-31	23:59:59.999999+00:00
1005	Alice	222	PW11	2005/01/01	9999/12/31	2014-02-26	00:45:48.450000-08:00	9999-12-31	23:59:59.999999+00:00
1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01	00:12:23.120000-08:00	9999-12-31	23:59:59.999999+00:00
1005	Alice	222	TW10	2004/12/01	2005/01/01	2014-02-26	00:45:48.450000-08:00	9999-12-31	23:59:59.999999+00:00
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01	12:00:00.350000-08:00	9999-12-31	23:59:59.999999+00:00

Both rows are still open in system time, because the row for Alice was not explicitly deleted, it was only updated. The row with Alice's information from before the change in terms is considered a history row in valid time. Its valid-time ending bound reflecting the beginning of the PA of the modification, when the old TW10 terms became obsolete. The new Alice row with the new PW11 terms reflects the current terms for Alice. Its valid-time beginning bound reflects the beginning of the PA of the modification, when the new terms became effective.

However, because there was a change to a row, and the table includes a system-time dimension to track every change, the original row for Alice, the row with the original valid-time start and end values, still exists in the table as a closed (logically deleted) row in system time. You can see this if you use a temporal query in system-time to display all open and closed rows in the table:

```
SELECT * FROM employee_bitemp
FOR SYSTEM_TIME BETWEEN TIMESTAMP'1900-01-01 22:14:02.820000-08:00' AND CURRENT_TIMESTAMP;
```

eid	ename	deptno	terms	job_start	job_end	sys_start	sys_end
1002	Ash	333	TA05	2003/01/01	2003/12/31	2003-12-01 12:11:00.000000-08:00	9999-12-31 23:59:59.999999+00:00
1005	Alice	222	PW11	2005/12/01	9999/12/31	2014-02-26 00:45:48.450000-08:00	9999-12-31 23:59:59.999999+00:00
1010	Mike	444	TW07	2015/01/01	2016/12/31	2004-12-01 00:12:23.120000-08:00	9999-12-31 23:59:59.999999+00:00
1005	Alice	222	PW11	2004/12/01	2005/01/31	2014-02-26 00:45:48.450000-08:00	9999-12-31 23:59:59.999999+00:00
1004	Fred	222	PW12	2001/05/01	9999/12/31	2001-05-01 12:00:00.350000-08:00	9999-12-31 23:59:59.999999+00:00
1005	Alice	222	TW10	2004/12/01	9999/12/31	2004-12-01 12:00:00.450000-08:00	2014-02-26 00:45:48.450000-08:00
1003	SRK	111	TM02	2004/02/10	2005/02/10	2004-02-10 00:00:00.000000-08:00	2004-12-01 00:12:23.120000+00:00
1001	Sania	111	TW08	2002/01/01	2006/12/31	2002-01-01 00:00:00.000000-08:00	2002-07-01 12:00:00.350000+00:00

With respect to valid time, the original row was simply modified, and a new row was added to record the new contract terms. The valid-time end of the original row was changed, and the valid-time start of the new row reflects when the new row terms became effective.

With respect to system time, the original row with the original valid-time period was deleted from the table. Because system time and valid time are independent dimensions, a change to the end of the valid-time period in the original row, constitutes a simple row modification that necessitates the original row be closed in system time and logically deleted from the table. So in addition to the two new Alice rows that resulted from the change in the original row in valid time, the original row itself is closed and constitutes a third Alice row, closed in system time, to provide a permanent record of the exact row before the change.

With respect to system time, the beginning and ending bound columns of the valid-time period are just regular data columns, so whenever either of those values changes, the original row as it existed before the change is marked closed in system time but preserved in the table.

Administration

This section provides information on any required administration for temporal table support.

System Clocks

Temporal table support depends on all database 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>.

The database 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.

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, rows are never physically deleted from system-time or bitemporal tables, but persist in the tables indefinitely.

For these reasons, temporal tables grow faster than nontemporal tables, at a rate that depends on how frequently they are modified, and on the nature of those modifications.

The following tables show how temporal tables that have a system-time dimension 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.

Note:

- 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: Lightly Modified Table

	System-time Table	Bitemporal Table
Table Size Before Modifications (rows)	100	100
Modification Type	Current (open rows)	FOR PORTION OF
Modifications per Year (percent of rows)	10% (0.19% weekly)	10% (0.19% weekly)
Number of Additional Rows Produced per Modification	1	1, 2, or 3
Largest Table Size After Modifications (rows)	110	130
Annual Increase in Table Size	10%	30%

Example: Moderately Modified Table

	System-time Table	Bitemporal Table
Table Size Before Modifications (rows)	100	100
Modification Type	Current	FOR PORTION OF
Modifications per Year (percent of rows)	30% (0.58% weekly)	30% (0.58% weekly)
Number of Additional Rows Produced per Modification	1	1, 2, or 3
Largest Table Size After Modifications (rows)	130	190
Annual Increase in Table Size	30%	90%

Example: Heavily Modified Table

	System-time Table	Bitemporal Table
Table Size Before Modifications (rows)	100	100
Modification Type	Current	FOR PORTION OF
Modifications per Year (percent of rows)	50% (0.96% weekly)	50% (0.96% weekly)
Number of Additional Rows Produced per Modification	1	1, 2, or 3
Largest Table Size After Modifications (rows)	150	250
Annual Increase in Table Size	50%	150%

Archiving 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...
Partitioning temporal tables	<ul style="list-style-type: none"> • Row Partitioning ANSI System-Time Tables • Row Partitioning ANSI Valid-Time Tables
ALTER TABLE TO CURRENT	<ul style="list-style-type: none"> • Row Partitioning ANSI System-Time Tables • Row Partitioning ANSI Valid-Time Tables • <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144

How to Read Syntax

This document uses the following syntax conventions.

Syntax Convention	Meaning
KEYWORD	Keyword. Spell exactly as shown. 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. 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. Example: x x x See note after table.
x [, ...]	You can repeat x, separating occurrences with commas. Example: x, x, x See note after table.
x [delimiter...]	You can repeat x, separating occurrences with specified delimiter. Examples: <ul style="list-style-type: none"> If <i>delimiter</i> is semicolon: x; x; x If <i>delimiter</i> is { , OR }, you can do either of the following: <ul style="list-style-type: none"> x, x, x x OR x OR x 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
-

Converting Transaction-Time Tables into ANSI System-Time Tables

Three methods for converting Teradata proprietary transaction-time tables into ANSI compliant system-time tables are described here. For more information on the differences between Teradata Temporal and ANSI Temporal tables, see [ANSI Temporal and Teradata Temporal](#). For more information on Teradata's proprietary non-ANSI-compatible temporal tables and syntax, see *Temporal Table Support*.

Teradata Temporal Tables

Prior to ANSI/ISO developing a standard for temporal tables, Teradata developed proprietary technology and syntax for a similar, but more sophisticated temporal paradigm.

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, three methods to convert transaction-time tables to system-time tables are provided here. 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 in *Teradata Vantage™ - Temporal Table Support*, B035-1182.

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
https://docs.teradata.com/	Search Teradata Documentation, customize content to your needs, and download PDFs. Customers: Log in to access Orange Books.
https://support.teradata.com	One-stop source for Teradata community support, software downloads, and product information. Log in for customer access to: <ul style="list-style-type: none">• Community support• Software updates• Knowledge articles
https://www.teradata.com/University/Overview	Teradata education network
https://support.teradata.com/community	Link to Teradata community