

Teradata Vantage™ - Time Series Tables and Operations

Release 17.10

July 2021

Copyright and Trademarks

Copyright © 2017 - 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 Teradata Time Series Tables and Operations	5
Changes and Additions	5
Chapter 2: Teradata Support for Time Series Data	6
Time Series Data Storage	6
Operations on Time Series Data	6
Standards Compatibility	6
Chapter 3: Time Series PTI Table Creation	7
Differences between PTI and non-PTI Tables	7
CREATE TABLE (Time Series Form)	7
Examples: CREATE TABLE (Time Series Form)	16
CREATE TABLE AS (Time Series Form)	18
Chapter 4: SQL Data Definition Language for PTI Tables	25
SQL DDL Statements and PTI Tables	25
ALTER TABLE	25
CREATE ERROR TABLE	26
CREATE INDEX, Join Indexes, and Hash Indexes	28
DROP TABLE	28
HELP Statements	28
RENAME TABLE	37
SHOW TABLE	37
Statistics Collection Statements	38
Chapter 5: SQL Data Manipulation Language for PTI Tables	39
SQL DML Statements and PTI Tables	39
DELETE	39
INSERT	39
INSERT SELECT	48
MERGE	50
SELECT	55
UPDATE	56
UPDATE (Upsert Form)	56
Chapter 6: Time Series System Functions and Macros	63
TD_GETTIMEBUCKET	63
TD_TIME_BUCKET_NUMBER	64
TD_TIMESERIES_RANGE	67

Chapter 7: Time Series Aggregates and SELECT Extensions	69
Differences between Traditional and Time Series Aggregates	69
Table and Data Definition for Time Series Aggregates Examples	69
GROUP BY TIME Clause	72
USING TIMECODE Clause	78
Examples: GROUP BY TIME and USING TIMECODE Clauses	79
System Virtual Columns Returned by Time Series Aggregates	87
Missing Value Imputation	101
Nested Aggregation	112
Aggregates That Return More Than One Result	116
Chapter 8: Time Series Aggregate Functions	119
EXPLAIN Statement Output for Time Series Aggregate Functions	119
Table and Data Definition for Time Series Aggregates Examples	120
AVERAGE	122
BOTTOM	125
COUNT	128
DELTA_T	129
DESCRIBE	142
FIRST	150
KURTOSIS	151
LAST	154
MAXIMUM	155
Median Absolute Deviation (MAD)	158
MEDIAN	161
MINIMUM	164
MODE	165
PERCENTILE	167
RANK (ANSI)	171
SKEW	174
STDDEV_POP	175
STDDEV_SAMP	176
SUM	178
TOP	179
VAR_POP	182
VAR_SAMP	184
Chapter 9: Tools and Utilities for PTI Tables	186
Appendix A: Notation Conventions	187
Appendix B: Additional Information	190

Introduction to Teradata Time Series Tables and Operations

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™ - Time Series Tables and Operations describes Teradata Vantage support for storing and processing time series data. This document provides information on creating and manipulating Primary Time Index (PTI) tables for storing time series data. It also describes a suite of aggregate functions that you can use to perform time-aware aggregate operations on time series data.

Changes and Additions

Date	Description
July 2021	Minor edits.

Teradata Support for Time Series Data

Time series data is unstructured machine-generated sensor data that is continuously produced and collected by a wide range of applications and devices that make up the Internet of Things. Time series data is typically composed of:

- A unique identifier: One or more columns that identify the data source, for example, BuoyID or Vehicle Identification Number (VIN).
- A timecode: A timestamp or date indicating when the data was collected, for example: '2017-07-07 10:32:12.122200'.
- A sequence number: Data is stored first in the order in which it was collected. If different data arrives at exactly the same time, the sequence number differentiates between the data.
- Measurements: One or more measurements collected at a point in time, for example:
 - Temperature, salinity, wave height, wave speed, and wave direction
 - Engine temperature, oil temperature, oil pressure, tire inflation ratio

Time Series Data Storage

Time series data is stored in a specialized table that contains a structure called a primary time index (PTI). PTI tables are created using the time series forms of CREATE TABLE or CREATE TABLE AS, which have a PRIMARY TIME INDEX clause. PTI tables also contain system-generated columns of time information. Vantage uses the time information to organize the data across the system.

For guidelines on choosing the primary time index (PTI) to result in an even distribution of your type of data, see [Guidelines for Choosing a Primary Time Index](#).

Operations on Time Series Data

Time series aggregate functions perform computations on a set of time series data, allowing users to extract meaning from vast quantities of sensor data. These functions use the GROUP BY TIME clause to present the results in terms of time. Time series functions are either traditional functions that can be used with a GROUP BY TIME clause or functions that operate only on time series data.

For more information on using the GROUP BY TIME clause, see [Time Series Aggregates and SELECT Extensions](#). For more information on time series aggregate functions, see [Time Series Aggregate Functions](#).

Standards Compatibility

The PRIMARY TIME INDEX and GROUP BY TIME clauses are extensions to the Teradata standard.

Time Series PTI Table Creation

Note:

This section covers information relevant to time series tables (tables having a primary time index). Most syntax diagrams show only the portion of syntax that is specifically relevant to time series PTI tables, which is a subset of the full SQL statement syntax. Unless otherwise noted, most of the existing rules and options that apply to conventional non-PTI tables also apply to time series tables. For information on SQL syntax for conventional tables, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144, *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146, and *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

Differences between PTI and non-PTI Tables

PTI tables are optimized to store and retrieve time series data. Time series data typically comes from monitoring sensors that regularly record various data, such as climate or ocean conditions, internal operating status of devices and vehicles, locations of GPS tracking sensors, stock prices, and so forth.

Time series tables are similar to standard, non-PTI Vantage tables, but include a primary time index (PTI) rather than a primary index (PI). Like a PI, a PTI can speed access to frequently queried data, determines how the table row data is distributed among the system AMPs (storage), and determines how the rows are ordered within the table. A properly chosen PTI can help ensure an efficient distribution of data that takes best advantage of the parallelism of Vantage.

Restrictions on PTI Tables

PTI tables support most features and options of standard Vantage tables, with the following exceptions:

- PTI tables cannot be created as standard primary index (PI) or primary AMP index (PA) tables, and cannot be created as NO PRIMARY INDEX (NoPI) tables..
- PTI tables cannot have join or hash indexes.
- PTI tables cannot include row or column partitioning.
- PTI tables cannot also be temporal tables.
- PTI tables cannot also be queue tables.
- PTI tables cannot use load isolation.

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

CREATE TABLE (Time Series Form)

PTI tables are optimized for storing and retrieving time series data, such as sensor data. PTI tables are very similar to standard, non-PTI database tables, except that they include a PRIMARY TIME INDEX clause, rather than a PRIMARY INDEX, PRIMARY AMP INDEX, or NO PRIMARY INDEX clause.

Except as noted, most of the CREATE TABLE options available for non-PTI tables are applicable to PTI tables. For the complete CREATE TABLE syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

CREATE TABLE Syntax (Time Series Form)

```
{ CREATE | CT } [ table_kind ] TABLE table_name
[ , create_table_options ]
( [ generated_columns, ] column_definitions )
PRIMARY TIME INDEX [ index_name ] (
    timecode_data_type
    [ , timezero_date ]
    [ , timebucket_duration ]
    [ , COLUMNS ( column_list ) ]
    [ , { NONSEQUENCED | SEQUENCED [ ( seq_max ) ] } ]
) [ secondary_index_definitions ] [ commit_options ] [ ; ]
```

Syntax Elements

table_kind

PTI tables can be defined as SET or MULTiset tables. By default they are permanent tables, but optionally can be defined as GLOBAL TEMPORARY or VOLATILE tables.

table_name

The name of the table, optionally prefaced by a database or user name.

create_table_options

All standard CREATE TABLE options for non-PTI tables apply to PTI tables. Syntax for these options is described in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

generated_columns

These columns are generated and maintained automatically by the database. They are described in [Automatically Generated Columns in PTI Tables](#).

column_definitions

Standard column definitions for the time series data that will be stored in the PTI table. For more information about standard column definitions in CREATE TABLE statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Note:

The Time Series system automatically generates the first usable column of the table to accommodate the timestamp portion of the time series data. You do not need to explicitly create a column for the timestamp.

index_name

An optional name for the PTI. For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

timecode_data_type

The *timecode_data_type* should reflect the form of the timestamp data in the time series. It can be one of the following data types:

- `TIMESTAMP(n)`, where *n* is the decimal precision of the fractional seconds in the timestamp.
- `TIMESTAMP(n) WITH TIME ZONE`
- `DATE`

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

timezero_date

A date that precedes the earliest date in the time series data.

timezero_date specifies the earliest time series data that the PTI table will accept. The database generates an error if inserted data has a timestamp earlier than *timezero_date*.

timezero_date must be of the format: `DATE 'YYYY-MM-DD'`.

The default value is `DATE '1970-01-01'`.

Although this parameter is optional, best practice is to specify a date near to the earliest timestamp in the time series data.

timebucket_duration

A duration that serves to break up the time continuum in the time series data into discrete groups or buckets. The time bucket value for each row is used, together with the *timezero* date, to help determine the data distribution among the AMPs.

timebucket_duration can be specified using the formal form *time_unit(n)*, where *n* is a positive integer, and *time_unit* can be any of the following: CAL_YEARS, CAL_MONTHS, CAL_DAYS, WEEKS, DAYS, HOURS, MINUTES, SECONDS, MILLISECONDS, or MICROSECONDS.

For example, HOURS(2).

You can also use a shorthand form to represent the *timebucket_duration*, as described in [Shorthand Equivalents for Time Unit Representation](#).

The optimal size of *timebucket_duration* depends on the analysis needs and application. Use a duration that matches the periods to be analyzed. For example, if your application analyzes data from two hour periods, you might specify a *timebucket_duration* as HOURS(2).

column_list

A list of one or more PTI table columns, separated by commas. These columns are used, together with the PTI, to determine how rows are distributed among the AMPs.

NONSEQUENCED SEQUENCED

Specifies whether the time series data readings are unique in time. Depending on the granularity of *timecode_data_type* and the frequency of sensor data collection, there could be multiple readings from a sensor that share the same timestamp.

- NONSEQUENCED assumes that there is only one sensor reading per timestamp. This is the default.

Vantage orders the rows in a nonsequenced PTI table based on the timestamp.

- SEQUENCED means more than one reading from the same sensor may have the same timestamp.

Vantage automatically inserts an extra column named TD_SEQNO in the PTI table to hold a unique sequential number to differentiate readings that have the same timestamp. It is the responsibility of the application collecting and sending the data to Vantage to populate the TD_SEQNO column with sequence numbers.

Vantage orders the rows in a sequenced table based on the timestamp, and within a single timestamp value, orders the rows according to the sequence number.

seq_max

A positive integer from 1 through 2147483647 that represents the maximum TD_SEQNO value. This specifies the maximum number of sensor data rows that can have the same timestamp. The default is 20000. If the value of the TD_SEQNO field exceeds *seq_max*, the database generates an error.

secondary_index_definitions

PTI tables can have USIs and NUSIs. Syntax for defining these indexes is described in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

commit_options

Determine whether to delete or preserve the rows of global temporary or volatile tables when a transaction completes. For more information on these *commit_options*, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Shorthand Equivalents for Time Unit Representation

The following table demonstrates shorthand equivalents that Vantage accepts as time unit duration input.

Time Unit	Formal Form Example	Shorthand Equivalents
Calendar Years	CAL_YEARS(4)	4cy 4cyear 4cyears
Calendar Months	CAL_MONTHS(5)	5cm 5cmonth 5cmonths
Calendar Days 24 hour periods starting at 00:00:00.000000 and ending at 23:59:59.999999 on the day identified by time zero.	CAL_DAYS(6)	6cd 6cd day 6cdays
Weeks	WEEKS(3)	3w 3week 3weeks
Days 24 hour periods starting from time zero.	DAYS(5)	5d 5day 5days
Hours	HOURS(4)	4h 4hr 4hrs 4hour 4hours
Minutes	MINUTES(23)	23m 23mins 23minute 23minutes
Seconds	SECONDS(33)	33s

Time Unit	Formal Form Example	Shorthand Equivalents
		33sec 33secs 33second 33seconds
Milliseconds	MILLISECONDS(12)	12ms 12msec 12msecs 12millisecond 12milliseconds
Microseconds	MICROSECONDS(10)	10us 10usec 10usecs 10microsecond 10microseconds

CREATE TABLE Usage Notes (Time Series Form)

Automatically Generated Columns in PTI Tables

Vantage creates up to three columns automatically in PTI tables, based on the PTI specification in the table definition:

- **TD_TIMECODE** holds the timestamp data from each sensor reading in the time series. This column is functionally the first column in the table. When you insert or load data into a PTI table, the timestamp associated with the data must be the first value inserted for each row. The data type of the column is determined by the *timecode_data_type* specified in the PRIMARY TIME INDEX clause.
- **TD_SEQNO** is generated only if the PTI definition includes the SEQUENCED keyword, indicating that the table is a sequenced table. Sequenced tables are likely to have multiple rows from the same sensor ID have the same timestamp, usually as a result of sensor data readings taken more frequently than the granularity chosen for the *timecode_data_type*. The sequence number in the TD_SEQNO column serves to distinguish different rows that share the same timestamp value. This column must be populated by your data collection application, and is neither populated nor maintained by Vantage. TD_SEQNO is the second column in a sequenced PTI table.
- **TD_TIMEBUCKET** is generated only if the PTI definition includes a *timebucket_duration* value. It is managed by the database, cannot be directly manipulated by SQL code, and does not appear in the output of any query. It is mentioned here only because it appears in the output of SHOW TABLE statements executed on these types of PTI tables.

Although these columns are generated automatically by the database, you can explicitly specify the TD_TIMECODE and TD_SEQNO columns in your PTI table definition if you want to specify a FORMAT or TITLE for the columns. If you want to specify any of these columns, you must explicitly specify *all* of

the automatically generated columns for the table (as appropriate, based on the parameters you add to the PTI). Use the syntax shown below and add your FORMAT and TITLE specifications after the main column definition:

- TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN

Note:

You must specify the TD_TIMEBUCKET column if your PRIMARY TIME INDEX clause includes a *timebucket_duration* parameter. This column is normally hidden and inaccessible to queries. You cannot add FORMAT or TITLE column attributes for the TD_TIMEBUCKET column.

- TD_TIMECODE *timecode_data_type* NOT NULL GENERATED TIMECOLUMN
where *timecode_data_type* must match the *timecode_data_type* used in the PTI definition clause.
- TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN

Guidelines for Choosing a Primary Time Index

The goals in selecting a primary time index are the same as for selecting a regular primary index for non-PTI tables:

- Choose a PTI that will identify each row to result in the most even distribution of time series data among the AMPs in the map used by the table. Use the table below to help determine what to include in your PTI.
- Choose a PTI that includes columns that are frequently queried, to speed data access for those queries.

These goals are discussed in more detail for primary indexes in *Teradata Vantage™ - Database Design*, B035-1094.

All PTIs use the timestamp data from the time series data, and require the *timecode_data_type* in the PTI definition. The following general guidelines can help you determine what else to specify in your PTI definition, based on the general nature of the time series data. In all cases, you can optionally include a *timezero_date* in your PTI definition.

Time Series Characteristics	Include in PTI	Basis for Data Distribution to AMPs
Single continuous long time series Example: A building monitoring system collects sensor data for a single building on a 7/24 continuous basis.	<i>timebucket_duration</i>	Timebucket value
Multiple continuous, long time series Example: Ongoing oceanic and atmospheric monitoring, such as ocean buoy sensor data, where	<i>timebucket_duration</i> and <i>column_list</i>	Timebucket value, and values of

Time Series Characteristics	Include in PTI	Basis for Data Distribution to AMPs
there are thousands of buoys, and data is received from each buoy over a long period of time.	Choose columns likely to be in common queries of the PTI table.	columns included in PTI
Short, static time series Example: Airline flight information, where each flight has an known departure time and arrival time within a short and static time frame. Other examples include scanning data from ultrasound and CAT scan imaging, and electron microscope crystal or cell structure data.	<i>column_list</i> Choose columns likely to be in common queries of the PTI table.	Values of columns included in PTI

NORMALIZE, EXPAND ON, and PTI Tables

NORMALIZE

You can use a NORMALIZE clause in the CREATE TABLE statement for a PTI table just as you would use it for a non-PTI table. The NORMALIZE clause coalesces rows of the table that have values of a Period data type column that meet or overlap, if all other values in the rows are equivalent to each other. For more information about the NORMALIZE clause, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Example: A Normalized PTI table

This example coalesces five inserted rows into two rows.

```
CREATE TABLE flight_sensors (
    FlightID    INTEGER,
    SensorID    INTEGER,
    SensorData  INTEGER,
    duration    PERIOD(TIMESTAMP(6)),
    NORMALIZE ALL BUT(SensorData) ON duration ON MEETS OR OVERLAPS)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2016-10-15', COLUMNS(flightID,
SensorID), NONSEQUENCED);

INSERT INTO flight_sensors (TIMESTAMP '2017-01-06 08:10:00.000000', 67,4,1,
    PERIOD (timestamp '2017-01-06 08:10:00.000000',
        timestamp '2017-01-06 08:10:00.000010'));

INSERT INTO flight_sensors (TIMESTAMP '2017-01-06 08:10:00.000000', 67,4, 1,
    PERIOD (timestamp '2017-01-06 08:10:00.000000',
```

```

        timestamp '2017-01-06 08:10:00.000020')));

INSERT INTO flight_sensors (TIMESTAMP '2017-01-06 08:10:02.000000', 67,5, 99,
    PERIOD (timestamp '2017-01-06 08:10:02.000000',
        timestamp '2017-01-06 08:10:02.000030')));

INSERT INTO flight_sensors (TIMESTAMP '2017-01-06 08:10:02.000000', 67,5, 99,
    PERIOD (timestamp '2017-01-06 08:10:02.000000',
        timestamp '2017-01-06 08:10:02.000040')));

INSERT INTO flight_sensors (TIMESTAMP '2017-01-06 08:10:02.000000', 67,5, 99,
    PERIOD (timestamp '2017-01-06 08:10:02.000000',
        timestamp '2017-01-06 08:10:02.000050')));

SELECT * FROM flight_sensors;

```

TD_TIMECODE	flightID	SensorID	SensorData	duration
2017-01-06 08:10:00.000000	67	4	1	('2017-01-06 08:10:00.000000', '2017-01-06 08:10:00.000020')
2017-01-06 08:10:02.000000	67	5	99	('2017-01-06 08:10:02.000000', '2017-01-06 08:10:02.000050')

EXPAND ON

You can use the EXPAND ON clause of a SELECT statement against PTI tables to create multiple rows in the result set for every table row returned, according to the specification in the EXPAND ON clause.

Example: Using the SELECT EXPAND ON Clause with a Normalized PTI Table

This example uses the table from the NORMALIZE example above to create a new PTI table that has two rows.

The SELECT EXPAND ON query expands two rows of the flight_sensors table into several rows that fill the period associated with each original row in 0.00001 increments.

```

SELECT td_timecode, flightID, SensorData, tsp
FROM flight_sensors
  EXPAND ON duration AS tsp BY INTERVAL '0.00001' SECOND
  FOR PERIOD (timestamp '2017-01-06 08:10:00.000000',
    timestamp '2017-01-06 08:10:02.000050');

```

TD_TIMECODE	FlightID	SensorData	tsp
2017-01-06 08:10:00.000000	67	1	('2017-01-06 08:10:00.000000', '2017-01-06 08:10:00.000010')
2017-01-06 08:10:00.000000	67	1	('2017-01-06 08:10:00.000010', '2017-01-06 08:10:00.000020')
2017-01-06 08:10:02.000000	67	99	('2017-01-06 08:10:02.000000', '2017-01-06 08:10:02.000010')
2017-01-06 08:10:02.000000	67	99	('2017-01-06 08:10:02.000010', '2017-01-06 08:10:02.000020')
2017-01-06 08:10:02.000000	67	99	('2017-01-06 08:10:02.000020', '2017-01-06 08:10:02.000030')
2017-01-06 08:10:02.000000	67	99	('2017-01-06 08:10:02.000030', '2017-01-06 08:10:02.000040')
2017-01-06 08:10:02.000000	67	99	('2017-01-06 08:10:02.000040', '2017-01-06 08:10:02.000050')

Examples: CREATE TABLE (Time Series Form)

Example: PTI Table to Hold Flight Time Series Data

An airline safety department wants to get the overall status of all in-flight planes (but not any specific flight). Consequently, the PTI for this table is defined to include a *timebucket_duration* value. Because neither SEQUENCED nor NONSEQUENCED is included in the PTI definition, the time series data is assumed to be nonsequenced, with each row having a unique timestamp.

```
CREATE TABLE flightinfo(
  flightid integer,
  airspeed integer,
  altitude integer)
PRIMARY TIME INDEX(TIMESTAMP(6), DATE '2016-04-19', MINUTES(1));
```

Example: PTI Table to Hold Ocean Buoy Time Series Data

Data from ocean buoy sensors is an example of a long, continuous time series. Consequently, the PTI for this table is defined to include a *timebucket_duration* and two of the data columns from the table. Because several rows of the table may share the same timestamp, the PTI table is defined as sequenced.

```
CREATE TABLE buoyinfo(
  buoyid integer,
  salinity integer,
  temperature integer)
```



```
PRIMARY TIME INDEX(TIMESTAMP(1), DATE '2016-04-19', HOURS(1), COLUMNS(buoyid,
salinity), SEQUENCED);
```

The output of a SHOW TABLE statement on this table shows the TD_TIMEBUCKET, TD_TIMECODE, and TD_SEQNO columns that are automatically generated by Vantage. In this case, the TD_TIMEBUCKET column is generated because the PTI definition includes a *timebucket_duration* value. The TD_TIMEBUCKET column is hidden and inaccessible to DML. Every PTI table includes a generated TD_TIMECODE column to hold the timestamp data from the time series. The TD_SEQNO column is generated because the PRIMARY TIME INDEX clause includes the SEQUENCED parameter.

```
SHOW TABLE buoyinfo;
```

```
CREATE SET TABLE buoyinfo, NO FALLBACK,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO,
    MAP = TD_MAP1
(
    TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
    TD_TIMECODE TIMESTAMP(1) NOT NULL GENERATED TIMECOLUMN,
    TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN,
    buoyid integer,
    salinity integer,
    temperature integer)
PRIMARY TIME INDEX(TIMESTAMP(1), DATE '2016-04-19', HOURS(1), COLUMNS(buoyid,
salinity), SEQUENCED(20000));
```

Example: PTI Tables Support Features of Non-PTI Tables

PTI tables can be global temporary or volatile tables, and can include secondary indexes and constraints like non-PTI tables. Note that in this example, the shorthand 1hr is used instead of HOURS(1) for the value of *timebucket_duration* in the PTI clause.

```
CREATE SET GLOBAL TEMPORARY TABLE nonseqtab ,FALLBACK,
    CHECKSUM = HIGH,
    FREESPACE = 50,
    DATABLOCKSIZE = 21249,
    BLOCKCOMPRESSION = DEFAULT
(
    buoyid INTEGER NOT NULL,
    salinity INTEGER NOT NULL,
    temperature INTEGER,
    CONSTRAINT pk_1 PRIMARY KEY (buoyid),
```

```

CONSTRAINT ch_1 CHECK (salinity > 50),
CONSTRAINT uq_1 UNIQUE(salinity) )
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2016-04-19', 1hr,
COLUMNS(buoyid, salinity))
UNIQUE INDEX (buoyid)
INDEX (buoyid, salinity) ORDER BY (buoyid)
ON COMMIT DELETE ROWS;

```

CREATE TABLE AS (Time Series Form)

The AS clause of a CREATE TABLE statement allows you to make a complete or partial copy of the definition of an existing table (the *source* table) to a new table (the *target* table), and optionally copy the data from the source table to the target table .

You can use CREATE TABLE AS to create a PTI target table from a PTI or non-PTI source table. You can also create a non-PTI target table from a PTI source table.

CREATE TABLE AS Syntax (Time Series Form)

For a complete description of the CREATE TABLE statement and AS clause, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

```

{ CREATE | CT } [ table_kind ] TABLE target_table_name
[ , create_table_options ]
( [ generated_columns, ] column_definitions )
AS { source_table_name | subquery } WITH [NO] DATA
[ PRIMARY TIME INDEX [ index_name ] (
    timecode_data_type
    [ , timezero_date ]
    [ , timebucket_duration ]
    [ , COLUMNS ( column_list ) ]
    [ , { NONSEQUENCED | SEQUENCED [ ( seq_max ) ] } ]
) [ secondary_index_definitions ] [ commit_options ]
] [ ; ]

```

Syntax Elements

target_table_name

The name of the target table, optionally prefaced by a database or user name, that CREATE TABLE AS creates.

create_table_options

All standard CREATE TABLE options for non-PTI tables apply to PTI tables. Syntax for these options is described in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

generated_columns

These columns are generated automatically by Vantage. They are described in [Automatically Generated Columns in PTI Tables](#).

column_definitions

Standard column definitions for the time series data that will be stored in the PTI table. For more information about standard column definitions in CREATE TABLE statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

source_table_name

The name of the source table, optionally prefaced by a database or user name. The source table can be a PTI or non-PTI table.

subquery

A SELECT statement that explicitly chooses a subset of columns from the source table to be reproduced in the target table. If you use a subquery, you must also use the WITH DATA option.

Note:

In order to create a PTI target table from a non-PTI source table, or create a non-PTI table from a PTI table, you must use a subquery, even if the source and target tables will have the same columns.

For more information about subqueries in CREATE TABLE AS statements, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

WITH [NO] DATA

Determines whether the target table is populated with data from the source table or subquery, or exists as an defined but empty table.

PRIMARY TIME INDEX

- If you specify the PRIMARY TIME INDEX clause, the target table will be a PTI table, regardless of the source table type.
- If you do not specify the PRIMARY TIME INDEX clause, the target table can be either a PTI table or a non-PTI table, depending on the source table type.

All elements of the PRIMARY TIME INDEX clause are the same as those described for [CREATE TABLE \(Time Series Form\)](#), and the consequences to the automatically generated columns are the same.

Usage Notes

- The source table, target table, or both can be PTI tables.
- If you specify a PRIMARY TIME INDEX clause, the automatically generated columns and their characteristics in the target table are determined by the PRIMARY TIME INDEX clause options, regardless of these columns and characteristics in the source PTI table.
- If you create a non-PTI target table from a PTI source table, any automatically generated columns from the source PTI table are created in the target table, however they are no longer managed by Vantage, and their definitions do not include the GENERATED [SYSTEM] TIMECOLUMN attribute from the source PTI table.
- Timestamps in a source PTI table cannot be prior to the *timezero_date* of the PRIMARY TIME INDEX clause.
- Sequence numbers of rows in a source PTI table cannot exceed the *seq_max* of the PRIMARY TIME INDEX clause.

Examples: CREATE TABLE AS (Time Series Form)

Example: Create a PTI table from a Non-PTI table

```

Create table src1(
  TD_TIMECODE TIMESTAMP(6) NOT NULL,
  C1 INTEGER,
  C2 INTEGER);

CREATE TABLE pt71_tgt
AS (SELECT * FROM src1) WITH DATA
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2016-01-01', HOURS(1));

show table pt71_tgt;

CREATE SET TABLE pt71_tgt,
NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT,
DEFAULT MERGEBLOCKRATIO,
MAP = TD_MAP1
(

```

```

    TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
    TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
    C1 INTEGER,
    C2 INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2016-01-01', HOURS(1), NONSEQUENCED);

```

Example: Copy a PTI Table with Data

To copy a PTI table definition and row data to a new PTI table, use a simple CREATE TABLE AS statement.

```
CREATE TABLE pt1_tgt AS pt1_src WITH DATA;
```

The resulting new PTI table, pt1_tgt, is an exact copy of the source table, pt1_src, including the PTI definition and automatically generated columns..

Example: Copy a PTI Table to a New PTI table with a Different PTI Definition

To change the PTI definition in the new PTI table, include a PRIMARY TIME INDEX clause in the CREATE TABLE AS statement.

```

Create table pti_src (
    C1 INTEGER,
    C2 INTEGER NOT NULL,
    C3 INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2016-01-01', HOURS(1),
columns(c1), SEQUENCED);

-- Because the PTI includes a timebucket duration parameter, the table will
include a TD_TIMEBUCKET column,
--   but this column is invisible to SELECT queries and other DML.
-- Because the PTI includes a SEQUENCED parameter, the table will include a
TD_SEQNO column.

SHOW TABLE pti_src;

Create SET TABLE pti_src,
NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT,
DEFAULT MERGEBLOCKRATIO,
MAP = TD_MAP1
(
    TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
    TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
    TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN,

```

```

C1 INTEGER,
C2 INTEGER NOT NULL,
C3 INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2016-01-01', HOURS(1),
columns(c1), SEQUENCED(20000));

-- Assume the table has been populated with data.

CREATE TABLE pti_tgt
AS pti_src WITH DATA
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2015-07-01', HOURS(1), NONSEQUENCED);

SHOW TABLE pti_tgt;

Create SET TABLE pti_tgt,
NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT,
DEFAULT MERGEBLOCKRATIO,
MAP = TD_MAP1
(
TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
TD_SEQNO INTEGER NOT NULL,
C1 INTEGER,
C2 INTEGER NOT NULL,
C3 INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2015-07-01', HOURS(1), NONSEQUENCED);

```

Note the following differences between the original and new PTI tables:

- The source table, `pti_src`, has a PRIMARY TIME INDEX clause that includes the COLUMNS option, so the PTI for the source table distributes the data from `pti_src` to system AMPs based on the values in both the hidden `TD_TIMEBUCKET` column and in the `c1` column. However, the PRIMARY TIME INDEX clause for the new table, `pti_tgt`, does not include a COLUMNS option, so the primary time index will not include any of the data columns from the table. For this new table, the data is distributed to system AMPs based only on values in the `TD_TIMEBUCKET` column.
- The new PTI table is NONSEQUENCED, rather than SEQUENCED. Therefore, the column attribute GENERATED TIMECOLUMN is not part of the `TD_SEQNO` column definition in the new PTI table, and `TD_SEQNO` is treated as a regular column.
- The PTI for the new table has a different *timezero_date* than the source table. The *timezero_date* limits the earliest timestamp date allowed for data stored in the table. Because this CREATE TABLE AS statement includes the WITH DATA clause, all rows from `pti_src` are populated to `pti_tgt`. If any of these

rows have TD_TIMECODE (timestamp) values that are earlier than the *timezero_date* specified for the target table, the database generates an error, and the CREATE TABLE AS statement fails.

Example: Create a non-PTI table from a PTI table

```
SHOW TABLE pti_src;

Create SET TABLE pti_src,
NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT,
DEFAULT MERGEBLOCKRATIO,
MAP = TD_MAP1
(
TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN,
C1 INTEGER,
C2 INTEGER NOT NULL,
C3 INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2016-01-01', HOURS(1),
columns(c1), SEQUENCED);

CREATE TABLE reg1_tgt AS (SELECT * FROM pti_src) WITH DATA;

show table reg1_tgt;

CREATE SET TABLE reg1_tgt ,
    NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO,
    MAP = TD_MAP1
    (
        TD_TIMEBUCKET BIGINT,
        TD_TIMECODE TIMESTAMP(6),
        TD_SEQNO INTEGER,
        C1 INTEGER,
        C2 INTEGER,
        C3 INTEGER)
    PRIMARY INDEX ( TD_TIMEBUCKET );
```

Note that, if otherwise unspecified in the CREATE TABLE AS statement, the primary index for the new non-PTI table uses the first column defined for the table. In this case, the PTI source table had a *timebucket_duration* parameter, HOURS(1), so the first column of the source and target tables is TD_TIMEBUCKET, which is used as the primary index for the new table. If the source PTI table did not have TD_TIMEBUCKET as the first column, TD_TIMECODE would be used as the primary index for the non-PTI table. You can explicitly specify a PRIMARY INDEX clause to override the default behavior.

SQL Data Definition Language for PTI Tables

SQL DDL Statements and PTI Tables

This section describes the SQL DDL statements, in addition to CREATE TABLE, that support PTI tables and how they are used with PTI tables.

ALTER TABLE

You can use the ALTER TABLE statement with time series tables. For complete information about ALTER TABLE syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

You cannot use the following ALTER TABLE options on time series tables:

- ALTER TABLE MODIFY PRIMARY
- ALTER TABLE FROM TIME ZONE
- ALTER TABLE SET or RESET DOWN
- ALTER TABLE TO CURRENT

However, you can use an ALTER TABLE MODIFY PRIMARY statement to add a name or drop an existing name for PRIMARY TIME INDEX.

Example: Alter a PTI Table

The table definition for this example includes an unnamed Primary Time Index.

```
CREATE TABLE ocean_buoy(c1 INTEGER, c2 INTEGER)
PRIMARY TIME INDEX(TIMESTAMP(2) WITH TIME ZONE, DATE '2016-01-03', HOURS(2));
```

You can use an ALTER TABLE statement to add a name for the PRIMARY TIME INDEX.

```
ALTER TABLE ocean_buoy MODIFY PRIMARY TIME INDEX my_pti_index;
```

A SHOW TABLE statement lists the table definition, which now includes a named PRIMARY TIME INDEX.

```
SHOW TABLE ocean_buoy;

CREATE SET TABLE ocean_buoy ,NO FALLBACK ,
      NO BEFORE JOURNAL,
      NO AFTER JOURNAL,
      CHECKSUM = DEFAULT,
```

```

DEFAULT MERGEBLOCKRATIO,
MAP = TD_MAP1
(
    TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
    TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
    TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN,
    c1 INTEGER,
    c2 INTEGER)
PRIMARY TIME INDEX my_pti_index (TIMESTAMP(6), DATE '2015-05-02', HOURS(1),
COLUMNS(c1,c2), NONSEQUENCED);

```

ALTER TABLE Usage Notes

System Generated Time Series Columns and ALTER TABLE

You cannot specify the system generated time series columns, TD_TIMEBUCKET, TD_TIMECODE, or TD_SEQNO in an ALTER TABLE statement, including:

- ALTER TABLE *table_name* ADD *column_name*
- ALTER TABLE *table_name* DROP *column_name*
- ALTER TABLE ADD FOREIGN KEY (*column_name*)
- ALTER TABLE ADD UNIQUE *column_name*
- ALTER TABLE ADD PRIMARY KEY *column_name*
- ALTER TABLE RENAME *column_name*

You cannot add the following column attributes to the system generated time series columns, TD_TIMEBUCKET, TD_TIMECODE, or TD_SEQNO:

- DEFAULT, DEFAULT NULL, DEFAULT DATE, DEFAULT TIME, or DEFAULT USER
- NAMED
- NULL or NOT NULL
- CASESPECIFIC or NOT CASESPECIFIC
- CS or NOT CS
- CHARACTER SET
- COMPRESS or NO COMPRESS
- UPPERCASE or UC

CREATE ERROR TABLE

You can define an error table for a PTI table. For complete information about CREATE ERROR TABLE syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

The system generated time series columns for a PTI table are included in the error table, but without the column attributes specific to time series. The error table columns do not include the NOT NULL GENERATED SYSTEM TIMECOLUMN attributes.

Example: Defining an Error Table for a PTI Table

Following is the table definition for this example.

```
CREATE TABLE ocean_buoy(
    TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
    TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
    TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN,
    c1 INTEGER,
    c2 INTEGER)
PRIMARY TIME INDEX(TIMESTAMP(6), DATE '2015-05-02', HOURS(1),
COLUMNS(c1,c2), SEQUENCED(20000));
```

You can use the statement below to create an error table for PTI table ocean_buoy. By default, the error table name begins with the prefix et_ followed by the name of the base table ocean_buoy.

```
CREATE ERROR TABLE FOR ocean_buoy;
```

You can use a SHOW TABLE statement to display the error table definition. In the error table, the NOT NULL GENERATED TIMECOLUMN attributes specific to time series are not included for the TD_TIMEBUCKET, TD_TIMECODE, and TD_SEQNO columns, in addition to the SYSTEM attribute for the TD_TIMEBUCKET column.

```
CREATE MULTISET TABLE et_ocean_buoy ,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO,
    MAP = TD_MAP1
(
    TD_TIMEBUCKET BIGINT,
    TD_TIMECODE TIMESTAMP(6),
    TD_SEQNO INTEGER,
    c1 INTEGER,
    c2 INTEGER,
    ETC_DBQL_QID DECIMAL(18,0) FORMAT '-(18)9' NOT NULL,
    ETC_DMLType CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC,
    ETC_ErrorCode INTEGER NOT NULL,
    ETC_ErrSeq INTEGER NOT NULL,
```

```

ETC_IndexNumber SMALLINT,
ETC_IdxErrType CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC,
ETC_RowId BYTE(16),
ETC_TableId BYTE(6),
ETC_FieldId SMALLINT,
ETC_RITableId BYTE(6),
ETC_RIFieldId SMALLINT,
ETC_TimeStamp TIMESTAMP(2) NOT NULL,
ETC_Blob BLOB(2033152))
PRIMARY INDEX mytsinx ( TD_TIMEBUCKET ,c1 ,c2 );

```

CREATE INDEX, Join Indexes, and Hash Indexes

You can use the CREATE INDEX statement to create secondary indexes on a PTI table. For complete information about CREATE INDEX syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. However, you cannot create a secondary index on the TD_TIMEBUCKET, TD_TIMECODE, or TD_SEQNO system generated time series columns.

You cannot create join indexes or hash indexes on PTI tables.

DROP TABLE

You can use the DROP TABLE statement to drop time series tables. For complete information about DROP TABLE syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Example: Drop a PTI Table

The statement below drops the table, timeseries_table1.

```
DROP TABLE timeseries_table1;
```

HELP Statements

These topics describe the HELP statement attributes specific to PTI tables. For complete information about HELP statement syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

HELP COLUMN

You can use HELP COLUMN to display information for the TD_TIMEBUCKET, TD_TIMECODE, and TD_SEQNO system generated time series columns.

HELP COLUMN output includes the attribute, Time Series Column Type. For other column types, the Time Series Column Type value is NULL. The table below lists the time series column type values.

Time Series Column	Time Series Column Type
TD_TIMEBUCKET	TB
TD_TIMECODE	TC
TD_SEQNO	TN

Example: HELP COLUMN with a PTI Table Column

This HELP COLUMN statement displays the information for the TD_TIMEBUCKET column of the PTI table, TimeSeriesTab.

```
HELP COLUMN TimeSeriesTab.TD_TIMEBUCKET;
```

The HELP COLUMN output includes the attribute, Time Series Column Type, with a value of TB:

```

Column Name TD_TIMEBUCKET
      Type I8
    Nullable N
      Format -(19)9
    Max Length 8
  Decimal Total Digits ?
  Decimal Fractional Digits ?
      Range Low ?
      Range High ?
    UpperCase N
  Table/View? T
    Indexed? Y
      Unique? N
    Primary? P
      Title ?
  Column Constraint ?
      Char Type ?
    IdCol Type ?
      UDT Name ?
  Temporal Column N
  Current ValidTime Unique ?
  Sequenced ValidTime Unique ?
  NonSequenced ValidTime Unique ?
  Current TransactionTime Unique ?
    Partitioning Column N
  Column Partition Number 0

```

```

Column Partition Format NA
Column Partition AC NA
Security Constraint N
Derived_UDT ?
Derived_UDTFieldID ?
Column Dictionary Name TD_TIMEBUCKET
Column SQL Name TD_TIMEBUCKET
Column Name UEscape ?
Dictionary Title ?
SQL Title ?
Title UEscape ?
UDT Database Dictionary Name ?
UDT Database SQL Name ?
UDT Database Name UEscape ?
UDT Dictionary Name ?
UDT SQL Name ?
UDT Name UEscape ?
Without Overlaps Unique ?
Storage Format ?
SchemaName ?
Inline Length ?
Transform Length ?
Time Series Column Type TB

```

HELP DATABASE

HELP DATABASE output includes the attribute Table Flavor for each table defined in the database to indicate whether or not the table is a PTI table, as shown in the table below.

Table Flavor	Value
PTI table	S
Non-PTI table	NULL

Example: HELP DATABASE with PTI Tables

This statement displays information for the database, TS_Database.

```
HELP DATABASE TS_Database;
```

The output for the PTI table tstab2 includes the Table Flavor attribute with a value of S and table Kind value T to indicate a PTI table.

```

Table/View/Macro name tstab2
      Kind T
      Comment ?
      Protection N
      Creator Name DBC
      Commit Option N
      Transaction Log Y
Table/View/Macro Dictionary Name tstab2
      Table/View/Macro SQL Name tstab2
      Table/View/Macro Name UEscape ?
      Creator Dictionary Name DBC
      Creator SQL Name DBC
      Creator Name UEscape ?
      Table Flavor S

```

HELP INDEX

For PTI tables, HELP INDEX displays the attributes TimeZero and Timebucket from the PRIMARY TIME INDEX clause.

Example: HELP INDEX with a PTI Table

Below is the table definition for the example.

```

CREATE SET TABLE DB1.tstab1, NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT,
DEFAULT MERGEBLOCKRATIO
(
TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN ,
TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN ,
TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN,
c1 INTEGER,
c2 INTEGER,
c3 VARCHAR(10) CHARACTER SET LATIN NOT CASESPECIFIC)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2015-05-02', HOURS(1), COLUMNS(c1,
c2), SEQUENCED) ;

```

This statement displays the index attributes for tstab1:

```
HELP INDEX tstab1;
```

Following is the help output for the primary index of the PTI table tstab1, which includes the TD_TIMEBUCKET, c1, and c2 columns. TimeZero '2015-05-02' indicates that the time series for the table tstab1 started on May 2, 2015 and Timebucket Hours(1) shows that each time series bucket duration is one hour, that is, HOURS(1).

```

Unique? N
Primary//or//Secondary? P
Column Names TD_TIMEBUCKET,c1,c2
Index Id 1
Approximate Count 0
Index Name ?
Ordered//or//Partitioned? P
CDT Index? N
Index Dictionary Name ?
Index SQL Name ?
Index Name UEscape ?
TimeZero '2015-05-02'
Timebucket Hours(1)

```

HELP TABLE

HELP TABLE displays the properties of the system generated TD_TIMEBUCKET, TD_TIMECODE, and TD_SEQNO columns for PTI Tables, in addition to the other columns of the table. HELP TABLE also includes the Time Series Column Type attribute, which can have the values listed in the table below.

Time Series Column	Time Series Column Type
TD_TIMEBUCKET	TB
TD_TIMECODE	TC
TD_SEQNO	TN

Example: HELP PTI Table

Below is the table definition for the example:

```

CREATE SET TABLE tstab1, NO FALLBACK, NO BEFORE JOURNAL, NO AFTER JOURNAL,
CHECKSUM = DEFAULT, DEFAULT MERGEBLOCKRATIO
( TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
  TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
  TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN,
  c1 INTEGER, c2 INTEGER, c3 VARCHAR(10) CHARACTER SET LATIN NOT CASESPECIFIC)

```



```
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2015-05-02',
HOURS(1 ) , COLUMNS( c1 ,c2 ) ,SEQUENCED) ;
```

This statement displays the help information for tstab1:

```
HELP TABLE tstab1;
```

Following is the HELP TABLE output for each of the columns, which includes the Time Series Column Type for each of the system generated columns:

- TB for the TD_TIMEBUCKET column
- TC for the TD_TIMECODE column
- TN for the TD_SEQNO column

The C1, C2, and C3 columns do not include a Time Series Column Type attribute.

```
Column Name TD_TIMEBUCKET
Type I8
Comment ?
Nullable N
Format -(19)9
Title ?
Max Length 8
Decimal Total Digits ?
Decimal Fractional Digits ?
Range Low ?
Range High ?
UpperCase N
Table/View? T
Default value ?
Char Type ?
IdCol Type ?
UDT Name ?
Temporal N
Column Dictionary Name TD_TIMEBUCKET
Column SQL Name TD_TIMEBUCKET
Column Name UEscape ?
Dictionary Title ?
SQL Title ?
Title UEscape ?
UDT Dictionary Name ?
UDT SQL Name ?
UDT Name UEscape ?
Time Series Column Type TB
```

```

Column Name TD_TIMECODE
Type TS
Comment ?
Nullable N
Format YYYY-MM-DDBHH:MI:SS.S(6)
Title ?
Max Length 26
Decimal Total Digits ?
Decimal Fractional Digits 6
Range Low ?
Range High ?
UpperCase N
Table/View? T
Default value ?
Char Type ?
IdCol Type ?
UDT Name ?
Temporal N
Column Dictionary Name TD_TIMECODE
Column SQL Name TD_TIMECODE
Column Name UEscape ?
Dictionary Title ?
SQL Title ?
Title UEscape ?
UDT Dictionary Name ?
UDT SQL Name ?
UDT Name UEscape ?
Time Series Column Type TC

```

```

Column Name TD_SEQNO
Type I8 Comment ?
Nullable N
Format -(19)9
Title ?
Max Length 8
Decimal Total Digits ?
Decimal Fractional Digits ?
Range Low ?
Range High ?
UpperCase N
Table/View? T
Default value ?
Char Type ?

```

```

IdCol Type ?
UDT Name ?
Temporal N
Column Dictionary Name TD_SEQNO
Column SQL Name TD_SEQNO
Column Name UEscape ?
Dictionary Title ?
SQL Title ?
Title UEscape ?
UDT Dictionary Name ?
UDT SQL Name ?
UDT Name UEscape ?
Time Series Column Type TN

```

```

Column Name c1
Type I
Comment ?
Nullable Y
Format -(10)9
Title ?
Max Length 4
Decimal Total Digits ?
Decimal Fractional Digits ?
Range Low ?
Range High ?
UpperCase N
Table/View? T
Default value ?
Char Type ?
IdCol Type ?
UDT Name ?
Temporal N
Column Dictionary Name c1
Column SQL Name c1
Column Name UEscape ?
Dictionary Title ?
SQL Title ?
Title UEscape ?
UDT Dictionary Name ?
UDT SQL Name ?
UDT Name UEscape ?

```

```

Column Name c2
Type I

```

```

Comment ?
Nullable Y
Format -(10)9
Title ?
Max Length 4
Decimal Total Digits ?
Decimal Fractional Digits ?
Range Low ?
Range High ?
UpperCase N
Table/View? T
Default value ?
Char Type ?
IdCol Type ?
UDT Name ?
Temporal N
Column Dictionary Name c2
Column SQL Name c2
Column Name UEscape ?
Dictionary Title ?
SQL Title ?
Title UEscape ?
UDT Dictionary Name ?
UDT SQL Name ?
UDT Name UEscape ?

```

```

Column Name c3
Type CV
Comment ?
Nullable Y
Format X(10)
Title ?
Max Length 10
Decimal Total Digits ?
Decimal Fractional Digits ?
Range Low ?
Range High ?
UpperCase N
Table/View? T
Default value ?
Char Type 1
IdCol Type ?
UDT Name ?
Temporal N

```

```

Column Dictionary Name c3
Column SQL Name c3
Column Name UEscape ?
Dictionary Title ?
SQL Title ?
Title UEscape ?
UDT Dictionary Name ?
UDT SQL Name ?
UDT Name UEscape ?

```

RENAME TABLE

You can use the RENAME TABLE statement to rename PTI tables. For complete information about RENAME TABLE syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

You cannot rename volatile tables that have a Primary Time Index.

You can only rename global temporary table that has a Primary Time Index when there are no materialized instances of that table anywhere in the system.

Example: Rename a PTI Table

The statement below renames the table ts_table to ts_table_new.

```
RENAME TABLE ts_table as ts_table_new;
```

SHOW TABLE

You can use SHOW TABLE to display the table definition for a PTI table. The output includes the system generated TD_TIMEBUCKET, TD_TIMECODE, and TD_SEQNO columns for PTI Tables. For complete information about SHOW TABLE syntax, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

To use the SHOW TABLE statement, you must have at least one privilege on the database containing the table.

Example: Show a PTI Table

Examples: Show a PTI Table

Below is the table definition for an example of a table that uses the timebucket option without sequencing:

```

CREATE TABLE ts_timebucketonly_nonseq (i INT, j INT, k INT)
  PRIMARY TIME INDEX(TIMESTAMP(6), DATE '2016-10-01', HOURS(2));

```

This statement lists the table definition for `ts_timebucketonly_nonseq`:

```
SHOW TABLE ts_timebucketonly_nonseq;
```

The `SHOW TABLE` output includes the two generated columns, `TD_TIMEBUCKET` and `TD_TIMECODE`, in addition to default values for the table creation options that were not specified.

```
CREATE SET TABLE ts_timebucketonly_nonseq, NO FALLBACK,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT,
  DEFAULT MERGEBLOCKRATIO,
  MAP = TD_MAP1
(
  TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
  TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
  i INTEGER,
  j INTEGER,
  k INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2016-10-01', HOURS(2), NONSEQUENCED);
```

Statistics Collection Statements

You can collect statistics on the system generated `TD_TIMECODE`, `TD_SEQNO`, and `TD_TIMEBUCKET` columns of a PTI table.

For complete information about `COLLECT STATISTICS`, `DROP STATISTICS`, `SHOW STATISTICS`, and `HELP STATISTICS`, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

SQL Data Manipulation Language for PTI Tables

SQL DML Statements and PTI Tables

This section lists the SQL DML statements that support PTI tables and describes how to use these statements with PTI tables. For complete information about SQL DML statement syntax, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

DELETE

You cannot specify the TD_TimeBucket column in the WHERE clause of the DELETE statement. For complete information about DELETE syntax, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

INSERT

You use the INSERT statement to insert rows into PTI tables. PTI tables can be either of the following types:

- Nonsequenced
- Sequenced

PTI tables include the system generated TD_TIMECODE column. The TD_TIMECODE value must be the first value in the value list for inserts into nonsequenced and sequenced PTI tables. When inserting into a PTI table, the TD_TIMECODE value cannot precede the timezero value specified in the PTI definition, for example DATE '2016-10-15'.

Sequenced PTI tables also include the TD_SEQNO column. This must be the second value in the value list for inserts into a sequenced PTI table.

In sequenced tables, rows are ordered according the TD_TIMECODE value and then the TD_SEQNO value. Row ordering is automatic, so inserts can occur in any order.

The table below summarizes inserts into the system generated PTI table columns.

Column	Description
TD_TIMECODE	This is a system generated column for sequenced PTI tables. This is a non-nullable column. You must provide a value. This must be the first value in the value list for inserts. You cannot provide CURRENT_TIMESTAMP, CURRENT_TIME, or CURRENT_DATE as a value.
TD_SEQNO	This is a system generated column for sequenced PTI tables. This is a non-nullable column. You must provide a value. This must be the second value in the value list for inserts into a sequenced PTI table.

INSERT Restrictions

Isolated loading is not supported for inserts into PTI tables.

The DEFAULT VALUES option is not supported for inserts into PTI tables.

INSERT Usage Notes

USING Request Modifier with INSERT

You can use the USING Request Modifier with INSERT SQL statements for PTI tables. You can use dot notation naming to import data from an unstructured JSON or Avro data file into a PTI Table.

INSERT Examples

Examples: INSERT into a PTI Table

Table and Data Definition for INSERT Examples

Below is the table definition for the nonsequenced table used in the following examples.

```
CREATE TABLE ocean_buoy_no_seq (buoyid INT, temperature INT)
PRIMARY TIME INDEX(TIMESTAMP(6), HOURS(1), COLUMNS(buoyid), NONSEQUENCED);
```

Following is the table definition displayed by the SHOW TABLE statement.

```
SHOW TABLE ocean_buoy_no_seq;

CREATE SET TABLE my_db.ocean_buoy_no_seq, NO FALLBACK,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT,
  DEFAULT MERGEBLOCKRATIO,
  MAP = TD_MAP1
(
  TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
  TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
  buoyid INTEGER,
  temperature INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '1970-01-01', HOURS(1),
COLUMNS(buoyid), NONSEQUENCED);
```


Below is the table definition for the sequenced table used in the following examples.

```
CREATE TABLE ocean_buoy_seq (buoyid INT, temperature INT)
PRIMARY TIME INDEX(TIMESTAMP(6), DATE '2016-01-01', HOURS(1),
COLUMNS(buoyid), SEQUENCED);
```

Following is the table definition displayed by the SHOW TABLE statement.

```
SHOW TABLE ocean_buoy_seq;

CREATE SET TABLE my_db.ocean_buoy_seq, NO FALLBACK,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO,
    MAP = TD_MAP1
(
    TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
    TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
    TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN,
    buoyid INTEGER,
    temperature INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2016-01-01', HOURS(1),
COLUMNS(buoyid), SEQUENCED(20000));
```

Examples: Inserting Data into a Nonsequenced Table

Examples: INSERT into a Nonsequenced Table

The INSERT values list begins with the TD_TIMECODE column value, then buoyid and temperature. The inserts can be in any order. You do not have to perform the inserts in time sequence order. The system automatically generates a TD_TIMEBUCKET value for each insert row. You do not supply a value for TD_TIMEBUCKET column.

```
INSERT INTO ocean_buoy_no_seq(TIMESTAMP '2017-01-06 10:32:12.123456', 111, 50);
```

```
INSERT INTO ocean_buoy_no_seq(TIMESTAMP '2017-01-06 10:32:12.123456', 111, 60);
```

```
INSERT INTO ocean_buoy_no_seq(TIMESTAMP '2017-01-06 10:44:15.123456', 111, 60);
```

```
INSERT INTO ocean_buoy_no_seq(TIMESTAMP '2017-01-06 10:39:10.123456', 111, 50);
```

```
INSERT INTO ocean_buoy_no_seq(TD_TIMECODE, buoyid, temperature) VALUES
(TIMESTAMP '2020-01-06 10:32:12.123456', 111, 50);
```

```
INSERT INTO ocean_buoy_no_seq(TD_TIMECODE, buoyid, temperature) VALUES
(TIMESTAMP '2020-01-06 10:42:22.123456', 222, 60);
```

```
INSERT INTO ocean_buoy_no_seq(TD_TIMECODE, buoyid, temperature) VALUES
(TIMESTAMP '2020-01-06 10:40:20.123456', 333, 70);
```

Examples: Invalid INSERT into a Nonsequenced Table

You cannot insert into the TD_TIMEBUCKET column. The following statements return an error.

```
INSERT INTO ocean_buoy_no_seq(, TIMESTAMP '2017-01-06 10:32:12.123456',
111, 50);
```

```
INSERT INTO ocean_buoy_no_seq(55, TIMESTAMP '2017-01-06 10:32:12.123456',
111, 50);
```

You cannot use CURRENT_TIMESTAMP as an insert value into the TD_TIMECODE column. The following statement returns an error.

```
INSERT INTO ocean_buoy_no_seq(CURRENT_TIMESTAMP, 111, 50);
```

You must specify a value for the TD_TIMECODE column. The following statement returns an error.

```
INSERT INTO ocean_buoy_no_seq( , 111, 50);
```

Examples: Inserting Data into a Sequenced Table

Examples: INSERT into a Sequenced Table

Following are 5 valid inserts into the table.

```
INSERT INTO ocean_buoy_seq(TIMESTAMP '2017-01-06 10:32:12', 1, 111, 50);
```

```
INSERT INTO ocean_buoy_seq(TIMESTAMP '2017-01-06 10:32:12', 2, 111, 60);
```

```
INSERT INTO ocean_buoy_seq(TIMESTAMP '2017-01-06 10:34:09', 1, 111, 70);
```

```
INSERT INTO ocean_buoy_seq(TIMESTAMP '2017-01-06 10:32:12', 1, 222, 80);
```

```
INSERT INTO ocean_buoy_seq(TIMESTAMP '2017-01-06 10:32:12', 2, 222, 90);
```

Examples: Invalid INSERT into a Sequenced Table

The following statement returns an error because the insert TD_TIMECODE value precedes the timezero DATE '2016-01-01' in the definition of the table ocean_buoy_seq.

```
INSERT INTO ocean_buoy_seq(TIMESTAMP '2015-01-01 10:32:12', 1, 111, 60);
```

This statement returns an error because the TD_SEQNO column value is missing:

```
INSERT INTO ocean_buoy_seq(TIMESTAMP '2017-01-06 10:32:12.133308', , 444, 50);
```

Examples: Legacy INSERT Examples

Examples: Legacy INSERT

Following are valid inserts that use the legacy column to value mapping syntax.

```
INSERT INTO ocean_buoy_no_seq (TD_TIMECODE, BUOYID, TEMPERATURE) VALUES  
(TIMESTAMP '2017-01-06 10:32:12.123456', 111, 50);
```

```
INSERT INTO ocean_buoy_no_seq (BUOYID, TEMPERATURE, TD_TIMECODE) VALUES (111,  
50, TIMESTAMP '2017-01-06 10:32:12.123455');
```

```
INSERT INTO ocean_buoy_no_seq VALUES (TIMESTAMP '2017-01-06 10:39:10.123456',  
111, 66);
```

Examples: Invalid Legacy INSERT

You cannot specify an isolated loading clause with inserts into a PTI table. This statement returns an error.

```
INSERT WITH CONCURRENT ISOLATED LOADING INTO ocean_buoy_seq VALUES(TIMESTAMP  
'2017-01-06 10:32:12.133308', 1, 444, 50);
```

You cannot specify default values for inserts into a PTI table. This statement returns an error.

```
INSERT INTO ocean_buoy_no_seq DEFAULT VALUES;
```

You cannot specify the TD_TIMEBUCKET column in inserts for a PTI table. This statement returns an error.

```
INSERT INTO ocean_buoy_no_seq (TD_TIMEBUCKET, BUOYID, TEMPERATURE, TD_TIMECODE)
VALUES (123, 111, 50, TIMESTAMP '2017-01-06 10:32:12.123456');
```

Examples: USING INSERT with PTI Tables

USING imports data rows from a client system as follows.

Table and Data Definition for USING INSERT Examples

These import files are used in the examples.

JsonBuoy.dat:

```
{"TS":{"Arriving_timestamp" : "2017-01-06 10:32:12.122200", "SeqNo" :
1, "BuoyId" : 777, "Temperature" : 55 }, "Source_country": "USA"}|777
```

AvroBuoy1.dat:

```
7B2274797065223A227265636F7264222C226E616D65223A227265635F30222
C226669656C6473223A5B7B226E616D65223A225453222C2274797065223A7
B2274797065223A227265636F7264222C226E616D65223A227265635F31222
C226669656C6473223A5B7B226E616D65223A224172726976696E675F74696
D657374616D70222C2274797065223A22737472696E67227D2C7B226E616D6
5223A225365714E6F222C2274797065223A22696E74227D2C7B226E616D6522
3A2242756F794964222C2274797065223A22696E74227D2C7B226E616D65223
A2254656D7065726174757265222C2274797065223A22696E74227D5D7D7D2C7
B226E616D65223A22536F757263655F636F756E747279222C2274797065223A22
737472696E67227D5D7D0034323031372D30312D30362031303A33323A31322E
31323232303002920C6E06555341
```

AvroBuoy2.dat:

```
7B2274797065223A227265636F7264222C226E616D65223A227265635F30222
C226669656C6473223A5B7B226E616D65223A225453222C2274797065223A7
B2274797065223A227265636F7264222C226E616D65223A227265635F31222
C226669656C6473223A5B7B226E616D65223A224172726976696E675F74696
D657374616D70222C2274797065223A22737472696E67227D2C7B226E616D6
5223A225365714E6F222C2274797065223A22696E74227D2C7B226E616D65223
A2242756F794964222C2274797065223A22696E74227D2C7B226E616D65223
A2254656D7065726174757265222C2274797065223A22696E74227D5D7D7D2C7
B226E616D65223A22536F757263655F636F756E747279222C2274797065223
A22737472696E67227D5D7D0034323031372D30312D30362031303A33323A31
322E31323232303002920C6E06555341|777
```

RawBuoyNoSeq.dat:

```
2017-01-06 10:32:12.122200|111|55
```

RawBuoySeq.dat:

```
2017-01-06 10:32:12.122200|1|111|55
```

Here is the definition for the table used in this example. The sequenced PTI table `ocean_buoy_seq` includes:

- Time bucket of one hour, `HOURS(1)`.
- Time zero specified as January 1, 2016, `DATE '2016-01-01'`.
- The `buoyid` column in the PRIMARY TIME INDEX, `COLUMNS(buoyid)`.

```
CREATE TABLE ocean_buoy_seq
(
    TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
    TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
    TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN,
    buoyid INTEGER,
    temperature INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2016-01-01', HOURS(1),
COLUMNS(buoyid), SEQUENCED(200));
```

Here is the definition for the table used in this example. The nonsequenced PTI table `ocean_buoy_no_seq` includes:

- Time bucket of one hour, `HOURS(1)`.
- Time zero specified as January 1, 1970, `DATE '1970-01-01'`.
- The `buoyid` column in the PRIMARY TIME INDEX, `COLUMNS(buoyid)`.

```
CREATE SET TABLE ocean_buoy_no_seq
(
    TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
    TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
    buoyid INTEGER,
    temperature INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '1970-01-01', HOURS(1),
COLUMNS(buoyid), NONSEQUENCED);
```

Examples: USING INSERT into a Nonsequenced Table

You import the raw data file using a BTEQ command such as:

```
.import vartext file = RawBuoyNoSeq.dat
```

Here is the USING request modifier for the INSERT statement:

```

USING (timecode varchar(50), buoyid varchar(10), temperature varchar(10))
INSERT INTO ocean_buoy_no_seq(td_timecode, buoyid, temperature)
VALUES (CAST(:timecode as TIMESTAMP(6)), :buoyid, :temperature);

```

You can use dot notation to extract the value for each insert column from a JSON file.

You import the raw data file using a BTEQ command such as:

```
.import vartext file = JsonBuoy.dat
```

This statement includes the USING request modifier to insert the values from the JSON file into the TD_TIMECODE, buoyid, and temperature columns of the ocean_buoy_no_seq table.

```

USING (RAWBUOY VARCHAR(500))
INSERT INTO ocean_buoy_no_seq(TD_TIMECODE, buoyid, temperature)
VALUES (CAST(CAST(:RAWBUOY AS JSON(500)).TS.Arriving_timestamp as TIMESTAMP(6)),
CAST(:RAWBUOY AS JSON(500)).TS.BuoyId,
CAST(:RAWBUOY AS JSON(500)).TS.Temperature);

```

Here is another example of using dot notation to extract the values to insert.

You import the raw data file using a BTEQ command such as:

```
.import vartext file = JsonBuoy.dat
```

This statement includes the USING request modifier to insert the values from the JSON file into the TD_TIMECODE, buoyid, and temperature columns of the table ocean_buoy_no_seq.

```

USING (RAWBUOY VARCHAR(500), buoyid INT)
INSERT INTO ocean_buoy_no_seq(TD_TIMECODE, buoyid, temperature)
VALUES (CAST(CAST(:RAWBUOY AS JSON(500)).TS.Arriving_timestamp
as TIMESTAMP(6)), :buoyid,
CAST(:RAWBUOY AS JSON(500)).TS.temperature);

```

You can use dot notation to extract the value for each insert column from Avro.

You import the Avro data file using a BTEQ command such as:

```
.import vartext file = AvroBuoy1.dat
```

This statement includes the USING request modifier to insert the values from the Avro file into the TD_TIMECODE, buoyid, and temperature columns of the table ocean_buoy_no_seq.

```

USING (RAWBUOY VARCHAR(5000))
INSERT INTO ocean_buoy_no_seq(TD_TIMECODE, buoyid, temperature)
VALUES (CAST(CAST(to_bytes(:RAWBUOY,translate('base16' using UNICODE_TO_LATIN))
as DATASET(500) STORAGE FORMAT AVRO).TS.Arriving_timestamp as TIMESTAMP(6)),
CAST(to_bytes(:RAWBUOY, translate('base16' using UNICODE_TO_LATIN)) as
DATASET(500) STORAGE FORMAT AVRO).TS.BuoyId,

```

```
CAST(to_bytes(:RAWBUOY, translate('base16' using UNICODE_TO_LATIN)) as
DATASET(500) STORAGE FORMAT AVRO).TS.Temperature);
```

Here is another example of using dot notation to extract the values from the Avro file to insert.

You import the Avro data file using a BTEQ command such as:

```
.import vartext file = AvroBuoy2.dat
```

This statement includes the USING request modifier to insert the values from the Avro file into the TD_TIMECODE, buoyid, and temperature columns of the table ocean_buoy_no_seq.

```
USING (RAWBUOY VARCHAR(5000), buoyId VARCHAR(10) )
INSERT INTO ocean_buoy_no_seq(TD_TIMECODE, buoyid, temperature)
VALUES (CAST(CAST(to_bytes(:RAWBUOY,translate('base16' USING UNICODE_TO_LATIN))
AS DATASET(500) STORAGE FORMAT AVRO).TS.Arriving_timestamp AS TIMESTAMP(6)),
:BuoyId,
CAST(to_bytes(:RAWBUOY, translate('base16' USING UNICODE_TO_LATIN)) AS
DATASET(500) STORAGE FORMAT AVRO).TS.Temperature);
```

Examples: USING INSERT into a Sequenced Table

This example imports a raw data file for the insert operation.

You import the raw data file using a BTEQ command such as:

```
.import vartext file = RawBuoySeq.dat;
```

This statement includes the USING request modifier to insert the values from the raw data file into the TD_TIMECODE, TD_SEQNO, buoyid, and temperature columns of the table ocean_buoy_seq.

```
USING (timecode VARCHAR(50), seqno VARCHAR(10), buoyid VARCHAR(10),
temperature VARCHAR(10))
INSERT INTO ocean_buoy_seq(TD_TIMECODE, TD_SEQNO, buoyid, temperature)
VALUES (CAST(:timecode AS TIMESTAMP(6)), :seqno, :buoyid, :temperature);
```

This example imports a JSON data file for the insert operation.

You import the JSON data file using a BTEQ command such as:

```
.import vartext file = JsonBuoy.dat
```

This INSERT statement with a USING request modifier uses dot notation to extract the values for each insert column from the JSON data file.

```
USING (RAWBUOY VARCHAR(500))
INSERT INTO ocean_buoy_seq(TD_TIMECODE, TD_SEQNO, buoyid, temperature)
VALUES (CAST(CAST(:RAWBUOY AS JSON(500)).TS.Arriving_timestamp as TIMESTAMP(6)),
CAST(CAST(:RAWBUOY AS JSON(500)).TS.SeqNo AS INT),
```

```
CAST(:RAWBUOY AS JSON(500)).TS.BuoyId,  
CAST(:RAWBUOY AS JSON(500)).TS.Temperature);
```

This INSERT statement with a USING request modifier uses dot notation to extract the values for each insert column from the Avro data file.

```
USING (RAWBUOY VARCHAR(5000))  
INSERT INTO ocean_buoy_seq(TD_TIMECODE, TD_SEQNO, buoyid, temperature)  
VALUES (CAST(CAST(to_bytes(:RAWBUOY,translate('base16' using UNICODE_TO_LATIN))  
as DATASET(500) STORAGE FORMAT AVRO).TS.Arriving_timestamp as TIMESTAMP(6)),  
CAST(to_bytes(:RAWBUOY,translate('base16' USING UNICODE_TO_LATIN)) as  
DATASET(500) STORAGE FORMAT AVRO).TS.SeqNo,  
CAST(to_bytes(:RAWBUOY,translate('base16' USING UNICODE_TO_LATIN)) as  
DATASET(500) STORAGE FORMAT AVRO).TS.BuoyId,  
CAST(to_bytes(:RAWBUOY,translate('base16' USING UNICODE_TO_LATIN)) as  
DATASET(500) STORAGE FORMAT AVRO).TS.Temperature);
```

INSERT SELECT

You can insert rows into a PTI table using INSERT SELECT syntax. See INSERT and INSERT SELECT in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

You can insert rows by selecting rows from a:

- Non-PTI table and then insert the rows into a PTI table.
- PTI table and then insert the rows into a non-PTI table.
- PTI table and then insert the rows into another PTI table.

You cannot reference the TD_TIMEBUCKET column or insert values into the TD_TIMEBUCKET column using the INSERT SELECT syntax. The TD_TIMEBUCKET column is a system generated column.

When performing an insert-select operation on a PTI table, the rows being inserted cannot have timecode value precedes the timezero specification in the primary time index of the PTI table, for example DATE '2016-01-01'.

Examples: INSERT SELECT and PTI Tables

Table and Data Definition for INSERT SELECT Examples

Below are the PTI table definitions for the INSERT SELECT examples.

```
CREATE TABLE ocean_buoy(buoyid INT,temperature INT)  
PRIMARY TIME INDEX(TIMESTAMP(6), DATE '2012-01-01', HOURS(1), COLUMNS(buoyid));
```

```
CREATE TABLE ocean_buoy_1(buoyid INT,temperature INT)  
PRIMARY TIME INDEX(TIMESTAMP(6), DATE '2012-01-01', HOURS(1), COLUMNS(buoyid));
```



```
CREATE TABLE ocean_buoy_2(buoyid INT,temperature INT)
PRIMARY TIME INDEX(TIMESTAMP(6), DATE '2012-01-01', COLUMNS(buoyid));
```

Following are the non-PTI table definitions for the INSERT SELECT examples.

```
CREATE TABLE normal_table (timecode TIMESTAMP(2), buoyid INT, temperature INT);
```

```
CREATE TABLE normal_table2 (my_timebucket BIGINT NOT NULL, my_timecode
TIMESTAMP(2), buoyid INT, temperature INT);
```

Examples: INSERT SELECT Using PTI and non-PTI Tables

Examples: INSERT SELECT Statements

These statements select data from the non-PTI table `normal_table` and insert the rows into the PTI table `ocean_buoy_1`:

```
INSERT INTO ocean_buoy_1 SELECT * FROM normal_table;
```

```
INSERT INTO ocean_buoy_1(TD_TIMECODE, buoyid, temperature) SELECT *
FROM normal_table;
```

```
INSERT INTO ocean_buoy_1(TD_TIMECODE, buoyid, temperature)
SELECT timecode, buoyid, temperature FROM normal_table;
```

These statements select data from the non-PTI table `normal_table2` and insert the rows into the PTI table `ocean_buoy`:

```
INSERT INTO ocean_buoy SELECT my_timecode, buoyid, temperature
FROM normal_table2;
```

```
INSERT INTO ocean_buoy (TD_TIMECODE, buoyid, temperature) SELECT my_timecode,
buoyid, temperature FROM normal_table2;
```

This is an INSERT SELECT from the PTI table `ocean_buoy` into the PTI table `ocean_buoy_1`:

```
INSERT INTO ocean_buoy_1 SELECT * FROM ocean_buoy;
```

This is an INSERT SELECT from the PTI table `ocean_buoy`, which has a specified time bucket duration, into the PTI table `ocean_buoy_2`, which does not have a specified time bucket duration:

```
INSERT INTO ocean_buoy_2 SELECT * FROM ocean_buoy;
```

This statement selects from the PTI table `ocean_buoy` and inserts the `TD_TIMECODE`, `buoyid`, `temperature` data into the PTI table `ocean_buoy_1`:

```
INSERT INTO ocean_buoy_1(TD_TIMECODE, buoyid, temperature) SELECT *
FROM ocean_buoy;
```

Examples: Invalid INSERT SELECT Statements

You cannot specify the TD_TIMEBUCKET column in an INSERT statement. This statement returns an error:

```
INSERT INTO ocean_buoy_1 (TD_TIMEBUCKET, TD_TIMECODE, buoyid, temperature)
SELECT * FROM ocean_buoy;
```

You cannot specify the TD_TIMEBUCKET column in a SELECT statement subquery. This statement returns an error:

```
INSERT INTO ocean_buoy_1
SELECT TD_TIMEBUCKET, TD_TIMECODE, buoyid, temperature FROM ocean_buoy;
```

You cannot insert values from another column into the TD_TIMEBUCKET column. In the statement below, an attempt to insert values from the my_timebucket column in the table normal_table2 into the TD_TIMEBUCKET column of the table ocean_buoy returns an error:

```
INSERT INTO ocean_buoy SELECT * FROM normal_table2;
```

MERGE

You can use MERGE statements with sequenced and nonsequenced PTI tables. For a complete description of the MERGE statement syntax, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

You cannot use the WITH ISOLATED LOADING clause in MERGE statements with PTI tables.

Usage Notes

Refer to the rules and restrictions below when using MERGE statements with PTI tables.

You cannot specify the system generated TD_TIMEBUCKET column in a MERGE statement, including *column_name*, *column_name_list*, *subquery_source*, *expression*, in a WHERE clause, and so forth. However, you can use the TD_GETTIMEBUCKET function to retrieve TD_TIMEBUCKET column values.

When you merge data into a PTI table, the system automatically generates the TD_TIMEBUCKET value based on the TD_TIMECODE.

When the target table is a PTI table, you must specify the following columns in the ON clause with an equality predicate:

- TD_TIMECODE column.
- TD_SEQNO column, for a sequenced PTI table.
- Each column specified in the COLUMNS clause.

When merging into a PTI table, you cannot merge rows where the source TD_TIMECODE value precedes the time zero value specified in the target table PTI clause, such as DATE '2016-01-03'.

The table below lists the various merge combinations.

Source Table	Target Table	ON Clause
PTI table	PTI table	Must specify an equality predicate for each of the following columns: <ul style="list-style-type: none"> • TD_TIMECODE column • Each column in the COLUMNS specification. • TD_SEQNO column for a sequenced table.
non-PTI table	PTI table	
PTI table	non-PTI table	Must specify an equality predicate for each of the following columns: <ul style="list-style-type: none"> • Each PI column in the primary index clause. • Each PPI column in the PARTITION BY clause.

Examples: MERGE with PTI and non-PTI Tables

Following are examples of merging a PTI table with another PTI table or merging a PTI table with a non-PTI table.

Table and Data Definition for MERGE Examples

Following are the definitions of the tables used in the examples.

Here is the table definition for the sequenced PTI table src_s, which includes a time zero specification of October 15, 2016, DATE '2016-10-15', and time bucket duration of 1 hour, HOURS(1).

```
CREATE TABLE src_s(
    TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
    TD_TIMECODE TIMESTAMP(1) NOT NULL GENERATED TIMECOLUMN,
    TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN,
    c1 INTEGER,
    c2 INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(1), DATE '2016-10-15', HOURS(1),
COLUMNS(C1), SEQUENCED(500));
```

Below is the table definition for the sequenced PTI table tgt_s, which includes a time zero specification of October 15, 2016, DATE '2016-10-15', and time bucket duration of 1 hour, HOURS(1).

```
CREATE TABLE tgt_s(
    TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
    TD_TIMECODE TIMESTAMP(1) NOT NULL GENERATED TIMECOLUMN,
    TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN,
    c1 INTEGER,
```

```

    c2 INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(1), DATE '2016-10-15', HOURS(1),
COLUMNS(C1), SEQUENCED(500));

```

Here is the table definition for the sequenced PTI table `tgt_s2`, which includes a time zero specification of October 15, 2016, DATE '2016-10-15', and time bucket duration of 2 hours, HOURS(2).

```

CREATE TABLE tgt_s2(
    TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
    TD_TIMECODE TIMESTAMP(1) NOT NULL GENERATED TIMECOLUMN,
    TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN,
    c1 INTEGER,
    c2 INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(1), DATE '2016-10-15', HOURS(2),
COLUMNS(C1), SEQUENCED(500));

```

Examples: MERGE Target PTI Time-Bucket Table with Source PTI Time-Bucket Table

This example merges the source sequenced PTI table `src_s` into the target sequenced PTI table `tgt_s`.

```

MERGE INTO
tgt_s AS tgt
USING src_s AS src
ON tgt.TD_TIMECODE = src.TD_TIMECODE AND
   tgt.TD_SEQNO = src.TD_SEQNO AND
   tgt.c1 = src.c1
WHEN MATCHED THEN UPDATE SET c2 = 70
WHEN NOT MATCHED THEN INSERT
(src.TD_TIMECODE, src.TD_SEQNO, src.c1, src.c2);

```

This example uses a subquery to merge the source sequenced PTI table `src_s` into the target sequenced PTI table `tgt_s`.

```

MERGE INTO
tgt_s AS tgt
USING (SELECT TD_TIMECODE, TD_SEQNO, c1, c2 FROM src_s) AS src
ON tgt.TD_TIMECODE = src.TD_TIMECODE AND
   tgt.TD_SEQNO = src.TD_SEQNO AND
   tgt.c1 = src.c1
WHEN MATCHED
THEN UPDATE SET c2 = 70

```

```

WHEN NOT MATCHED THEN
INSERT (src.TD_TIMECODE,src.TD_SEQNO, src.c1, src.c2);

```

This example merges the source sequenced PTI table `src_s` into the target sequenced PTI table `tgt_s`, using `INSERT VALUES` for the `WHEN NOT MATCHED` condition.

```

MERGE INTO
tgt_s AS tgt
USING src_s AS src
ON tgt.TD_TIMECODE = src.TD_TIMECODE AND
   tgt.TD_SEQNO = src.TD_SEQNO AND
   tgt.c1 = src.c1
WHEN MATCHED THEN UPDATE SET c2 = 70
WHEN NOT MATCHED THEN INSERT (TD_TIMECODE, TD_SEQNO, C1, C2)
VALUES (src.TD_TIMECODE, src.TD_SEQNO, src.c1, src.c2);

```

This merge attempt results in an error because the source sequenced PTI table `src_s` time bucket duration of `HOURS(1)` does not match the target sequenced PTI table `tgt_s2` time bucket duration of `HOURS(2)`.

```

MERGE INTO
tgt_s2 AS tgt
USING src_s AS src
ON tgt.TD_TIMECODE = src.TD_TIMECODE AND
   tgt.TD_SEQNO = src.TD_SEQNO
WHEN MATCHED THEN UPDATE SET c2 = 70
WHEN NOT MATCHED THEN INSERT
(src.TD_TIMECODE, src.TD_SEQNO, src.c1, src.c2);

```

Examples: MERGE non-PTI Table with a PTI Time-Bucket Table

Below is the table definition for the nonsequenced PTI table `mrg_tgt_ns`, with a time zero specification of October 15, 2016, `DATE '2016-10-15'`, and a time bucket duration of 1 hour, `HOURS(1)`.

```

CREATE TABLE mrg_tgt_ns(c1 INT, c2 INT)
PRIMARY TIME INDEX(TIMESTAMP(1), DATE '2016-10-15',
HOURS(1));

```

Below is the table definition for the non-PTI table `mrg_src_r`.

```

CREATE TABLE mrg_src_r
(c0 TIMESTAMP(1), c1 INT, c2 INT);

```

This example merges the non-PTI source table `mrg_src_r` into the nonsequenced PTI target table `mrg_tgt_ns`.

```
MERGE INTO mrg_tgt_ns AS tgt
USING mrg_src_r AS src
ON src.c0 = tgt.TD_TIMECODE
WHEN MATCHED THEN UPDATE SET c2 = 70
WHEN NOT MATCHED THEN INSERT (src.c0, src.c1, src.c2);
```

This example merges the non-PTI source table `mrg_src_r` into the nonsequenced PTI target table `mrg_tgt_ns` with an `ON` clause equality condition for the target table `TD_TIMECODE` column.

```
MERGE INTO mrg_tgt_ns AS tgt
USING mrg_src_r AS src
ON tgt.TD_TIMECODE = TIMESTAMP '2016-10-16 10:32:12.2'
WHEN MATCHED THEN UPDATE SET c2 = 72
WHEN NOT MATCHED THEN INSERT (TIMESTAMP '2016-10-16 10:32:12.2',
src.c1, src.c2);
```

Example: MERGE PTI Time-Bucket Table with a PTI No-Time-Bucket Table

Following are the table definitions for the examples.

Below is the definition for the sequenced PTI table `src_s` with a time zero specification of October 15, 2016, DATE '2016-10-15', and time bucket duration of 1 hour, HOURS(1).

```
CREATE TABLE src_s (
    TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
    TD_TIMECODE TIMESTAMP(1) NOT NULL GENERATED TIMECOLUMN,
    TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN,
    c1 INTEGER,
    c2 INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(1), DATE '2016-10-15', HOURS(1), SEQUENCED(500));
```

This definition for the sequenced PTI table `tgt_nb_s` includes a time zero specification of October 15, 2016, DATE '2016-10-15', but does not have a time bucket duration.

```
CREATE TABLE tgt_nb_s (
    TD_TIMECODE TIMESTAMP(1) NOT NULL GENERATED TIMECOLUMN,
    TD_SEQNO INT NOT NULL GENERATED TIMECOLUMN,
    c1 INTEGER,
    c2 INTEGER)
```

```
PRIMARY TIME INDEX (TIMESTAMP(1), DATE '2016-10-15',
COLUMNS(c1), SEQUENCED(500));
```

The definition for this non-PTI table `tgt_r2` includes columns named `timecode` and `seqno`, in addition to a primary index consisting of the column `c1`.

```
CREATE TABLE tgt_r2(
  timecode TIMESTAMP(1),
  seqno INTEGER,
  c1 INTEGER,
  c2 INTEGER)
PRIMARY INDEX ( c1 );
```

In this example, the sequenced PTI table `src_s` with a time bucket is merged into sequenced PTI table `tgt_nb_s`, which does not have a time bucket.

```
MERGE INTO
tgt_nb_s AS tgt
USING (SELECT TD_TIMECODE, TD_SEQNO, c1, c2 FROM src_s)AS src
ON tgt.TD_TIMECODE = src.TD_TIMECODE AND
   tgt.TD_SEQNO = src.TD_SEQNO AND
   tgt.c1 = src.c1
WHEN MATCHED THEN UPDATE SET c2 = 70
WHEN NOT MATCHED THEN
INSERT (src.TD_TIMECODE, src.TD_SEQNO, src.c1, src.c2);
```

Example: MERGE PTI Time-Bucket Table with a non-PTI Table

In this example, sequenced PTI table `src_s`, which does not have a time bucket, is merged into non-PTI table `tgt_r2`.

```
MERGE INTO
tgt_r2 AS tgt
USING (SELECT * FROM src_s) AS src
ON  tgt.c1 = src.c1
WHEN MATCHED THEN UPDATE SET c2 = 70
WHEN NOT MATCHED THEN
INSERT (src.TD_TIMECODE, src.TD_SEQNO, src.c1, src.c2);
```

SELECT

When querying PTI tables, you can use the GROUP BY TIME clause. See [GROUP BY TIME Clause](#). For complete information about SELECT, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

When referencing a PTI table in a SELECT statement:

- Specifying SELECT * from a PTI table defined with a TD_TIMEBUCKET column returns all of the columns in the table except for the system generated TD_TIMEBUCKET column.
- When querying PTI tables, you can specify the system generated TD_TIMECODE and TD_SEQNO columns in the SELECT statement.
- You cannot specify the system generated TD_TIMEBUCKET column in a SELECT statement. However, you can use the TD_GETTIMEBUCKET() function to obtain TD_TIMEBUCKET data. See [TD_GETTIMEBUCKET](#).
- The BETWEEN logical predicate is inclusive of the lower and upper bounds, which can span multiple time bucket boundaries. If you specify GROUP BY TIME (HOURS(1)), each time bucket only includes the lower bound of the one hour time range. A timestamp of 08:00:00 is in one timebucket and a timestamp of 09:00:00 is in the next time bucket. For example, if you specify BETWEEN 08:00:00 AND 09:00:00 and one hour time buckets are defined, that is, HOURS(1), rows from two time buckets can be returned.
- As with any SELECT statement, you must specify an order, otherwise rows are returned unordered. For example, SELECT * FROM *table_name* ORDER BY *column_name*.

UPDATE

You cannot reference the system generated TD_TIMEBUCKET column in an UPDATE statement, including referring to the TD_TIMEBUCKET column in the WHERE clause of an UPDATE statement. For complete information about the UPDATE statement, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

UPDATE (Upsert Form)

You cannot perform an isolated load on a PTI table.

You cannot specify the DEFAULT VALUES option to update a PTI table.

For complete information about UPDATE (Upsert Form), see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

UPDATE (Upsert) Rules and Restrictions

You cannot specify the system generated TD_TIMEBUCKET column in the UPDATE (Upsert Form) statement.

You cannot update the system generated columns TD_TIMEBUCKET, TD_TIMECODE, or TD_SEQNO in an UPDATE (Upsert Form) statement. Also, you cannot update any of the columns defined in the COLUMNS specification of the PRIMARY TIME INDEX.

The WHERE clause must include equality predicate on the following columns to match a single row for updating:

- TD_TIMECODE column
- TD_SEQNO column, for sequenced PTI tables
- All of the columns defined in the COLUMNS specification of the PRIMARY TIME INDEX.

If a match cannot be found, an insert operation is performed.

Examples: UPDATE (Upsert Form) with PTI Tables

Following are examples of update (upsert form) operations on a PTI table, with storage distribution based on:

- Time bucket
- Time bucket and columns clause
- Columns clause

Examples: Upsert with Storage Distribution Based on Time Bucket

Here is the definition for the table used in this example. The nonsequenced PTI table ocean_buoy_no_seq includes:

- Time zero specified as October 15, 2016, DATE '2016-10-15'
- One hour time bucket, HOURS(1)

```
CREATE TABLE ocean_buoy_no_seq (
  TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
  TD_TIMECODE    TIMESTAMP(1) NOT NULL GENERATED TIMECOLUMN,
  salinity       INTEGER,
  temperature    INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(1), DATE '2016-10-15', HOURS(1), NONSEQUENCED);
```

These insert operations populate the table with rows of data:

```
INSERT INTO ocean_buoy_no_seq(TIMESTAMP '2016-10-16 10:33:12.1', 23, 33);
```

```
INSERT INTO ocean_buoy_no_seq(TIMESTAMP '2016-10-16 11:44:12.1', 24, 34);
```

```
INSERT INTO ocean_buoy_no_seq(TIMESTAMP '2016-10-16 12:55:12.1', 25, 35);
```

This statement updates the row where the TD_TIMECODE value matches TIMESTAMP '2016-10-16 10:33:12.1' to set the salinity to 28:

```
UPDATE ocean_buoy_no_seq
SET salinity = 28
WHERE TD_TIMECODE = TIMESTAMP '2016-10-16 10:33:12.1'
ELSE INSERT INTO ocean_buoy_no_seq (TIMESTAMP '2016-10-16 10:33:12.1', 28, 33);
```

This statement inserts a new row after not finding a match in the TD_TIMECODE column for the value, TIMESTAMP '2016-10-16 10:33:12.6':

```
UPDATE ocean_buoy_no_seq
SET salinity = 28
WHERE TD_TIMECODE = TIMESTAMP '2016-10-16 10:33:12.6'
ELSE INSERT INTO ocean_buoy_no_seq (TIMESTAMP '2016-10-16 10:33:12.6', 28, 33);
```

Examples: Upsert with Storage Distribution Based on Time Bucket and Columns Clause

Here is the definition for the table used in this example. The nonsequenced PTI table ocean_buoy_no_seq includes:

- Time zero specified as October 15, 2016, DATE '2016-10-15'
- One hour time bucket, HOURS(1)
- The buoy_id column in the PRIMARY TIME INDEX, COLUMNS(buoy_id)

```
CREATE TABLE ocean_buoy_no_seq (
  TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
  TD_TIMECODE   TIMESTAMP(1) NOT NULL GENERATED TIMECOLUMN,
  BUOY_ID       INTEGER,
  SALINITY      INTEGER,
  TEMPERATURE   INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(1), DATE '2016-10-15', HOURS(1),
COLUMNS(buoy_id), NONSEQUENCED);
```

These insert operations populate the table with rows of data:

```
INSERT INTO ocean_buoy_no_seq(TIMESTAMP '2016-10-16 10:33:12.1', 333, 23, 33);
```

```
INSERT INTO ocean_buoy_no_seq(TIMESTAMP '2016-10-16 11:44:12.1', 444, 24, 34);
```

```
INSERT INTO ocean_buoy_no_seq(TIMESTAMP '2016-10-16 12:55:12.1', 555, 25, 35);
```

This statement updates the row where the TD_TIMECODE value matches TIMESTAMP '2016-10-16 10:33:12.1' for buoy_id 333 to set the salinity to 28:

```
UPDATE ocean_buoy_no_seq
SET salinity = 28
WHERE TD_TIMECODE = TIMESTAMP '2016-10-16 10:33:12.1' AND
```

```

        buoy_id = 333
ELSE INSERT INTO ocean_buoy_no_seq (TIMESTAMP '2016-10-16 10:33:12.1', 333,
28, 33);

```

This statement inserts a new row after not finding a match in the TD_TIMECODE column for the value, TIMESTAMP '2016-10-16 10:33:12.6' for buoy_id 666:

```

UPDATE ocean_buoy_no_seq
SET salinity = 28
WHERE TD_TIMECODE = TIMESTAMP '2016-10-16 10:33:12.6' AND
        buoy_id = 666
ELSE INSERT INTO ocean_buoy_no_seq (TIMESTAMP '2016-10-16 10:33:12.6', 666,
28, 33);

```

Here is the definition for the table used in this example. The sequenced PTI table ocean_buoy_seq includes:

- Time zero specified as October 15, 2016, DATE '2016-10-15'
- One hour time bucket, HOURS(1)
- The buoy_id column in the PRIMARY TIME INDEX, COLUMNS(buoy_id)

```

CREATE TABLE ocean_buoy_seq (
    TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
    TD_TIMECODE    TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
    TD_SEQNO       INTEGER NOT NULL GENERATED TIMECOLUMN,
    buoy_id        INTEGER,
    salinity        INTEGER,
    temperature     INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2016-10-15', HOURS(1),
COLUMNS(buoy_id), SEQUENCED(200));

```

These insert operations populate the table with rows of data:

```

INSERT INTO ocean_buoy_seq(TIMESTAMP '2016-10-16 10:33:12.1', 1, 333, 23, 33);

INSERT INTO ocean_buoy_seq(TIMESTAMP '2016-10-16 11:44:12.1', 1, 444, 24, 34);

INSERT INTO ocean_buoy_seq(TIMESTAMP '2016-10-16 12:55:12.1', 1, 555, 25, 35);

```

This statement updates the row where the TD_TIMECODE value matches TIMESTAMP '2016-10-16 10:33:12.1' and TD_SEQNO =1 for buoy_id 333 to set the salinity to 28:

```

UPDATE ocean_buoy_seq
SET salinity = 28
WHERE TD_TIMECODE = TIMESTAMP '2016-10-16 10:33:12.1' AND
        TD_SEQNO = 1 AND buoy_id = 333

```

```
ELSE INSERT INTO ocean_buoy_seq (TIMESTAMP '2016-10-16 10:33:12.1', 1, 333,
28, 33);
```

This statement inserts a new row after not finding a match in the TD_TIMECODE column for the value, TIMESTAMP '2016-10-16 10:33:12.6' with TD_SEQNO = 1 for buoy_id 666:

```
UPDATE ocean_buoy_seq
SET salinity = 28
WHERE TD_TIMECODE = TIMESTAMP '2016-10-16 10:33:12.6' AND
      TD_SEQNO = 1 AND buoy_id = 666
ELSE INSERT INTO ocean_buoy_seq (TIMESTAMP '2016-10-16 10:33:12.6', 1, 666,
28, 33);
```

Examples: Upsert with Storage Distribution Based on Columns Clause

Here is the definition for the table used in this example. The nonsequenced PTI table ocean_buoy_no_seq includes:

- Time zero specified as October 15, 2016, DATE '2016-10-15'
- The ocean_zone and buoy_id columns in the PRIMARY TIME INDEX, COLUMNS(ocean_zone, buoy_id)

```
CREATE TABLE ocean_buoy_no_seq (
  TD_TIMECODE  TIMESTAMP(1) NOT NULL GENERATED TIMECOLUMN,
  ocean_zone   CHAR(2),
  buoy_id      INTEGER,
  salinity     INTEGER,
  temperature  INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(1), DATE '2016-10-15', COLUMNS(ocean_zone,
buoy_id), NONSEQUENCED);
```

These insert operations populate the table with rows of data:

```
INSERT INTO ocean_buoy_no_seq(TIMESTAMP '2016-10-16 10:33:12.1', 'PC',
333, 23,33);
```

```
INSERT INTO ocean_buoy_no_seq(TIMESTAMP '2016-10-16 11:44:12.1', 'AT',
444, 24,34);
```

```
INSERT INTO ocean_buoy_no_seq(TIMESTAMP '2016-10-16 12:55:12.1', 'IN',
555, 25,35);
```

This statement updates the row where the TD_TIMECODE value matches TIMESTAMP '2016-10-16 10:33:12.1' and ocean_zone = 'PC' for buoy_id 333 to set the salinity to 28:

```
UPDATE ocean_buoy_no_seq
SET salinity = 28
WHERE TD_TIMECODE = TIMESTAMP '2016-10-16 10:33:12.1' AND
      ocean_zone = 'PC' AND buoy_id = 333
ELSE INSERT INTO ocean_buoy_no_seq (TIMESTAMP '2016-10-16 10:33:12.1', 'PC',
333, 23, 33);
```

This statement inserts a new row after not finding a match in the TD_TIMECODE column for the value, TIMESTAMP '2016-10-16 10:33:12.6' and ocean_zone = 'PC' for buoy_id 666:

```
UPDATE ocean_buoy_no_seq
SET salinity = 28
WHERE TD_TIMECODE = TIMESTAMP '2016-10-16 10:33:12.6' AND
      ocean_zone = 'PC' AND buoy_id = 666
ELSE INSERT INTO ocean_buoy_no_seq (TIMESTAMP '2016-10-16 10:33:12.6', 'PC',
666, 28, 33);
```

Here is the definition for the table used in this example. The sequenced PTI table ocean_buoy_seq includes:

- Time zero specified as October 15, 2016, DATE '2016-10-15'
- The ocean_zone and buoy_id columns in the PRIMARY TIME INDEX, COLUMNS(ocean_zone, buoy_id)

```
CREATE TABLE ocean_buoy_seq (
  TD_TIMECODE    TIMESTAMP(1) NOT NULL GENERATED TIMECOLUMN,
  TD_SEQNO       INTEGER NOT NULL GENERATED TIMECOLUMN,
  ocean_zone     CHAR(2),
  buoy_id        INTEGER,
  salinity       INTEGER,
  temperature    INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(1), DATE '2016-10-15', COLUMNS(ocean_zone,
buoy_id), SEQUENCED(20));
```

These insert operations populate the table with rows of data:

```
INSERT INTO ocean_buoy_seq(TIMESTAMP '2016-10-16 10:33:12.1', 1, 'PC',
333, 23,33);
```

```
INSERT INTO ocean_buoy_seq(TIMESTAMP '2016-10-16 11:44:12.1', 1, 'AT',
444, 24,34);
```

```
INSERT INTO ocean_buoy_seq(TIMESTAMP '2016-10-16 12:55:12.1', 1, 'IN',
555, 25,35);
```

This statement updates the row where the TD_TIMECODE value matches TIMESTAMP '2016-10-16 10:33:12.1', TD_SEQNO = 1, and ocean_zone = 'PC' for buoy_id 333 to set the salinity to 28:

```
UPDATE ocean_buoy_seq
SET salinity = 28
WHERE TD_TIMECODE = TIMESTAMP '2016-10-16 10:33:12.1'
AND TD_SEQNO = 1 AND ocean_zone = 'PC' AND buoy_id = 333
ELSE INSERT INTO ocean_buoy_seq (TIMESTAMP '2016-10-16 10:33:12.1', 1, 'PC',
333, 23, 33);
```

This statement inserts a new row after not finding a match in the TD_TIMECODE column for the value, TIMESTAMP '2016-10-16 10:33:12.6', TD_SEQNO = 1, and ocean_zone = 'PC' for buoy_id 666:

```
UPDATE ocean_buoy_seq
SET salinity = 28
WHERE TD_TIMECODE = TIMESTAMP '2016-10-16 10:33:12.6'
AND TD_SEQNO = 1 AND ocean_zone = 'PC' AND buoy_id = 666
ELSE INSERT INTO ocean_buoy_seq (TIMESTAMP '2016-10-16 10:33:12.6', 1, 'PC',
666, 28, 33);
```

Time Series System Functions and Macros

Time series system functions and macros provide information about the content of PTI tables.

TD_GETTIMEBUCKET

The TD_GETTIMEBUCKET system function retrieves the TD_TIMEBUCKET column value from a PTI table. Because a timebucket is a hash key, a DBA can use the TD_TIMEBUCKET column value to determine how well the rows of a PTI table are being distributed across AMPs, avoiding skew.

The return type of this function is BIGINT.

TD_GETTIMEBUCKET Syntax

```
[TD_SYSFNLIB.] TD_GETTIMEBUCKET ( [ [ table_name. ] TD_TIMECODE ] )
```

Syntax Elements

TD_SYSFNLIB

The name of the database where the function is located.

table_name

The name of the PTI table.

TD_TIMECODE

The PTI table column indicating when the data was collected.

Usage Notes

The TD_TIMEBUCKET column cannot be directly referenced in an SQL request, so you can use this function to see the TD_TIMEBUCKET value. This function can be referenced in any SQL request where a system function call is valid.

If an SQL request references more than one PTI table, use *table_name*.TD_TIMECODE to specify the table from which the TD_TIMEBUCKET is being retrieved.

Example: Get a TD_TIMEBUCKET Value from a TD_TIMEBUCKET Column

```
create table ocean_buoys(buoyid integer, salinity integer, temperature integer)
PRIMARY TIME INDEX(TIMESTAMP(6), DATE '2016-04-19', HOURS(1));
```

```
-- The following two adjacent SELECT statements should return the same output:
SELECT td_gettimebucket(td_timecode) from ocean_buoys;
SELECT td_gettimebucket(ocean_buoys.td_timecode) from ocean_buoys;
```

```
TD_GETTIMEBUCKET(TD_TIMECODE)
-----
289
```

```
-- The column title contains TIMEBUCKET_COL.
SELECT td_gettimebucket(ocean_buoys.td_timecode) as TimeBucket_COL
from ocean_buoys;
```

```
TIMEBUCKET_COL
-----
289
```

```
SELECT buoyid FROM ocean_buoys WHERE td_gettimebucket(td_timecode) > 200;
```

```
BUOYID
-----
101
```

TD_TIME_BUCKET_NUMBER

The TD_TIME_BUCKET_NUMBER system function calculates the time bucket number. You can use this function with the HASHROW, HASHBUCKET, or HASHAMP functions to see how time series rows are distributed across the system. Other uses include:

- Determining which candidate rows are assigned to which bucket in a GROUP BY TIME operation
- Using the result in a join or to insert into a table if you assign bucket numbers to rows outside of a GROUP BY TIME operation

The return type of this function is BIGINT.

Usage Notes

You can reference this function in any SQL request where a system function call is valid.

TD_TIME_BUCKET_NUMBER Syntax

```
[TD_SYSFNLIB.] TD_TIME_BUCKET_NUMBER ( timezero, timecode, timebucket_duration )
```

Syntax Elements

TD_SYSFNLIB

The name of the database where the function is located.

timezero

Any expression evaluating to a DATE or TIMESTAMP that is used as time zero for this function. A time zone for the timestamp is optional.

timecode

Any expression evaluating to a DATE or TIMESTAMP that is used as the timecode for this function. A time zone for the timestamp is optional.

timebucket_duration

{

CAL_YEARS |

CAL_MONTHS |

CAL_DAYS |

WEEKS |

DAYS |

HOURS |

MINUTES |

SECONDS |

MILLISECONDS |

MICROSECONDS

}

(

pos_int

)

A time duration that can be specified using any of the units of time shown in the diagram. Abbreviations are allowed for the duration:

Time Unit	Formal Form Example	Shorthand Equivalents
Calendar Years	CAL_YEARS(4)	4cy 4cyear 4cyears
Calendar Months	CAL_MONTHS(5)	5cm 5cmonth 5cmonths

Time Unit	Formal Form Example	Shorthand Equivalents
Calendar Days 24 hour periods starting at 00:00:00.000000 and ending at 23:59:59.999999 on the day identified by time zero.	CAL_DAYS(6)	6cd 6cday 6cdays
Weeks	WEEKS(3)	3w 3week 3weeks
Days 24 hour periods starting from time zero.	DAYS(5)	5d 5day 5days
Hours	HOURS(4)	4h 4hr 4hrs 4hour 4hours
Minutes	MINUTES(23)	23m 23mins 23minute 23minutes
Seconds	SECONDS(33)	33s 33sec 33secs 33second 33seconds
Milliseconds	MILLISECONDS(12)	12ms 12msec 12msecs 12millisecond 12milliseconds
Microseconds	MICROSECONDS(10)	10us 10usec 10usecs 10microsecond 10microseconds

pos_int

A 16-bit positive integer with a maximum value of 32,767.

Example: Use TD_TIME_BUCKET_NUMBER to Calculate a Time Bucket Number

```
SELECT
  TD_TIMECODE,
  TD_TIME_BUCKET_NUMBER(DATE '1900-01-01', TD_TIMECODE, CAL_YEARS(10)) FROM
  ts_group_by_time_tb
WHERE id = 0
ORDER BY 2;

TD_TIMECODE
TD_TIME_BUCKET_NUMBER(1900-01-01,TD_TIMECODE,CAL_YEARS(10))
2006-06-06 06:06:06.006002      11
```

TD_TIMESERIES_RANGE

The TD_TIMESERIES_RANGE macro finds the valid ranges of the TD_TIMECODE and TD_SEQNO columns in a PTI table.

Usage Notes

Use this macro to show the upper limit of values that can be stored in the TD_TIMECODE and TD_SEQNO columns of a PTI table. Sensor data can flow rapidly into a system and cause a PTI overflow. This macro provides available space information for planning purposes and can also be helpful in resolving out of range errors.

TD_TIMESERIES_RANGE Syntax

```
[DBC.] TD_TIMESERIES_RANGE (DBname.TSTable') [;]
```

Syntax Elements

DBC

The name of the database where the function is located.

DBname.TSTable

The name of the database and PTI table.

Example: Get the Valid Range of Values for the TD_TIMECODE and TD_SEQNO Columns

```

Create table MyDB.ocean_buoy (
  TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
  TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
  TD_SEQNO INTEGER NOT NULL GENERATED TIMECOLUMN,
  BuoyID INTEGER,
  Temperature INTEGER NOT NULL)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2016-01-01', HOURS(1), SEQUENCED);

exec DBC.TD_TIMESERIES_RANGE('MyDB.ocean_buoy');

TD_TIMECODE_RANGE
-----

TD_TIMECODE  BETWEEN TIMESTAMP '2016-01-0100:00:00.000000+00:00' AND
TIMESTAMP '2030-08-1214:23:21.842737+00:00'

TD_SEQNO_RANGE
-----

TD_SEQNO  BETWEEN 1  AND 20000

-- you can specify the table name in the macro and will get the same output as
above example.

exec DBC.TD_TIMESERIES_RANGE('ocean_buoy');
```

Time Series Aggregates and SELECT Extensions

Differences between Traditional and Time Series Aggregates

Time series aggregate functions perform computations on a set of time series data and interpret the results in terms of time. Conceptually, traditional functions that are time series enabled behave the same when they are operating on time series data and other data; for example, the times series AVERAGE function and a traditional AVERAGE function compute the average for each specified group. However, only time series aggregate functions have these characteristics:

- The specification of the groups: Time series aggregates are always grouped by time, and optionally by other columns (see [GROUP BY TIME Clause](#)). Traditional aggregates are not grouped by time.
- The resulting columns: Time series aggregates return system virtual columns which may or may not be selected by the user and included in further calculations or results. Traditional aggregates do not return any system virtual columns. See [System Virtual Columns Returned by Time Series Aggregates](#).
- The composition of the result set: Time series aggregates are aware of expected groups in the desired time range. If there is no aggregate result for one or more of these expected groups, time series aggregates allow a user to specify how to handle empty result sets. Traditional aggregates are not aware of expected groups and therefore do not provide any mechanism for handling empty results. For more information, see [Missing Value Imputation](#).

Restrictions

Time series functions cannot be combined with traditional aggregate, ordered analytic, or statistics functions:

- If a GROUP BY TIME clause is present, all aggregates are interpreted as time series aggregates. The presence of any functions not included in the time series aggregate group results in an error.
- If a GROUP BY clause is present, all functions are interpreted as traditional aggregate, ordered analytic, or statistics functions. The presence of any aggregates only valid for time series data results in an error.

Table and Data Definition for Time Series Aggregates Examples

The following SQL creates sample tables with data for use in time series examples.

```
/*PTI Table*/
CREATE TABLE OCEAN_BUOYS(BUOYID INTEGER, SALINITY INTEGER, TEMPERATURE INTEGER )

PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2012-01-01', HOURS(1),
COLUMNS(BUOYID), nonsequenced);
```

```

INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 08:00:00.000000', 0, 55, 10);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 08:08:59.999999', 0, 55, );
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 08:09:59.999999', 0, 55, 99);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 08:10:00.000000', 0, 55, 10);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 08:10:00.000000', 0, 55, 100);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 09:01:25.122200', 1, 55, 77);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 09:01:25.122200', 1, 55, 70);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 09:02:25.122200', 1, 55, 78);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 09:02:25.122200', 1, 55, 71);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 09:03:25.122200', 1, 55, 79);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 09:03:25.122200', 1, 55, 72);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 21:01:25.122200', 2, 55, 80);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 21:02:25.122200', 2, 55, 81);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 21:03:25.122200', 2, 55, 82);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:00:24.000000', 44, 55, 43);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:00:24.333300', 44, 55, 43);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:00:25.122200', 44, 55, 43);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:00:26.122200', 44, 55, 43);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:01:25.122200', 44, 55, 53);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:01:25.122200', 44, 55, 54);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:02:25.122200', 44, 55, 53);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:02:25.122200', 44, 55, 55);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:03:25.122200', 44, 55, 53);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:03:25.122200', 44, 55, 56);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:12:00.000000', 44, 55, 43);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:32:12.122200', 44, 55, 43);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:52:00.000009', 44, 55, 43);

```

/* Sequenced PTI table */

```

CREATE TABLE OCEAN_BUOYS_SEQ(BUOYID INTEGER, SALINITY INTEGER, TEMPERATURE INTEGER,
DATES DATE)

```

```

PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2012-01-01', HOURS(1),
COLUMNS(BUOYID), sequenced);

```

```

INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:32:12.122200',1, 44, 55,
43, '2014-01-01');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:32:12.122200',1, 22, 25,
23, '2014-01-01');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:52:00.000009',2, 44, 55,
43, '2014-02-02');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:12:00.000000',3, 44, 55,
43, '2014-03-03');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:00:24.000000',4, 44, 55,
43, '2014-04-04');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:00:24.333300',5, 44, 55,
43, '2014-05-05');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:00:25.122200',6, 44, 55,
43, '2014-06-06');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:00:26.122200',7, 44, 55,
43, '2014-07-07');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:01:25.122200',8, 44, 55,
53, '2014-08-08');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:02:25.122200',9, 44, 55,
53, '2014-09-09');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:03:25.122200',10, 44,
55, 53, '2014-10-10');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 09:01:25.122200',11, 1, 55,
70, '2014-11-11');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 09:02:25.122200',12, 1, 55,
71, '2014-12-12');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 09:03:25.122200',13, 1, 55,
72, '2015-01-13');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 21:01:25.122200',14, 2, 55,
80, '2015-02-14');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 21:02:25.122200',15, 2, 55,

```

```

81, '2015-03-15');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 21:03:25.122200',16, 2, 55,
82, '2015-04-16');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 08:09:59.999999',17, 0, 55,
99, '2015-05-17');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 08:08:59.999999',18, 0,
55, , '2015-06-18');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 08:10:00.000000',19, 0, 55,
10, '2015-07-19');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:01:25.122200',20, 44,
55, 54, '2015-08-20');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:02:25.122200',21, 44,
55, 55, '2015-09-21');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:03:25.122200',22, 44,
55, 56, '2015-10-22');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 09:01:25.122200',23, 1, 55,
77, '2015-11-23');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 09:02:25.122200',24, 1, 55,
78, '2015-12-24');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 09:03:25.122200',25, 1, 55,
79, '2016-01-25');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 08:00:00.000000',26, 0, 55,
10, '2016-02-26');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 08:10:00.000000',27, 0, 55,
100, '2016-03-27');

```

```
/*Non-PTI Table*/
```

```
CREATE TABLE OCEAN_BUOYS_NONPTI(TIMECODE TIMESTAMP(6), BUOYID INTEGER, SALINITY
INTEGER, TEMPERATURE INTEGER)
```

```
PRIMARY INDEX (BUOYID);
```

```

INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 08:00:00.000000', 0,
55, 10);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 08:08:59.999999',
0, 55, );
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 08:09:59.999999', 0,
55, 99);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 08:10:00.000000', 0,
55, 10);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 08:10:00.000000', 0,
55, 100);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 09:01:25.122200', 1,
55, 77);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 09:01:25.122200', 1,
55, 70);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 09:02:25.122200', 1,
55, 78);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 09:02:25.122200', 1,
55, 71);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 09:03:25.122200', 1,
55, 79);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 09:03:25.122200', 1,
55, 72);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 21:01:25.122200', 2,
55, 80);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 21:02:25.122200', 2,
55, 81);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 21:03:25.122200', 2,
55, 82);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:00:24.000000', 44,
55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:00:24.333300', 44,
55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:00:25.122200', 44,
55, 43);

```

```

INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:00:26.122200', 44,
55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:01:25.122200', 44,
55, 53);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:01:25.122200', 44,
55, 54);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:02:25.122200', 44,
55, 53);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:02:25.122200', 44,
55, 55);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:03:25.122200', 44,
55, 53);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:03:25.122200', 44,
55, 56);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:12:00.000000', 44,
55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:32:12.122200', 44,
55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:52:00.000009', 44,
55, 43);

```

GROUP BY TIME Clause

The GROUP BY TIME clause is an enhancement to the SELECT statement that allows a set of aggregate functions to be computed on data grouped in terms of time. Although the grouping is optimized for PTI tables, it is also supported on non-PTI tables because a timecode column is explicitly specified in the USING TIMECODE clause.

Note:

The syntax diagram and descriptions discuss only the portions of the SELECT syntax that are specific to the GROUP BY TIME clause. For information about the other syntax that you can use with SELECT, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

GROUP BY TIME Clause Syntax

```

{ GROUP BY options |

  GROUP BY TIME ( { timebucket_duration | * }
    [ AND value_expression [,...] ] )
    [ USING ( timestamp_date_col [, seqno_col ] ) ]
    [ FILL ( { NULLS | numeric_constant | PREVIOUS | NEXT } ) ]
}

```

Syntax Elements

options

The optional clauses for GROUP BY TIME are: USING TIMECODE and FILL.

timebucket_duration

```

{ CAL_YEARS |
  CAL_MONTHS |
  CAL_DAYS |
  WEEKS |
  DAYS |
  HOURS |
  MINUTES |
  SECONDS |
  MILLISECONDS |
  MICROSECONDS
} ( pos_int )

```

A time duration that can be specified using any of the units of time shown in the diagram. Abbreviations are allowed for the duration:

Time Unit	Formal Form Example	Shorthand Equivalents
Calendar Years	CAL_YEARS(4)	4cy 4cyear 4cyears
Calendar Months	CAL_MONTHS(5)	5cm 5cmmonth 5cmonths
Calendar Days 24 hour periods starting at 00:00:00.000000 and ending at 23:59:59.999999 on the day identified by time zero.	CAL_DAYS(6)	6cd 6cd day 6cdays
Weeks	WEEKS(3)	3w 3week 3weeks
Days 24 hour periods starting from time zero.	DAYS(5)	5d 5day 5days
Hours	HOURS(4)	4h 4hr 4hrs 4hour 4hours
Minutes	MINUTES(23)	23m 23mins 23minute

Time Unit	Formal Form Example	Shorthand Equivalents
		23minutes
Seconds	SECONDS(33)	33s 33sec 33secs 33second 33seconds
Milliseconds	MILLISECONDS(12)	12ms 12msec 12msecs 12millisecond 12milliseconds
Microseconds	MICROSECONDS(10)	10us 10usec 10usecs 10microsecond 10microseconds

The time unit representation (CAL_YEARS, CAL_MONTHS, and so on) is used to specify the *timebucket_duration* for both the CREATE TABLE (Time Series Form) and the GROUP BY TIME clause.

When the time unit is used within a GROUP BY TIME clause, it defines the duration of each timebucket for aggregation and is used to assign each potential timebucket a unique number. You can access the timebucket number in the \$TD_GROUP_BY_TIME virtual column. For more information, see [\\$TD_GROUP_BY_TIME](#).

The time units do not store values such as the year or the month. For example, CAL_YEARS(2017) does not set the year to 2017. It sets the *timebucket_duration* to intervals of 2017 years. Similarly, CAL_MONTHS(7) does not set the month to July. It sets the *timebucket_duration* to intervals of 7 months.

A calendar day (CAL_DAYS) is a 24 hour period starting at 00:00:00.000000 and ending at 23:59:59.999999; for example, Friday from 00:00:00.000000 to 23:59:59.999999.

A Days time unit is a 24 hour span relative to any moment in time. For example, in a GROUP BY TIME query with time zero equal to 2016-10-01 12:00:00, the day buckets are 2016-10-01 12:00:00.000000 – 2016-10-02 11:59:59.999999. This spans multiple calendar days, but encompasses one 24 hour period representative of a day.

value_expression

The *value_expression* is a column or any expression involving columns (except for scalar subqueries). These expressions are used for grouping purposes not related to time.

There can be one or more comma separated value expressions.

The `value_expression` must not be a column reference to a view column that is derived from a function and cannot contain any ordered analytical or aggregate functions. The `value_expression` cannot be a literal.

timestamp_date_col

A column expression (with an optional table name) that serves as the timecode for a non-PTI table.

seqno_col

A column expression (with an optional table name) that is the sequence number. For a PTI table, it can be `TD_SEQNO` or any other column that acts as a sequence number. For a non-PTI table, *seqno_col* is a column that plays the role of `TD_SEQNO` (because non-PTI tables do not have `TD_SEQNO`).

FILL

The FILL clause allows you to provide values for missing timebucket values. The following variables and values are used by the FILL clause. For more information, see [Using the FILL Clause to Handle Missing Timebuckets](#).

NULLS

The missing timebuckets are returned to the user with a null value for all aggregate results.

numeric_constant

Any Teradata Vantage supported Numeric literal. The missing timebuckets are returned to the user with the specified constant value for all aggregate results. If the data type specified in the FILL clause is incompatible with the input data type for an aggregate function, an error is reported.

PREVIOUS PREV

The missing timebuckets are returned to the user with the aggregate results populated by the value of the closest previous timebucket with a non-missing value. If the immediate predecessor of a missing timebucket is also missing, both buckets, and any other immediate predecessors with missing values, are loaded with the first preceding non-missing value. If a missing timebucket has no predecessor with a result (for example, if the timebucket is the first in the series or all the preceding timebuckets in the entire series are missing), the missing timebuckets are returned to the user with a null value for all aggregate results. The abbreviation PREV may be used instead of PREVIOUS.

NEXT

The missing timebuckets are returned to the user with the aggregate results populated by the value of the closest succeeding timebucket with a non-missing value. If the immediate

successor of a missing timebucket is also missing, both buckets, and any other immediate successors with missing values, are loaded with the first succeeding non-missing value. If a missing timebucket has no successor with a result (for example, if the timebucket is the last in the series or all the succeeding timebuckets in the entire series are missing), the missing timebuckets are returned to the user with a null value for all aggregate results.

pos_int

A 16-bit positive integer with a maximum value of 32767.

A positive integer in the range of 1 to 32767 inclusively.

Usage Notes

GROUP BY TIME and GROUP BY cannot be used together in the same query (this restriction includes GROUP BY ROLLUP, GROUP BY CUBE, and so on).

If GROUP BY TIME is used on a non-PTI table, the USING TIMECODE clause must be included; otherwise, an error is reported.

A supported Time Series function must be used in conjunction with a GROUP BY TIME clause; otherwise, an error is reported.

The timebuckets (which serve as the first level of grouping, if specified) are computed based on time zero. For more information about how time zero is calculated, see [GROUP BY TIME - Time Zero Value](#).

Grouping is determined first by timebucket, and then by all other fields specified in the GROUP BY TIME clause (if any). A timebucket duration is required with the GROUP BY TIME clause. Failure to include it results in an error.

The HAVING clause is supported for filtering results of aggregates with the GROUP BY TIME clause.

The QUALIFY and WITH BY clauses are NOT supported when the GROUP BY TIME clause is present.

The USING TIMECODE and FILL clauses are optional and may only be used with a GROUP BY TIME clause.

Note:

The only time a USING TIMECODE clause can be omitted in a GROUP BY TIME query is when there is a single source present anywhere in the scope of the GROUP BY TIME clause that is a PTI table.

The timecode column cannot be specified as a *value_expression* in the GROUP BY TIME clause, whether the timecode column is the system generated TD_TIMECODE column of a PTI table or specified in the USING TIMECODE clause. Similarly, the sequence number column (TD_SEQNO or *seqno_col* of the USING TIMECODE clause) cannot be specified as a *value_expression* in the GROUP BY TIME clause.

The PTI tables require the TD_TIMECODE and TD_SEQNO columns to be non NULL. When the USING TIMECODE clause is used to specify a timecode or sequenced number column, rows with NULL values are filtered out for these columns. The following example shows this:

```

CREATE SET TABLE aggr_table ,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO
(
    TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
    col1 DECIMAL(20,5),
    vc VARCHAR(20) CHARACTER SET LATIN NOT CASESPECIFIC,
    ts TIMESTAMP(6))
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2000-01-01',
COLUMNS(col1), NONSEQUENCED);

SEL $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, MODE(col1)
FROM aggr_table
GROUP BY TIME(CAL_YEARS(100))
USING TIMECODE(ts)
order by 1,2,3;

```

The USING TIMECODE clause causes the query to be rewritten internally to include the following condition on the "ts" column:

```

SEL $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, MODE(col1)
FROM aggr_table
WHERE ts IS NOT NULL
GROUP BY TIME(CAL_YEARS(100))
USING TIMECODE(ts)
order by 1,2,3;

```

Any conditions added implicitly are chained to any existing conditions using an AND condition.

Time series queries with unbounded time (GROUP BY TIME (*)) can have at most one DELTA_T aggregate specified in the SELECT list. DELTA_T is the only aggregate function that can be used with unbounded time.

Limitations

GROUP BY TIME works only if applied directly on a base PTI table. In all other cases, including joins, USING TIMECODE is used to explicitly specify the timecode column for the times series aggregate. For example:

```

ct t1(a1 int, b1 int, c1 int) primary index (a1);
insert into t1(0,0,0);
insert into t1(1,1,1);

```

The join spool is just like any other join in Teradata. There is nothing specific to Time Series.

```
SELECT * FROM OCEAN_BUOYS, t1 WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06
08:00:00' AND TIMESTAMP '2014-01-06 10:30:00' AND buoyid = t1.b1;
```

Result:

TD_TIMECODE	BUOYID	SALINITY	TEMPERATURE	a1	b1	c1
2014-01-06 09:02:25.122200	1	55	78	1	1	1
2014-01-06 08:00:00.000000	0	55	10	0	0	0
2014-01-06 09:02:25.122200	1	55	71	1	1	1
2014-01-06 08:08:59.999999	0	55	?	0	0	0
2014-01-06 09:03:25.122200	1	55	79	1	1	1
2014-01-06 08:10:00.000000	0	55	10	0	0	0
2014-01-06 09:01:25.122200	1	55	70	1	1	1
2014-01-06 08:09:59.999999	0	55	99	0	0	0
2014-01-06 09:03:25.122200	1	55	72	1	1	1
2014-01-06 08:10:00.000000	0	55	100	0	0	0
2014-01-06 09:01:25.122200	1	55	77	1	1	1

USING TIMECODE Clause

The USING TIMECODE clause designates columns to use for timecode (TD_TIMECODE or the timecode specified in the USING TIMECODE clause) and sequence number (TD_SEQNO or *seqno_col*) in the GROUP BY TIME queries. It is used implicitly for PTI tables but can also be specified explicitly by the user.

The USING TIMECODE clause allows you to specify which columns of the available source columns will serve as the timecode and optional sequence number for the purpose of grouping by time. If this information can be inferred (as explained in [Usage Notes](#)), the USING TIMECODE clause is not needed. If not, then any columns from any source may be specified, if they are of the correct data type, to represent the timecode and (optional) sequence number for the duration of this query. Note, you must choose carefully because using columns which do not accurately represent the timecode and optional sequence number for the source data will likely result in unexpected and potentially non-deterministic results.

Usage Notes

- The USING TIMECODE clause must be specified after the GROUP BY TIME clause.
- The database may infer TD_TIMECODE as the timecode column when there is a single source present anywhere in the scope of the GROUP BY TIME clause that is a PTI table, if the desired columns are not explicitly stated in the USING TIMECODE clause.
- The database may infer TD_SEQNO as the sequence number column when there is a single source present anywhere in the scope of the GROUP BY TIME clause that is a sequenced PTI table, if the desired columns are not explicitly stated in the USING TIMECODE clause. Note, to infer the sequence number, the source must be a sequenced PTI table.
- Column expressions are not allowed in the USING TIMECODE clause; only column names or column aliases are allowed.
- When the USING TIMECODE clause is not specified:
 - If there is a PTI table in the GROUP BY TIME query, the database will infer TD_TIMECODE as the timecode column if there is only one PTI table.
 - If there is no PTI table in the GROUP BY TIME query an error is reported.
 - If there are Multiple PTI tables in the GROUP BY TIME query an error is reported.
 - Note that these restrictions apply per query block and all Vantage rules apply when specifying columns and their sources implicitly. In particular, the USING TIMECODE clause is explicitly needed when a GROUP BY TIME clause is used in a subquery with no FROM clause.

Examples: GROUP BY TIME and USING TIMECODE Clauses

Table and Data Definition for GROUP BY TIME and USING TIMECODE Examples

The table defined below is used in the following examples, in addition to the tables specified in [Table and Data Definition for Time Series Aggregates Examples](#).

```
CREATE TABLE non_pti_tbl_seq (
  timecode TIMESTAMP(6) NOT NULL,
  sequenceno INTEGER NOT NULL,
  buoyid INTEGER,
  salinity INTEGER,
  temperature INTEGER);
```

```
INSERT INTO non_pti_tbl_seq
SELECT TD_TIMECODE, TD_SEQNO, buoyid, salinity, temperature FROM ocean_buoys_seq;
```

Example: Override TD_TIMECODE as the DEFAULT TIMECODE Column

PTI tables have their own timecode column. To override TD_TIMECODE as the default timecode column, specify it in the USING TIMECODE clause. For example, suppose the OCEAN_BUOYS table had another column called my_timecode that was of TIMESTAMP(6) type. Note that when the USING TIMECODE clause is not specified, the timecode column is the default system generated TD_TIMECODE column when a PTI table is given in the query:

```
-- use default timecode TD_TIMECODE
select avg(temperature)
from ocean_buoys
group by time(minutes(10));

-- override default
create table my_ocean_buoys as ocean_buoys with no data;
alter table my_ocean_buoys add my_timecode timestamp(6);
show table my_ocean_buoys;
CREATE SET TABLE my_ocean_buoys ,NO FALLBACK ,
      NO BEFORE JOURNAL,
      NO AFTER JOURNAL,
      CHECKSUM = DEFAULT,
      DEFAULT MERGEBLOCKRATIO
      (
        TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN,
        TD_TIMECODE TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN,
        BUOYID INTEGER,
        SALINITY INTEGER,
        TEMPERATURE INTEGER,
        my_timecode TIMESTAMP(6))
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2012-01-01', HOURS(1),
COLUMNS(BUOYID), NONSEQUENCED);

-- use my_timecode instead of TD_TIMECODE.
select avg(temperature)
from my_ocean_buoys
group by time(minutes(100))
using timecode(my_timecode);
```

Examples: Distinguish between Multiple Candidate Columns

When there are multiple candidate columns that can be inferred to be the timecode column, the USING TIMECODE clause provides a way to disambiguate which to use; for example, when there are joins between PTI tables. If the USING TIMECODE clause is not specified in ambiguous cases, an error is reported.

Example: USING TIMECODE Clause with Multiple Candidate Columns

The example shows explicitly specifying the USING TIMECODE clause in a join:

```
select avg(b.temperature)
from ocean_buoys_seq a , ocean_buoys_seq b
```



```
group by time(minutes(100))
using timecode(a.td_timecode);
```

Example: Invalid Example with Ambiguous Multiple Candidate Columns

The example shows that the timecode is ambiguous without specifying the USING TIMECODE clause in this join:

```
select avg(a.temperature)
from ocean_buoys_seq a, ocean_buoys b
group by time(minutes(100));
```

```
*** Failure 4359 Time Series: Timecode specification is ambiguous.
```

Examples: GROUP BY TIME with non-PTI Tables

Because GROUP BY TIME queries can be used with non-PTI tables, the USING TIMECODE clause provides a way to specify the timecode to use. If the USING TIMECODE clause is not specified when there are no PTI tables to infer a timecode from, an error is reported.

Example: GROUP BY TIME and USING TIMECODE with non-PTI Table

A non-PTI table does not have a primary time index, so specify the timecode explicitly with the USING TIMECODE clause:

```
select avg(temperature)
from non_pti_tbl_seq
group by time(minutes(10))
using timecode(timecode);
```

Example: Invalid GROUP BY TIME with non-PTI Table

A non-PTI table does not have a primary time index, so if the USING TIMECODE clause is omitted an error is reported:

```
select avg(temperature)
from non_pti_tbl_seq
group by time(minutes(10));
```

```
*** Failure 4359 Time Series: GROUP BY TIME clause may not be used without a
TIMECODE specification.
```

Examples: GROUP BY TIME and TD_SEQNO Column

The USING TIMECODE clause allows you to specify an optional sequence number column. Results are further grouped and ordered by the sequence number column if values in the timecode column are the same. The default system generated sequence number column, TD_SEQNO, is inferred if there is a single sequenced PTI table found. If there are multiple candidates for the TD_SEQNO column to use, then no sequence number is inferred by the database.

Note:

You can use the USING TIMECODE clause to override this behavior by specifying a timecode but no sequence number.

The example infers TD_SEQNO as the sequence number column:

```
select avg(temperature)
from ocean_buoys_seq
group by time(minutes(10));
```

The next example shows that TD_SEQNO is not inferred as the sequence number column because the USING TIMECODE clause overrides the inference. If a sequence number is not specified in the USING TIMECODE clause that implies a sequence number should not be used for the purpose of grouping:

```
select avg(temperature)
from ocean_buoys_seq
group by time(minutes(10))
using timecode(td_timecode);
```

The example shows that TD_SEQNO is not inferred as the sequence number column because there are multiple sequenced PTI tables:

```
select avg(b.temperature)
from ocean_buoys_seq a , ocean_buoys_seq b
group by time(minutes(100))
using timecode(a.td_timecode);
```

If your data is sequenced in a non-PTI table, then you can specify a sequence number to use from that table as well. No columns are inferred by the USING TIMECODE clause for non-PTI sources.

The example shows how to specify the sequence number:

```
select avg(temperature)
from non_pti_tbl_seq
```

```
group by time(minutes(10))
using timecode(timecode, sequenceno);
```

Examples: GROUP BY TIME with Multiple Sources in a Query

When there are multiple possible sources in a query, only PTI tables are considered when the database is inferring the timecode column. The TD_TIMECODE column is inferred by the database only when it is not ambiguous.

Example: Multiple Sources in a Query

In the example only ocean_buoys is considered when inferring the timecode, so no USING TIMECODE clause is necessary.

```
sel avg(a.temperature), mode(b.temperature)
from ocean_buoys a inner join ocean_buoys_nonpti b
on a.buoyid = b.buoyid
group by time(minutes(10));
```

Example: Invalid Example with Ambiguous Timecode Specification

In the example, TD_TIMECODE is ambiguous so it cannot be inferred and an error is reported. Because both join tables have TD_TIMECODE, USING TIMECODE is required to specify which TD_TIMECODE to use.

```
sel avg(a.temperature), mode(b.temperature)
from ocean_buoys a inner join ocean_buoys_seq b
on a.buoyid = b.buoyid
group by time(minutes(10));
```

```
*** Failure 4359 Time Series: Timecode specification is ambiguous.
```

Examples: Invalid Examples with GROUP BY TIME and No Timecode Specification

If there are no PTI tables from which to infer the timecode an error is returned.

In the following examples, the USING TIMECODE clause is required because none of the source tables are PTI tables.

```
sel avg(a.temperature), mode(b.temperature)
from non_pti_tbl_seq a inner join ocean_buoys_nonpti b
on a.buoyid = b.buoyid
group by time(minutes(10));
```

```
*** ** Failure 4359 Time Series: GROUP BY TIME clause may not be used without
a TIMECODE specification.
```

```
sel avg(a.temperature), mode(b.temperature)
from non_pti_tbl_seq a, ocean_buoys_nonpti b
group by time(minutes(10));
```

```
*** Failure 4359 Time Series: GROUP BY TIME clause may not be used without a
TIMECODE specification.
```

```
sel avg(a.temperature), mode(a.temperature)
from non_pti_tbl_seq a
group by time(minutes(10));
```

```
*** Failure 4359 Time Series: GROUP BY TIME clause may not be used without a
TIMECODE specification.
```

Examples: GROUP BY TIME and Subqueries

The rules and restrictions for inferring the timecode are per query, so if there are subqueries you may need to specify the USING TIMECODE clause. For more information on GROUP BY TIME rules and restrictions, see [Usage Notes](#) and for the rules and restrictions on USING TIMECODE see [Usage Notes](#).

Example: GROUP BY TIME with Subqueries

In the example, the timecode is inferred in the inner subquery and the outer query specifies the USING TIMECODE clause:

```
sel avg(temperature) as a
from non_pti_tbl_seq
group by time(minutes(10))
using timecode(timecode)
having a IN ( sel count(*)
              from ocean_buoys
              group by time(minutes(1)) );
```

Example: Invalid GROUP BY TIME with Subqueries

In the example, the timecode is inferred in the inner subquery but not in the outer query, so an error is reported:

```
sel avg(temperature) as a
from non_pti_tbl_seq
```

```
group by time(minutes(10))
having a IN ( sel count(*)
             from ocean_buoys
             group by time(minutes(1)) );
```

*** Failure 4359 Time Series: GROUP BY TIME clause may not be used without a TIMECODE specification.

Example: GROUP BY TIME with a Derived Table

The following example again shows the rules and restrictions for inferring the timecode are per query, this time showing an inner and an outer query. The GROUP BY TIME query in the inner query does not need a USING TIMECODE clause because there is a single PTI table from which to infer the timecode column. However, the outer GROUP BY TIME query needs a USING TIMECODE clause because the derived table (min_temp) is not a PTI table, so no timecode column can be inferred:

```
SELECT AVG(min_temp)
FROM
(
  SELECT MIN(temperature)
  FROM OCEAN_BUOYS
  GROUP BY TIME(minutes(1) AND BUOYID)
) AS "inner" (min_temp)
GROUP BY TIME(minutes(10));
```

*** Failure 4359 Time Series: GROUP BY TIME clause may not be used without a TIMECODE specification.

The following example uses the USING clause to specify the TD_TIMECODE derived from a derived table for the outer block:

```
SELECT AVG(min_temp)
FROM
(
  SELECT MIN(temperature), BEGIN($td_timecode_range) d1
  FROM OCEAN_BUOYS
  GROUP BY TIME(minutes(1))
) AS "inner" (min_temp, td_timecode)
GROUP BY TIME(minutes(10)) USING TIMECODE(td_timecode);
```

*** Failure 3706 Syntax error:
Virtual columns \$TD_TIMECODE_RANGE and \$TD_GROUP_BY_TIME are not allowed in the GROUP BY TIME clause.

Examples: GROUP BY TIME and UNION

The following examples again show the rules for inferring the timecode are per query, this time showing this rule with UNIONS.

Example: GROUP BY TIME with UNION

The following example shows that the GROUP BY TIME queries inside the union do not need USING TIMECODE clauses because the timecode can be inferred in each subquery in these particular examples.

```
select Q.avrg, Q.md, Q.grp
from
(
  sel avg(temperature) as avrg, mode(temperature) as md, 1 as grp
  from OCEAN_BUOYS
  group by time(minutes(5) and buoyid)

  union

  sel avg(temperature) as avrg, mode(temperature) as md, 0 as grp
  from OCEAN_BUOYS
  group by time(minutes(10) and buoyid)
) as Q
order by 1;
```

Example: Invalid GROUP BY TIME with UNION

The example shows there is no timecode to infer for the query after the UNION:

```
select Q.avrg, Q.md, Q.grp
from
(
  sel avg(temperature) as avrg, mode(temperature) as md, 1 as grp
  from ocean_buoys
  group by time(minutes(5) and buoyid)

  union

  sel avg(temperature) as avrg, mode(temperature) as md, 0 as grp
  from ocean_buoys_nonpti
  group by time(minutes(10) and buoyid)
) as Q

order by 1;
```

*** Failure 4359 Time Series: GROUP BY TIME clause may not be used without an implicit or explicit TIMECODE specification

System Virtual Columns Returned by Time Series Aggregates

When a GROUP BY TIME query is executed, two system virtual columns are generated and are accessible to users: \$TD_TIMECODE_RANGE and \$TD_GROUP_BY_TIME.

\$TD_GROUP_BY_TIME and \$TD_TIMECODE_RANGE exist only in the presence of a GROUP BY TIME clause that is in the same scope as the virtual columns. A time series supported aggregate function must be used whenever virtual columns are used. However, these system virtual columns cannot be used as arguments to time series supported aggregate functions.

Virtual columns can be used in the SELECT and ORDER BY clauses. Within those clauses, virtual columns may be used in any expression including passed into UDFs and in cast expressions (such that they can change the format associated with embedded timestamps). However, virtual columns cannot be used with WHERE, HAVING, or GROUP BY TIME clauses. Virtual columns cannot be column names in a time series table.

When GROUP BY TIME is used with unbounded time (such as, GROUP BY TIME(*)), the following rules apply to the system virtual columns:

- \$TD_GROUP_BY_TIME: This always has a value of 1, since there is only one timebucket
- \$TD_TIMECODE_RANGE: This is composed of the first and last timecode values read for the group.

\$TD_TIMECODE_RANGE

\$TD_TIMECODE_RANGE is a virtual system column that specifies the range of time over which an aggregate result is computed. Its data type is period(timestamp(6) with time zone). The timecode range column can store a time range such as: ('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:30:00.000000+00:00'). It can be used in the SELECT and ORDER BY clauses.

To retrieve and use the column, enter \$TD_TIMECODE_RANGE in the SELECT or ORDER BY clause:

- The default TITLE is TIMECODE_RANGE.
- The user may change the name of the default title with the AS clause.
- The data type of the column is the Period Data Type (PDT).
- Use the \$TD_TIMECODE_RANGE name label in the same manner as you would use a PDT name label:
 - It can be referenced directly
 - It can launch any of the PDT associated FUNCTIONS ...
BEGIN(\$TD_TIMECODE_RANGE), END(\$TD_TIMECODE_RANGE)
 - You can use it in any expression (including passing it into functions)

- You can use it in an ORDER BY clause similar to other PDT columns

\$TD_TIMECODE_RANGE Values

The returned values follow the PDT convention, so the specified range includes the beginning time but does not include the ending time. The beginning time of the range associated with a particular GROUP BY TIME interval number is:

- For a single range: the beginning time is the starting time of the range.
- For multiple ranges separated by ORs: the beginning time is the earliest starting time among all the ranges specified.

For more information, see [GROUP BY TIME - Time Zero Value](#).

In the following example there is just a single time range specified as part of the query:

```
SELECT ... WHERE TD_TIMECODE BETWEEN TIMESTAMP '2016-10-01 08:00:00' AND
TIMESTAMP '2016-10-01 09:00:00' GROUP BY TIME(MINUTES(15))
```

Where:

- GROUP BY TIME time zero value is 2016-10-01 08:00:00
- GROUP BY TIME interval number is 15
- GROUP BY TIME interval duration is MINUTES

For more information, see [GROUP BY TIME - Time Zero Value](#).

This produces results with the numbering 1,2,3,4:

GROUP BY TIME (MINUTES(15))	TIMECODE_RANGE
1	'2016-10-01 08:00:00', '2016-10-01 08:15:00'
2	'2016-10-01 08:15:00', '2016-10-01 08:30:00'
3	'2016-10-01 08:30:00', '2016-10-01 08:45:00'
4	'2016-10-01 08:45:00', '2016-10-01 09:00:00'

The example snippet shows multiple time ranges specified as part of the query:

```
SELECT ... WHERE TD_TIMECODE BETWEEN TIMESTAMP '2016-10-01 08:00:00' AND
TIMESTAMP '2016-10-01 09:00:00'
OR TIMECODE BETWEEN TIMESTAMP '2016-10-01 12:15:00' AND
TIMESTAMP '2016-10-01 13:30:00'
GROUP BY TIME(MINUTES(15))
```

Where (in this case there were multiple ranges):

- GROUP BY TIME time zero value is '2016-10-01 08:00:00' because it is the earliest time zero among both of the ranges
- GROUP BY TIME interval number is 15
- GROUP BY TIME interval number duration is MINUTES

This produces the following results:

GROUP BY TIME (MINUTES(15))	TIMECODE_RANGE
1	'2016-10-01 08:00:00', '2016-10-01 08:15:00'
2	'2016-10-01 08:15:00', '2016-10-01 08:30:00'
3	'2016-10-01 08:30:00', '2016-10-01 08:45:00'
4	'2016-10-01 08:45:00', '2016-10-01 09:00:00'
18	'2016-10-01 12:15:00', '2016-10-01 12:30:00'
19	'2016-10-01 12:30:00', '2016-10-01 12:45:00'
20	'2016-10-01 12:45:00', '2016-10-01 13:00:00'
21	'2016-10-01 13:00:00', '2016-10-01 13:15:00'
22	'2016-10-01 13:15:00', '2016-10-01 13:30:00'

\$TD_TIMECODE_RANGE Values and System Time Zone Settings

\$TD_TIMECODE_RANGE values are based on system time zone settings specified by the `tdlocaledef` utility or the following DBS Control Fields:

- 16. System TimeZone Hour
- 17. System TimeZone Minute
- 18. System TimeZone String
- 57. TimeDateWZControl

Consider the following table, data, and DBS Control settings:

```
CREATE TABLE ocean_buoy (BUOYID INTEGER, SALINITY INTEGER, TEMPERATURE INTEGER)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2012-01-01', HOURS(1),
COLUMNS(BUOYID), nonsequenced);

INSERT INTO ocean_buoy VALUES(TIMESTAMP '2014-01-06 08:00:00.000000', 0,
55, 10);
```

DBS Control Fields default system time zone settings:

- 16. System TimeZone Hour = 0
- 17. System TimeZone Minute = 0

- 57. TimeDateWZControl = 2

The following query:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, AVG(temperature)
FROM ocean_buoy
GROUP BY TIME(HOURS(1)) ORDER BY 1;
```

Returns this result:

TIMECODE_RANGE	GROUP BY TIME(HOURS(1))	Average(TEMPERATURE)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 09:00:00.000000+00:00')	17673	10

If the DBS Control Fields settings are the following:

- 16. System TimeZone Hour = 8
- 17. System TimeZone Minute = 0
- 57. TimeDateWZControl = 2

The following query:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, AVG(temperature)
FROM ocean_buoy
GROUP BY TIME(HOURS(1)) ORDER BY 1;
```

Returns this result:

TIMECODE_RANGE	GROUP BY TIME(HOURS(1))	Average(TEMPERATURE)
('2014-01-06 16:00:00.000000+08:00', '2014-01-06 17:00:00.000000+08:00')	17673	10

If the DBS Control Fields settings are the following:

- 16. System TimeZone Hour = 5
- 17. System TimeZone Minute = 0
- 57. TimeDateWZControl = 3

The following query:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, AVG(temperature)
FROM ocean_buoy
GROUP BY TIME(HOURS(1)) ORDER BY 1;
```

Returns this result:

TIMECODE_RANGE	GROUP BY TIME(HOURS(1))	Average(TEMPERATURE)
('2014-01-06 08:00:00.000000+05:00', '2014-01-06 09:00:00.000000+05:00')	17668	10

If you use the `tdlocaledef` utility to set the `TimeZoneString` to "America Pacific", then the following query:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, AVG(temperature)
FROM ocean_buoy
GROUP BY TIME(HOURS(1)) ORDER BY 1;
```

Returns this result:

TIMECODE_RANGE	GROUP BY TIME(HOURS(1))
('2014-01-06 00:00:00.000000-08:00', '2014-01-06 01:00:00. 000000-08:00')	17673

Ragged \$TD_TIMECODE_RANGE Values

When any timebucket of a range is specified so that it does not complete an entire timebucket (based on the duration specified) the resulting `$TD_TIMECODE_RANGE` value reflects this discrepancy as shown in this example:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 08:30:00' OR
TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 10:02:00' AND TIMESTAMP
'2014-01-06 10:15:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 2, 3;
```

Result:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Average (TEMPERATURE)	Count (*)
('2014-01-06 08:00:00. 000000+00:00', '2014-01-06 08: 10:00.000000+00:00')	1	0	54	3
('2014-01-06 08:10:00. 000000+00:00', '2014-01-06 08: 20:00.000000+00:00')	2	0	55	2

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Average (TEMPERATURE)	Count (*)
('2014-01-06 10:02:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	54.25	4
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:15:00.000000+00:00')	14	44	43	1

Related Information

- [\\$TD_GROUP_BY_TIME](#)
- [GROUP BY TIME - Time Zero Value](#)
- [GROUP BY TIME Interval](#)

\$TD_GROUP_BY_TIME

The \$TD_GROUP_BY_TIME system virtual column stores the number of the timebucket, such as 1, 2, 3, and so on.

To retrieve and use the group by time system virtual column, access \$TD_GROUP_BY_TIME in a function or an ORDER BY clause:

- The default TITLE is GROUP BY TIME
- You may change the name of the default title via the AS clause
- The data type of the column is BIGINT
- Utilize the \$TD_GROUP_BY_TIME name label in the same manner as you use BIGINT:
 - You can reference \$TD_GROUP_BY_TIME directly
 - You can use \$TD_GROUP_BY_TIME in any expression (including passing it to functions)
 - You can use \$TD_GROUP_BY_TIME in an ORDER BY clause similar to any other BIGINT column

\$TD_GROUP_BY_TIME Values

Starting at the GROUP BY TIME time zero value, the time continuum is broken into a series of equally sized continuous time segments each having a duration equal to the GROUP BY TIME time interval. For example, if the time zero value is 2016-10-01 08:00 and the GROUP BY TIME time interval is 15 minutes, then the time continuum is broken into a series of continuous segments of size 15 minutes, starting at the time zero value.

The numbering scheme for these time segments is to label the first segment number 1 and then do a positive incremental integral assignment thereafter, such as: 1,2,3,4, ... *Last*. This numbering assignment is referred to as the GROUP BY TIME timebucket number.

The following example shows a single time range specified as part of the query:

```
SELECT ... WHERE TD_TIMECODE BETWEEN TIMESTAMP '2016-10-01 08:00:00' AND
TIMESTAMP '2016-10-01 09:00:00' GROUP BY TIME(MINUTES(15))
```

This produces the following results:

GROUP BY TIME(MINUTES(15))
1
2
3
4

The following example query specifies multiple time ranges:

```
SELECT ... WHERE TD_TIMECODE BETWEEN TIMESTAMP '2016-10-01 08:00:00' AND
TIMESTAMP '2016-10-01 09:00:00'
OR TIMECODE BETWEEN TIMESTAMP '2016-10-01 12:15:00' AND
TIMESTAMP '2016-10-01 13:30:00'
GROUP BY TIME(MINUTES(15))
```

Result:

GROUP BY TIME(MINUTES(15))
1
2
3
4
18
19
20
21
22

Related Information

- [GROUP BY TIME - Time Zero Value](#)
- [GROUP BY TIME Interval](#)

Example: Using the System Virtual Columns in a Query

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:00:00'
GROUP BY TIME(MINUTES(30)) AND BUOYID);
```

Result:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(30))	BUOYID	AVG (TEMPERATURE)
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:30:00.000000+00:00')	3	1	74
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:30:00.000000+00:00')	1	0	55

GROUP BY TIME - Time Zero Value

Every GROUP BY TIME operation has a time zero value associated with it.

If the time series query has just one associated time range, then the time zero value is equal to the starting time associated with the range. For example, in a statement such as,

```
SELECT ... WHERE TD_TIMECODE BETWEEN a and b
```

a is the time zero value.

If the time series query has multiple time ranges specified (with intervening ORs), then the time zero value is the value corresponding to the oldest starting time among the time ranges specified in the WHERE clause. For example, in a statement such as:

```
SELECT ... WHERE TD_TIMECODE BETWEEN a AND b OR TIMECODE BETWEEN c AND d OR
TIMECODE BETWEEN e AND f
```

The starting times from each range (in this case, a, c, and e) are compared and the one with the earliest time is the time zero value.

If the time series query has no specified time range, the GROUP BY TIME operation is applied to the entire time scope of the series. The time zero value is set to the time zero value associated with the PTI table against which this time series query is being issued or if no PTI tables are present time zero is set to the UNIX epoch time (1970-01-01 00:00:00).

Note:

If the USING TIMECODE clause is specified, the column that is used for the time zero value is the column specified in the USING TIMECODE clause.

The following rules explain how time zero is calculated.

Rule 1: Literal Constants

Values for the timecode can only be calculated from literal constants or from an expression that evaluates to a literal constant; for example:

- Values such as `TIMESTAMP '01-01-31 08:00:00'` can be specified.
- An expression, such as `ADD_MONTHS(TIMESTAMP '2013-12-06 08:00:00', 1)`, can be specified if the expression evaluates to a literal.
- Timecode ranges cannot include any column references.

For example:

```
/*PTI Table*/

SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS

WHERE TD_TIMECODE BETWEEN ADD_MONTHS(TIMESTAMP '2013-12-06 08:00:00', 1) AND
TIMESTAMP '2014-01-06 10:30:00'

GROUP BY TIME (MINUTES(10) AND BUOYID)

ORDER BY 2, 3;
```

Result:

TIMECODE_RANGE	GROUP BY TIME(MINUTES(10))	BUOYID	Average(TEMPERATURE)	COUNT(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	54	3
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	55	2
('2014-01-06 09:00:00.000000+00:00',	7	1	74	6

TIMECODE_RANGE	GROUP BY TIME(MINUTES(10))	BUOYID	Average(TEMPERATURE)	COUNT(*)
'2014-01-06 09:10:00.000000+00:00')				
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	50	10
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	43	1

Rule 2: Literals Must Be a Compatible Data Type

The timecode literals must be of a data type that can be converted to the data type of the timecode field either the inferred TD_TIMECODE or the explicit column specified in the USING TIMECODE clause). The previous example (in Rule 1: Literal Constants) shows this.

Rule 3: Predicates Must Be Translatable into TIMECODE Ranges

The predicates against the timecode (either the inferred TD_TIMECODE or the explicit column specified in the USING TIMECODE clause) must be translatable into a series of one or more timecode ranges:

- A number of ranges may be connected by OR.
- In assigning the timebucket number (for example, 1, 2, 3), the earliest starting point (the one closest to negative infinity) is treated as time zero. This means that each timebucket after time zero has its timebucket number computed based on this one time zero (not the starting point of a particular range, but the starting point of all of the ranges).
- All empty timebuckets within specified ranges are filled as specified by the FILL clause. Timebuckets that are excluded based on the ranges specified are not filled.

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE),
COUNT(*) FROM OCEAN_BUOYS
```

```
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 08:30:00' OR
```

```
TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 10:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
```

```
GROUP BY TIME (MINUTES(10) AND BUOYID)
```

```
ORDER BY 2, 3;
```



```
*** Query completed. 4 rows found. 5 columns returned.
```

```
*** Total elapsed time was 1 second.
```

Result:

TIMECODE_RANGE	GROUP BY TIME(MINUTES(10))	BUOYID	Average(TEMPERATURE)	COUNT(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	54	3
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	55	2
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	50	10
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	43	1

Rule 4: Starting Point is Negative Infinity and Ending Point is a Literal

If the starting point of a range is negative infinity and the endpoint is a literal, the following applies:

- For the GROUP BY TIME timebucket number assignment:
 - If the source table is a PTI table, the time zero associated with the source table is used to generate the timebucket number.
 - If the source table is not a PTI table, EPOCH time is used to generate the timebucket number.
- For the FILL clause, each processed series is considered independently:
 - The starting time (the first time relevant to that series) associated with that particular series is the first relevant timebucket.
 - All missing timebuckets found between the first timebucket and the provided end point are filled.

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE <= TIMESTAMP '2014-01-06 09:00:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 2, 3;
```

Result:

TIMECODE_RANGE	GROUP BY TIME(MINUTES(10))	BUOYID	Average(TEMPERATURE)	COUNT(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	106033	0	54	3
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	106034	0	55	2

Rule 5: The Starting Point is a Literal and the End Point is Positive Infinity

If a literal is provided for the starting point of a range and the endpoint is positive infinity, the following applies:

- For GROUP BY TIME, the provided literal starting point is the time zero value used for the timebucket number assignment.
- For the FILL clause, each processed series is considered independently:
 - The provided starting time is time zero.
 - Each series has an associated last row which belongs to the GROUP BY TIME timebucket that constitutes its last timebucket.
 - Missing timebuckets between the first timebucket (specified by the starting time) and the ending point timebucket are filled.

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE >= TIMESTAMP '2014-01-06 08:00:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 2, 3;
```

Result:

TIMECODE_RANGE	GROUP BY TIME(MINUTES(10))	BUOYID	Average(TEMPERATURE)	COUNT(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	54	3
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	55	2

TIMECODE_RANGE	GROUP BY TIME(MINUTES(10))	BUOYID	Average(TEMPERATURE)	COUNT(*)
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	74	6
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	50	10
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	43	1
('2014-01-06 10:30:00.000000+00:00', '2014-01-06 10:40:00.000000+00:00')	16	44	43	1
('2014-01-06 10:50:00.000000+00:00', '2014-01-06 11:00:00.000000+00:00')	18	44	43	1
('2014-01-06 21:00:00.000000+00:00', '2014-01-06 21:10:00.000000+00:00')	79	2	81	3

Rule 6: Starting Point of Negative Infinity and Ending Point of Positive Infinity

If the starting point of a range is negative infinity and the endpoint of a range is positive infinity (for example, `SELECT BUOYID, AVG(TEMP) FROM T1 GROUP BY TIME(MINUTES(15) AND BUOYID)`), the following applies:

- For the `GROUP BY TIME` timebucket number assignment:
 - If the source table is a PTI, the time zero associated with the source table is used to generate the timebucket number.
 - If the source table is not a PTI, EPOCH time is used to generate the timebucket number.
- For the `FILL` clause, each processed series is considered independently:
 - The starting time (the first row relevant to that series) associated with that particular series is the first relevant timebucket.
 - The ending time (the last row relevant to that series) associated with a particular series is the last relevant timebucket.
 - All missing timebuckets found between the first relevant timebucket and the last relevant timebucket point are filled. This is repeated for each series processed.

```

SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 2, 3;

```

Result:

TIMECODE_RANGE	GROUP BY TIME(MINUTES(10))	BUOYID	Average(TEMPERATURE)	COUNT(*)
('2014-01-06 08: 00:00.000000+00:00', '2014-01-06 08:10:00. 000000+00:00')	106033	0	54	3
('2014-01-06 08: 10:00.000000+00:00', '2014-01-06 08:20:00. 000000+00:00')	106034	0	55	2
('2014-01-06 09: 00:00.000000+00:00', '2014-01-06 09:10:00. 000000+00:00')	106039	1	74	6
('2014-01-06 10: 00:00.000000+00:00', '2014-01-06 10:10:00. 000000+00:00')	106045	44	50	10
('2014-01-06 10: 10:00.000000+00:00', '2014-01-06 10:20:00. 000000+00:00')	106046	44	43	1
('2014-01-06 10: 30:00.000000+00:00', '2014-01-06 10:40:00. 000000+00:00')	106048	44	43	1
('2014-01-06 10: 50:00.000000+00:00', '2014-01-06 11:00:00. 000000+00:00')	106050	44	43	1
('2014-01-06 21: 00:00.000000+00:00', '2014-01-06 21:10:00. 000000+00:00')	106111	2	81	3

Example: WHERE Clause Results with a Timecode that Precedes Time Zero

It is possible to structure a WHERE clause such that rows are read that have a timecode less than the computed time zero; for example:

```
CREATE TABLE complexTimeZero(BUOYID INTEGER, SALINITY INTEGER,
TEMPERATURE INTEGER )

PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2012-01-01', HOURS(1),
COLUMNS(BUOYID), nonsequenced);

INSERT INTO complexTimeZero VALUES(TIMESTAMP '2013-01-06 10:00:24.000000', 1,
55, 43);

INSERT INTO complexTimeZero VALUES(TIMESTAMP '2014-01-06 10:00:24.333300', 44,
56, 44);

SELECT AVG(TEMPERATURE)
FROM complexTimeZero
WHERE TD_TIMECODE >= TIMESTAMP'2014-01-01 00:00:00' OR buoyid=1
GROUP BY TIME(MINUTES(10));
```

Based on the condition on TD_TIMECODE, time zero is equal to TIMESTAMP'2014-01-01 00:00:00'.

However, the OR condition will also allow a row with TD_TIMECODE equal to TIMESTAMP '2013-01-06 10:00:24.000000' to be included in the aggregation. In all cases where the timecode is less than time zero, an error is returned indicating the timecode precedes time zero.

GROUP BY TIME Interval

Each GROUP BY TIME operation must have a time interval specified in the GROUP BY TIME clause. For example, in the clause GROUP BY TIME(MINUTES(15)) the time interval is 15 minutes.

Missing Value Imputation

If there are missing values in the data several modes are supported for the imputation of missing values both during the intermediate computation of an aggregate and in the result set for all timebuckets. Notice in the table created in [Table and Data Definition for Time Series Aggregates Examples](#) there are some null values for the TEMPERATURE column and not all the timebuckets are represented.

With missing values, consider the following:

- How the missing values within a timebucket affect the result for that group; for example, should the missing values be ignored, treated as 0, or handled another way.
- If you want a result for each timebucket, you must specify the manner in which this result should be computed (for example, should the result be ignored, returned with a constant value, and so on).

There are different approaches for each issue presented above which affect the overall result obtained. The following describes the supported modes of operation for all time series aggregates and how to specify the desired behavior in SQL.

Note that only the AVERAGE function is used as an example, but the modes described apply to all aggregates.

Handling Missing Values within a Timebucket

When a missing value is present within a timebucket (for example, the NULL value inserted in the TEMPERATURE column in [Table and Data Definition for Time Series Aggregates Examples](#)), the value is ignored and it is not included in the calculation of the aggregate result. This is consistent with the behavior of existing aggregate functions in Vantage.

The example shows the result of an AVERAGE with NULL values:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
AND BUOYID = 0
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 2,3;
```

Result:

TIMECODE_RANGE	GROUP BY TIME(MINUTES(10))	BUOYID	Average(TEMPERATURE)	COUNT(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	54	3
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	55	2

The null value was omitted from the result calculation.

If this behavior is not desired there are strategies to clean the series data so the missing values are removed or updated with a value. For more information on these strategies, see:

- [Option 1: Remove Missing Values](#)
- [Option 2: Replace Missing Values with a Constant Value](#)
- [Option 3: Replace Missing Values with an Estimated Value](#)
- [Using the FILL Clause to Handle Missing Timebuckets](#)

Option 1: Remove Missing Values

You can use the DELETE statement to remove all rows with a missing value. The example shows this for the TEMPERATURE column in [Table and Data Definition for Time Series Aggregates Examples](#).

```
DELETE FROM OCEAN_BUOYS WHERE TEMPERATURE IS NULL;
```

After removing the NULL rows, run your query; for example, run a query that performs an AVERAGE:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
AND BUOYID=0
GROUP BY TIME (MINUTES(10)) AND BUOYID
ORDER BY 2,3;
```

Result: Note the result in the first row.

TIMECODE_RANGE	GROUP BY TIME(MINUTES(10))	BUOYID	Average(TEMPERATURE)	COUNT(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	54	2
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	55	2

Option 2: Replace Missing Values with a Constant Value

You can use the UPDATE statement to update all rows with a missing value to a constant value. The example shows this for the TEMPERATURE column in [Table and Data Definition for Time Series Aggregates Examples](#).

In the example, all null values are set to have a value of 50:

```
UPDATE OCEAN_BUOYS SET TEMPERATURE=50 WHERE TEMPERATURE IS NULL;
```

The example shows a query that performs an AVERAGE after setting the null values to a constant:

```

SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
AND BUOYID=0
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 2,3;

```

Result: Note the Average in the first row has been affected.

TIMECODE_RANGE	GROUP BY TIME(MINUTES(10))	BUOYID	Average(TEMPERATURE)	COUNT(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	53	3
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	55	2

Option 3: Replace Missing Values with an Estimated Value

You can estimate a value to update the rows with a missing value.

Note:

The value loaded can be the result of any function, including other aggregates.

The example shows this for the TEMPERATURE column in [Table and Data Definition for Time Series Aggregates Examples](#). The example updates all missing values in each timebucket with the average of the present values:

```

Create table ocean_buoys2 as ocean_buoys with no data;
INSERT INTO OCEAN_BUOYS2 VALUES(TIMESTAMP '2014-01-06 08:00:00.000000',
0, 55, );
INSERT INTO OCEAN_BUOYS2 VALUES(TIMESTAMP '2014-01-06 08:09:59.999999',
0, 55, );
INSERT INTO OCEAN_BUOYS2 VALUES(TIMESTAMP '2014-01-06 09:01:25.122200',
1, 55, );
INSERT INTO OCEAN_BUOYS2 VALUES(TIMESTAMP '2014-01-06 09:02:25.122200',
1, 55, );

```



```

MERGE INTO OCEAN_BUOYS2
USING (SELECT TD_TIMECODE, BUOYID, Avg(TEMPERATURE) FROM OCEAN_BUOYS GROUP BY
(TD_TIMECODE, BUOYID) WHERE TEMPERATURE IS NOT NULL) AS S(a,b,c)
ON TD_TIMECODE=S.a AND BUOYID=S.b AND TEMPERATURE IS NULL
WHEN MATCHED THEN UPDATE SET TEMPERATURE=S.c;

```

```
select * from ocean_buoys2 order by 1;
```

Result:

TIMECODE	BUOYID	SALINITY	TEMPERATURE
2014-01-06 08:00:00.000000	0	55	10
2014-01-06 08:09:59.999999	0	55	99
2014-01-06 09:01:25.122200	1	55	73
2014-01-06 09:02:25.122200	1	55	74

```

SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
AND BUOYID=0
GROUP BY TIME (MINUTES(10)) AND BUOYID)
ORDER BY 2,3;

```

Result: Note the effect on the Average column.

TIMECODE_RANGE	GROUP BY TIME(MINUTES(10))	BUOYID	Average(TEMPERATURE)	COUNT(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	54	3
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	55	2

Using the FILL Clause to Handle Missing Timebuckets

If an entire timebucket is missing a particular value from the aggregate result rows, you can use the FILL clause to set the values. Several FILL values are provided. The FILL clause must follow a GROUP BY

TIME clause. For information about the FILL clause syntax and the available values, see [GROUP BY TIME Clause](#).

When the FILL clause is omitted from the GROUP BY TIME clause, the missing timebuckets are ignored.

Note:

Keywords are case insensitive.

Sample Query for FILL Clause Examples

The following examples illustrate each of the fill schemes. All the examples use the same initial query (see [Table and Data Definition for Time Series Aggregates Examples](#)), which is missing values for timebuckets 1, 3, 6, 7, 8. The missing timebuckets are shown in the table, although by default (without a FILL clause), they are omitted.

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 09:45:00' AND TIMESTAMP
'2014-01-06 11:45:00'
AND BUOYID=44
GROUP BY TIME (MINUTES(15) AND BUOYID)
ORDER BY 2,3;
```

TIMECODE_RANGE	GROUP BY TIME(MINUTES(15))	BUOYID	AVG(TEMPERATURE)
('2014-01-06 09:45:00.000000+00:00', '2014-01-06 10:00:00.000000+00:00')	1	44	MISSING!
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:15:00.000000+00:00')	2	44	49
('2014-01-06 10:15:00.000000+00:00', '2014-01-06 10:30:00.000000+00:00')	3	44	MISSING!
('2014-01-06 10:30:00.000000+00:00', '2014-01-06 10:45:00.000000+00:00')	4	44	43
('2014-01-06 10:45:00.000000+00:00', '2014-01-06 11:00:00.000000+00:00')	5	44	43
('2014-01-06 11:00:00.000000+00:00', '2014-01-06 11:15:00.000000+00:00')	6	44	MISSING!
('2014-01-06 11:15:00.000000+00:00', '2014-01-06 11:30:00.000000+00:00')	7	44	MISSING!
('2014-01-06 11:30:00.000000+00:00', '2014-01-06 11:45:00.000000+00:00')	8	44	MISSING!

Example: FILL Clause is Omitted

If there are timebuckets in the source data with missing values and the FILL clause is omitted, the timebuckets with missing values are omitted from the result set.

The example shows the results of a query when the FILL clause is omitted and there is missing data in the source table:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 09:45:00' AND TIMESTAMP
'2014-01-06 11:45:00'
AND BUOYID=44
GROUP BY TIME (MINUTES(15)) AND BUOYID)
ORDER BY 2,3;
```

Result:

TIMECODE_RANGE	GROUP BY TIME(MINUTES(15))	BUOYID	AVG(TEMPERATURE)
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:15:00.000000+00:00')	2	44	49
('2014-01-06 10:30:00.000000+00:00', '2014-01-06 10:45:00.000000+00:00')	4	44	43
('2014-01-06 10:45:00.000000+00:00', '2014-01-06 11:00:00.000000+00:00')	5	44	43

Example: FILL with NULLS

Use the FILL clause to replace missing values with NULL for timebuckets with missing values.

The example shows the results of the query when the FILL clause is used with NULLS:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 09:45:00' AND TIMESTAMP
'2014-01-06 11:45:00'
AND BUOYID=44
```

```
GROUP BY TIME (MINUTES(15) AND BUOYID) FILL(NULLS)
ORDER BY 2,3;
```

Result: The timebuckets with no value are included in the result set, with nulls in place of the missing values.

TIMECODE_RANGE	GROUP BY TIME(MINUTES(15))	BUOYID	AVG(TEMPERATURE)
('2014-01-06 09:45:00.000000+00:00', '2014-01-06 10:00:00.000000+00:00')	1	44	?
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:15:00.000000+00:00')	2	44	49
('2014-01-06 10:15:00.000000+00:00', '2014-01-06 10:30:00.000000+00:00')	3	44	?
('2014-01-06 10:30:00.000000+00:00', '2014-01-06 10:45:00.000000+00:00')	4	44	43
('2014-01-06 10:45:00.000000+00:00', '2014-01-06 11:00:00.000000+00:00')	5	44	43
('2014-01-06 11:00:00.000000+00:00', '2014-01-06 11:15:00.000000+00:00')	6	44	?
('2014-01-06 11:15:00.000000+00:00', '2014-01-06 11:30:00.000000+00:00')	7	44	?
('2014-01-06 11:30:00.000000+00:00', '2014-01-06 11:45:00.000000+00:00')	8	44	?

Examples: FILL with a Constant Value

Use the FILL clause to replace missing values with a constant value for timebuckets with missing values. The data type of the constant must be compatible with the data type of the data being aggregated.

Example: Invalid Case: Fill with an Incompatible Constant Value

In the example the data type of the constant is incompatible with the data type of the data being aggregated, in this case TEMPERATURE:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 09:45:00' AND TIMESTAMP
'2014-01-06 11:45:00'
AND BUOYID=44
GROUP BY TIME (MINUTES(15) AND BUOYID) FILL('a')
```

```
ORDER BY 2,3;
```

*** Failure 3706 Data type specified in FILL clause is incompatible with input data types to some aggregate functions.

Example: Fill with a Constant Value

The example shows a query when the FILL clause is used with a constant value with a compatible data type:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 09:45:00' AND TIMESTAMP
'2014-01-06 11:45:00'
AND BUOYID=44
GROUP BY TIME (MINUTES(15) AND BUOYID) FILL(45)
ORDER BY 2,3;
```

Result: The timebuckets with no value are included in the result set with the specified constant value returned in place of the missing values.

TIMECODE_RANGE	GROUP BY TIME(MINUTES(15))	BUOYID	AVG(TEMPERATURE)
('2014-01-06 09:45:00.000000+00:00', '2014-01-06 10:00:00.000000+00:00')	1	44	45
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:15:00.000000+00:00')	2	44	49
('2014-01-06 10:15:00.000000+00:00', '2014-01-06 10:30:00.000000+00:00')	3	44	45
('2014-01-06 10:30:00.000000+00:00', '2014-01-06 10:45:00.000000+00:00')	4	44	43
('2014-01-06 10:45:00.000000+00:00', '2014-01-06 11:00:00.000000+00:00')	5	44	43
('2014-01-06 11:00:00.000000+00:00', '2014-01-06 11:15:00.000000+00:00')	6	44	45
('2014-01-06 11:15:00.000000+00:00', '2014-01-06 11:30:00.000000+00:00')	7	44	45
('2014-01-06 11:30:00.000000+00:00', '2014-01-06 11:45:00.000000+00:00')	8	44	45

Example: FILL with Previous Timebucket Value

Use the FILL(PREVIOUS) or FILL(PREV) clause to replace missing values with the previous timebucket value for timebuckets with missing values.

Note:

FILL(PREVIOUS) and FILL(PREV) are identical and produce the same results.

The example shows the results of the query when the FILL clause is used with PREVIOUS/PREV:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 09:45:00' AND TIMESTAMP
'2014-01-06 11:45:00'
AND BUOYID=44
GROUP BY TIME (MINUTES(15) AND BUOYID) FILL(PREVIOUS)
ORDER BY 2,3;
```

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 09:45:00' AND TIMESTAMP
'2014-01-06 11:45:00'
AND BUOYID=44
GROUP BY TIME (MINUTES(15) AND BUOYID) FILL(PREV)
ORDER BY 2,3;
```

Result: All timebuckets are included as part of the result set. The missing value for timebucket #1 is null because there is no previous timebucket from which to infer the value. The missing value for timebucket #3 is set to the value of its immediate non-missing predecessor (timebucket #2, in this case) resulting in a value of 49. The missing values for timebuckets #6, #7 and #8 are all set to their immediate non-missing predecessor (timebucket #5, in this case) resulting in a value of 43.

Note:

The results of both queries are identical, so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME(MINUTES(15))	BUOYID	AVG(TEMPERATURE)
('2014-01-06 09:45:00.000000+00:00', '2014-01-06 10:00:00.000000+00:00')	1	44	?
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:15:00.000000+00:00')	2	44	49

TIMECODE_RANGE	GROUP BY TIME(MINUTES(15))	BUOYID	AVG(TEMPERATURE)
('2014-01-06 10:15:00.000000+00:00', '2014-01-06 10:30:00.000000+00:00')	3	44	49
('2014-01-06 10:30:00.000000+00:00', '2014-01-06 10:45:00.000000+00:00')	4	44	43
('2014-01-06 10:45:00.000000+00:00', '2014-01-06 11:00:00.000000+00:00')	5	44	43
('2014-01-06 11:00:00.000000+00:00', '2014-01-06 11:15:00.000000+00:00')	6	44	43
('2014-01-06 11:15:00.000000+00:00', '2014-01-06 11:30:00.000000+00:00')	7	44	43
('2014-01-06 11:30:00.000000+00:00', '2014-01-06 11:45:00.000000+00:00')	8	44	43

Example: FILL with Next Timebucket Value

For timebuckets with missing values, use the FILL(NEXT) clause to replace the missing values with the next non-missing timebucket value. If every succeeding timebucket is also missing values, the timebucket is set to NULL.

The example shows the results of the query when the FILL clause is used with NEXT:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 09:45:00' AND TIMESTAMP
'2014-01-06 11:45:00'
AND BUOYID=44
GROUP BY TIME (MINUTES(15) AND BUOYID) FILL(NEXT)
ORDER BY 2,3;
```

Result: All timebuckets are included as part of the result set. The missing value for timebucket #1 is set to the value of its immediate non-missing successor (timebucket #2, in this case) resulting in a value of 49. The missing value for timebucket #3 is set to the value of its immediate non-missing successor (timebucket #4, in this case) resulting in a value of 43. The missing values for timebuckets #6, #7 and #8 are all set to null because there is no succeeding timebucket from which to infer the value.

TIMECODE_RANGE	GROUP BY TIME(MINUTES(15))	BUOYID	AVG(TEMPERATURE)
('2014-01-06 09:45:00.000000+00:00', '2014-01-06 10:00:00.000000+00:00')	1	44	49

TIMECODE_RANGE	GROUP BY TIME(MINUTES(15))	BUOYID	AVG(TEMPERATURE)
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:15:00.000000+00:00')	2	44	49
('2014-01-06 10:15:00.000000+00:00', '2014-01-06 10:30:00.000000+00:00')	3	44	43
('2014-01-06 10:30:00.000000+00:00', '2014-01-06 10:45:00.000000+00:00')	4	44	43
('2014-01-06 10:45:00.000000+00:00', '2014-01-06 11:00:00.000000+00:00')	5	44	43
('2014-01-06 11:00:00.000000+00:00', '2014-01-06 11:15:00.000000+00:00')	6	44	?
('2014-01-06 11:15:00.000000+00:00', '2014-01-06 11:30:00.000000+00:00')	7	44	?
('2014-01-06 11:30:00.000000+00:00', '2014-01-06 11:45:00.000000+00:00')	8	44	?

Nested Aggregation

Direct nesting of aggregates is not supported in Vantage, such as `SELECT AVG(AVG(column))...`.

However, nested aggregates can be evaluated using a derived table that contains the aggregates to be nested. This is supported by both the `GROUP BY TIME` clause and the `GROUP BY` clause. The following examples show this for the `GROUP BY TIME` clause.

Average of Averages

The two example queries show the average of nested averages. In the first query, the table is a PTI table and the second example shows a non-PTI table.

```
/*PTI Table*/
SELECT AVG(TEMP_AVG) FROM (
  SELECT AVG(TEMPERATURE) AS TEMP_AVG
  FROM OCEAN_BUOYS
  WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
  GROUP BY TIME (MINUTES(10) AND BUOYID)
) AS NESTED_AVG_TABLE;
```

```
/*Non-PTI Table*/
SELECT AVG(TEMP_AVG) FROM (
```



```

SELECT AVG(TEMPERATURE) AS TEMP_AVG
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
) AS NESTED_AVG_TABLE;

```

The results get loaded into the derived table NESTED_AVG_TABLE. They are shown in ascending order, but this is not necessarily the expected order of the values in the derived table.

Note:

The results of both queries are identical, so only one result set is shown.

TEMP_AVG
43
50
54
55
74

Result: The actual result of the average of results (which are nested in the NESTED_AVG_TABLE):

```

Average(TEMP_AVG)
-----
55.2

```

Median of Averages

The two example queries show the median of nested averages. In the first query, the table is a PTI table and the second example shows a non-PTI table:

```

/*PTI Table*/
SELECT MEDIAN(TEMP_AVG) FROM (
  SELECT AVG(TEMPERATURE) AS TEMP_AVG
  FROM OCEAN_BUOYS
  WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
  GROUP BY TIME (MINUTES(10) AND BUOYID)
) AS NESTED_AVG_TABLE;

```

```

/*Non-PTI Table*/
SELECT MEDIAN(TEMP_AVG) FROM (
  SELECT AVG(TEMPERATURE) AS TEMP_AVG
  FROM OCEAN_BUOYS_NONPTI
  WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
  GROUP BY TIME (MINUTES(10) AND BUOYID)
  USING TIMECODE(TIMECODE)
) AS NESTED_AVG_TABLE;

```

The results get loaded into the derived table NESTED_AVG_TABLE. They are shown in ascending order, but this is not necessarily the expected order of the values in the derived table.

Note:

The results of both queries are identical, so only one result set is shown.

TEMP_AVG
43
50
54
55
74

Result: The actual result of the median of results (which are nested in the NESTED_AVG_TABLE):

```

Median(TEMP_AVG)
-----
54

```

Average of Maximums

The two example queries show the average of nested maximums. In the first query, the table is a PTI table and the second example shows a non-PTI table:

```

/*PTI Table*/
SELECT AVG(TEMP_MAX) FROM (
  SELECT MAX(TEMPERATURE) AS TEMP_MAX
  FROM OCEAN_BUOYS

```

```

WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
) AS NESTED_MAX_TABLE;

/*Non-PTI Table*/
SELECT AVG(TEMP_MAX) FROM (
  SELECT MAX(TEMPERATURE) AS TEMP_MAX
  FROM OCEAN_BUOYS_NONPTI
  WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
  GROUP BY TIME (MINUTES(10) AND BUOYID)
  USING TIMECODE(TIMECODE)
) AS NESTED_MAX_TABLE;

```

The results get loaded into the derived table NESTED_MAX_TABLE. They are shown in ascending order, but this is not necessarily the expected order of the values in the derived table.

Note:

The results of both queries are identical, so only one result set is shown.

TEMP_MAX
43
56
79
99
100

Result: The actual result of the maximum of results (which are nested in the NESTED_MAX_TABLE):

```

Average(TEMP_MAX)
-----
75

```

Average of Minimums

The two example queries show the average of nested minimums. In the first query, the table is a PTI table and the second example shows a non-PTI table:

```

/*PTI Table*/
SELECT AVG(TEMP_MIN) FROM (
    SELECT MIN(TEMPERATURE) AS TEMP_MIN
    FROM OCEAN_BUOYS
    WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
    GROUP BY TIME (MINUTES(10) AND BUOYID)
) AS NESTED_MIN_TABLE;

/*Non-PTI Table*/
SELECT AVG(TEMP_MIN) FROM (
    SELECT MIN(TEMPERATURE) AS TEMP_MIN
    FROM OCEAN_BUOYS_NONPTI
    WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
    GROUP BY TIME (MINUTES(10) AND BUOYID)
    USING TIMECODE(TIMECODE)
) AS NESTED_MIN_TABLE;

```

The results get loaded into the derived table NESTED_MIN_TABLE. They are shown in ascending order, but this is not necessarily the expected order of the values in the derived table.

Note:

The results of both queries are identical, so only one result set is shown.

TEMP_MIN
10
10
43
43
70

Result: The actual result of the minimum of results (which are nested in the NESTED_MIN_TABLE):

```

Average(TEMP_MIN)
-----
35

```

Aggregates That Return More Than One Result

Some aggregate functions can create multiple results. For example, when computing the MODE of a set of data, if there is more than one value which is the most frequently observed (a tie of two or more values) each is considered the MODE of the data. The following general rule applies to any aggregates which return more than one result:

- If a query has only one aggregate function with multiple results, all results are returned. There will be more than one result row for at least one timebucket, where the only value which differs in the result rows is the result of the function which had multiple results.
- If more than one aggregate function has more than one result, only one result per timebucket is returned, along with a warning that this occurred. In this case, there is no way of predicting which result will be processed first. This type of request can be seen in [Example: Multiple Results Found for One or More Time Series Aggregate Functions](#).

Example: Multiple Results Found for One or More Time Series Aggregate Functions

The example shows that more than one aggregate function has more than one result. Only one result per timebucket is returned. A warning message is also returned indicating that this has occurred.

```
CREATE TABLE ts_group_by_time_tbl(BUOYID INTEGER, SALINITY INTEGER,
TEMPERATURE INTEGER )
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2012-01-01', HOURS(1),
COLUMNS(BUOYID), nonsequenced);
INSERT INTO ts_group_by_time_tbl VALUES(TIMESTAMP '2014-01-06 10:00:24.000000', 44,
55, 43);
INSERT INTO ts_group_by_time_tbl VALUES(TIMESTAMP '2014-01-06 10:00:24.333300', 44,
56, 44);
INSERT INTO ts_group_by_time_tbl VALUES(TIMESTAMP '2014-01-06 10:10:24.000000', 44,
55, 43);
INSERT INTO ts_group_by_time_tbl VALUES(TIMESTAMP '2014-01-06 10:10:24.333300', 44,
56, 44);

/*should give one result each and 1 warning*/

SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, MODE(TEMPERATURE),
MODE(SALINITY), COUNT(*)
FROM ts_group_by_time_tbl
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 10:00:00' AND TIMESTAMP
'2014-01-06 10:10:00'
GROUP BY TIME (MINUTES(10) AND BUOYID);

*** Query completed. One row found. 6 columns returned.
*** Warning: 4001 Multiple results found for one or more Time Series aggregate
functions in this query, but only one result was returned. To get all results,
resubmit this query with these aggregates isolated.
*** Total elapsed time was 1 second.
```

Result:

```
TIMECODE_RANGE ('2014-01-06 10:00:00.000000+00:00',
'2014-01-06 10:10:00.000000+00:00')
GROUP BY TIME(MINUTES(10))          1
BUOYID                             44
```

MODE(TEMPERATURE)	43
MODE(SALINITY)	55
Count(*)	2

Time Series Aggregate Functions

A set of aggregate functions is provided to support time series data (optionally stored in Primary Time Index (PTI) tables). Additionally, some traditional functions support time series as well. To operate on time series data, both time series-specific functions and traditional functions are invoked in a GROUP BY TIME clause.

Parallel and Single Threaded Aggregate Functions

The aggregate functions are either fully parallel (FP) aggregates or single-threaded (ST) aggregates, depending on how they are evaluated. FP aggregates do not require the entire set of data to be present in order to compute the result. Rather, the result is computed by evaluating the aggregation on small subsets of the data in parallel and then computing a final aggregation of the intermediate results. ST aggregates require the entire set of data to be present in order to compute the result.

You can use the following aggregate functions on time series data in PTI tables by using the GROUP BY TIME clause and in non-PTI tables by using the GROUP BY TIME clause with the USING TIMECODE option.

- AVERAGE
- COUNT
- KURTOSIS
- MAXIMUM
- MINIMUM
- RANK (ANSI)
- SKEW
- STANDARD DEVIATION OF A POPULATION (STDDEV_POP)
- STANDARD DEVIATION OF A SAMPLE (STDDEV_SAMP)
- SUM
- VARIANCE OF A POPULATION (VAR_POP)
- VARIANCE OF A SAMPLE (VAR_SAMP)

EXPLAIN Statement Output for Time Series Aggregate Functions

The output of the EXPLAIN statement includes information about time zero when using VERBOSE EXPLAIN; for example:

```
...
We begin the time series aggregate calculation using
    time zero <time_zero>
...
```

<time_zero> displays the type and constant value calculated as the time zero to use for a GROUP BY TIME query (such as, `TIMESTAMP '2014-01-06 08:00:00.000000+00:00'`). This information appears in the SUM step of the EXPLAIN output.

Other information about time series queries can be deduced from the current EXPLAIN output:

- Information about which optimization plan for single threaded aggregate execution is in use
- Specification of each aggregate function as fully parallel or single threaded
- Specification of any sorting or redistribution performed prior to the aggregation (this is important for single threaded aggregate evaluation)
- Specification of any sorting or redistribution performed after the aggregation (this is important for FILL directive evaluation)
- Execution of FILL directive prints (fill empty groups in the time series with <value>).

Table and Data Definition for Time Series Aggregates Examples

The following SQL creates sample tables with data for use in time series examples.

```
/*PTI Table*/
CREATE TABLE OCEAN_BUOYS(BUOYID INTEGER, SALINITY INTEGER, TEMPERATURE INTEGER )

PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2012-01-01', HOURS(1),
COLUMNS(BUOYID), nonsequenced);

INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 08:00:00.000000', 0, 55, 10);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 08:08:59.999999', 0, 55, );
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 08:09:59.999999', 0, 55, 99);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 08:10:00.000000', 0, 55, 10);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 08:10:00.000000', 0, 55, 100);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 09:01:25.122200', 1, 55, 77);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 09:01:25.122200', 1, 55, 70);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 09:02:25.122200', 1, 55, 78);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 09:02:25.122200', 1, 55, 71);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 09:03:25.122200', 1, 55, 79);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 09:03:25.122200', 1, 55, 72);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 21:01:25.122200', 2, 55, 80);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 21:02:25.122200', 2, 55, 81);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 21:03:25.122200', 2, 55, 82);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:00:24.000000', 44, 55, 43);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:00:24.333300', 44, 55, 43);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:00:25.122200', 44, 55, 43);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:00:26.122200', 44, 55, 43);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:01:25.122200', 44, 55, 53);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:01:25.122200', 44, 55, 54);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:02:25.122200', 44, 55, 53);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:02:25.122200', 44, 55, 55);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:03:25.122200', 44, 55, 53);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:03:25.122200', 44, 55, 56);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:12:00.000000', 44, 55, 43);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:32:12.122200', 44, 55, 43);
INSERT INTO OCEAN_BUOYS VALUES(TIMESTAMP '2014-01-06 10:52:00.000009', 44, 55, 43);

/* Sequenced PTI table */
CREATE TABLE OCEAN_BUOYS_SEQ(BUOYID INTEGER, SALINITY INTEGER, TEMPERATURE INTEGER,
DATES DATE)
```



```

PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2012-01-01', HOURS(1),
COLUMNS(BUOYID), sequenced);

INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:32:12.122200',1, 44, 55,
43, '2014-01-01');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:32:12.122200',1, 22, 25,
23, '2014-01-01');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:52:00.000009',2, 44, 55,
43, '2014-02-02');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:12:00.000000',3, 44, 55,
43, '2014-03-03');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:00:24.000000',4, 44, 55,
43, '2014-04-04');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:00:24.333300',5, 44, 55,
43, '2014-05-05');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:00:25.122200',6, 44, 55,
43, '2014-06-06');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:00:26.122200',7, 44, 55,
43, '2014-07-07');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:01:25.122200',8, 44, 55,
53, '2014-08-08');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:02:25.122200',9, 44, 55,
53, '2014-09-09');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:03:25.122200',10, 44,
55, 53, '2014-10-10');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 09:01:25.122200',11, 1, 55,
70, '2014-11-11');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 09:02:25.122200',12, 1, 55,
71, '2014-12-12');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 09:03:25.122200',13, 1, 55,
72, '2015-01-13');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 21:01:25.122200',14, 2, 55,
80, '2015-02-14');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 21:02:25.122200',15, 2, 55,
81, '2015-03-15');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 21:03:25.122200',16, 2, 55,
82, '2015-04-16');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 08:09:59.999999',17, 0, 55,
99, '2015-05-17');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 08:08:59.999999',18, 0,
55, , '2015-06-18');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 08:10:00.000000',19, 0, 55,
10, '2015-07-19');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:01:25.122200',20, 44,
55, 54, '2015-08-20');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:02:25.122200',21, 44,
55, 55, '2015-09-21');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 10:03:25.122200',22, 44,
55, 56, '2015-10-22');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 09:01:25.122200',23, 1, 55,
77, '2015-11-23');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 09:02:25.122200',24, 1, 55,
78, '2015-12-24');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 09:03:25.122200',25, 1, 55,
79, '2016-01-25');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 08:00:00.000000',26, 0, 55,
10, '2016-02-26');
INSERT INTO OCEAN_BUOYS_SEQ VALUES(TIMESTAMP '2014-01-06 08:10:00.000000',27, 0, 55,
100, '2016-03-27');

/*Non-PTI Table*/

CREATE TABLE OCEAN_BUOYS_NONPTI(TIMECODE TIMESTAMP(6), BUOYID INTEGER, SALINITY
INTEGER, TEMPERATURE INTEGER)

PRIMARY INDEX (BUOYID);

```

```

INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 08:00:00.000000', 0,
55, 10);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 08:08:59.999999',
0, 55, );
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 08:09:59.999999', 0,
55, 99);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 08:10:00.000000', 0,
55, 10);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 08:10:00.000000', 0,
55, 100);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 09:01:25.122200', 1,
55, 77);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 09:01:25.122200', 1,
55, 70);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 09:02:25.122200', 1,
55, 78);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 09:02:25.122200', 1,
55, 71);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 09:03:25.122200', 1,
55, 79);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 09:03:25.122200', 1,
55, 72);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 21:01:25.122200', 2,
55, 80);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 21:02:25.122200', 2,
55, 81);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 21:03:25.122200', 2,
55, 82);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:00:24.000000', 44,
55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:00:24.333300', 44,
55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:00:25.122200', 44,
55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:00:26.122200', 44,
55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:01:25.122200', 44,
55, 53);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:01:25.122200', 44,
55, 54);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:02:25.122200', 44,
55, 53);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:02:25.122200', 44,
55, 55);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:03:25.122200', 44,
55, 53);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:03:25.122200', 44,
55, 56);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:12:00.000000', 44,
55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:32:12.122200', 44,
55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI VALUES(TIMESTAMP '2014-01-06 10:52:00.000009', 44,
55, 43);

```

AVERAGE

To invoke the time series version of this function, use the `GROUP BY TIME` clause. For more information, see [GROUP BY TIME Clause](#).

For more information about this function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Examples: Using AVERAGE with Time Series

Example: Calculating an Average for PTI and non-PTI Tables

The following example shows how the AVG function can be used with both Primary Time Index (PTI) and non-PTI tables.

The examples use the tables and data setup in [Table and Data Definition for Time Series Aggregates Examples](#).

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 2, 3;
```

```
/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 2, 3;
```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	AVG (TEMPERATURE)	COUNT(*)
('2014-01-06 08:00:00. 000000+00:00', '2014-01-06 08: 10:00.000000+00:00')	1	0	54	3
('2014-01-06 08:10:00. 000000+00:00', '2014-01-06 08: 20:00.000000+00:00')	2	0	55	2
('2014-01-06 09:00:00. 000000+00:00', '2014-01-06 09: 10:00.000000+00:00')	7	1	74	6

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	AVG (TEMPERATURE)	COUNT(*)
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	50	10
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	43	1

Example: Using the HAVING Clause to Filter the AVG Function Results for PTI and non-PTI Tables

The following example shows how the AVG function can be used with the HAVING clause to filter the results.

The examples use the tables and data setup in [Table and Data Definition for Time Series Aggregates Examples](#).

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
HAVING AVG(TEMPERATURE) > 50
ORDER BY 2, 3;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
HAVING AVG(TEMPERATURE) > 50
ORDER BY 2, 3;

```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	AVG (TEMPERATURE)	COUNT(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	54	3
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	55	2
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	74	6

BOTTOM

Returns the smallest *number_of_values* in the *value_expression* for each group, with or without ties. BOTTOM is a single-threaded function.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

Return Values

The return values are actual values from the input data set.

Nulls are not included in the computation.

Usage Notes

BOTTOM is valid only for numeric data.

WITH TIES implies that the rows returned include the specified number of rows in the ordered set for each timebucket. It includes any rows where the sort key value is the same as the sort key value in the last row that satisfies the specified number or percentage of rows. If this clause is omitted and ties are found, the earliest value in terms of timecode is returned.

BOTTOM Syntax

```
BOTTOM [ WITH TIES ] ('number_of_values', 'value_expression')
```

Syntax Elements

number_of_values

Teradata integer value representing the number of values to return.

value_expression

A literal or column expression from which the bottom values are taken.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

Examples

Example: Using **BOTTOM** without Ties

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, BOTTOM(2, TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 10:00:00' AND TIMESTAMP
'2014-01-06 10:06:00' AND BUOYID=44
GROUP BY TIME (MINUTES(2) AND BUOYID) FILL(NULLS)
ORDER BY 1, 4, 3;
```

```
/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, BOTTOM(2, TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 10:00:00' AND TIMESTAMP
'2014-01-06 10:06:00' AND BUOYID=44
GROUP BY TIME (MINUTES(2) AND BUOYID)
USING TIMECODE(TIMECODE)
FILL(NULLS)
ORDER BY 1, 4, 3;
```

The results are the same for both tables:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	BOTTOM(2, TEMPERATURE)
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:02:00.000000+00:00')	1	44	43
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:02:00.000000+00:00')	1	44	43
('2014-01-06 10:02:00.000000+00:00', '2014-01-06 10:04:00.000000+00:00')	2	44	53
('2014-01-06 10:02:00.000000+00:00', '2014-01-06 10:04:00.000000+00:00')	2	44	53
('2014-01-06 10:04:00.000000+00:00', '2014-01-06 10:06:00.000000+00:00')	3	44	?

Example: Using BOTTOM with Ties

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, BOTTOM WITH
TIES(2, TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 10:00:00' AND TIMESTAMP
'2014-01-06 10:06:00' AND BUOYID=44
GROUP BY TIME (MINUTES(2) AND BUOYID) FILL(NULLS)
ORDER BY 1, 4, 3;

```

```

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, BOTTOM WITH
TIES(2, TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 10:00:00' AND TIMESTAMP
'2014-01-06 10:06:00' AND BUOYID=44
GROUP BY TIME (MINUTES(2) AND BUOYID)
USING TIMECODE(TIMECODE)
FILL(NULLS)

```

The results are the same for both tables:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	TIMECODE	BUOYID	TEMPERATURE
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:02:00.000000+00:00')	1	2014-01-06 10:00:24.000000	44	43
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:02:00.000000+00:00')	1	2014-01-06 10:00:24.333300	44	43
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:02:00.000000+00:00')	1	2014-01-06 10:00:25.122200	44	43
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:02:00.000000+00:00')	1	2014-01-06 10:00:26.122200	44	43
('2014-01-06 10:02:00.000000+00:00', '2014-01-06 10:04:00.000000+00:00')	2	2014-01-06 10:02:25.122200	44	53

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	TIMECODE	BUOYID	TEMPERATURE
('2014-01-06 10:02:00.000000+00:00', '2014-01-06 10:04:00.000000+00:00')	2	2014-01-06 10:03:25.122200	44	53
('2014-01-06 10:04:00.000000+00:00', '2014-01-06 10:06:00.000000+00:00')	3	?	44	?

COUNT

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

For more information about this function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Example: Using COUNT with Time Series

Example: Calculating a COUNT for PTI and non-PTI Tables

The following example shows how the COUNT function can be used with both Primary Time Index (PTI) and non-PTI tables.

The examples use the tables and data setup in [Table and Data Definition for Time Series Aggregates Examples](#).

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, COUNT(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 2,3,4;
```

```
/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, COUNT(TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 2,3,4;
```


Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Count (TEMPERATURE)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	2
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	2
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	6
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	10
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	1

DELTA_T

Calculates the time difference, or DELTA_T, between a starting and an ending event. The calculation is performed against a time-ordered time series data set.

To invoke the DELTA_T function, use the GROUP BY TIME clause.

Return Value

DELTA_T returns a PERIOD(TIMESTAMP WITH TIME ZONE) type composed of the start and end timecode of each start-end pair.

Use the PERIOD DATA TYPES functions to inspect the results, as described in "Period Functions and Operators" in *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211.

One result is returned per complete start-end pair found within the GROUP BY TIME window. The start-end pair process is as follows:

- If the current source data meets the start condition, the current timecode is saved as the start time.
- If the current source data meets the end condition, and a saved start timecode already exists, the start timecode is saved with the end timecode encountered as a result pair.

The processing algorithm implies that multiple results may be found in each group.

If no start-end pair is encountered, no result row is returned.

Any result of DELTA_T which has a delta of < 1 microsecond (including a delta of 0, in the case of a result which comes from a single point in time) is automatically rounded to 1 microsecond. This is strictly enforced to match Period data type semantics which dictate that a starting and ending bound of a Period type may

not be equivalent. The smallest granularity supported in Vantage is the microsecond, so these results are rounded accordingly.

Usage Notes

You can only explicitly specify DELTA_T in the SELECT list one time, but can further alias DELTA_T in the SELECT list, ORDER BY and HAVING clauses.

Specify the starting and ending conditions using any SQL syntax supported in a WHERE clause. Use DELTA_T to track the amount of time elapsed between two events, such as:

- The time between a user calling a customer service line and reaching a customer service representative.
- The time between minimum and maximum observations (such as temperature, wind speed, or salinity).
- The time between reporting a software defect and its resolution.

Understanding how long a customer waits to speak to a service representative at different times during the day, or on different days during the week, helps companies staff their customer service department according to actual needs. The companies can spend money more efficiently and provide better service to customers.

The DELTA_T function cannot be combined with any other functions.

DELTA_T is the only aggregate function used with GROUP BY TIME(*).

When using GROUP BY TIME with unbounded time (such as GROUP BY TIME(*)), the following rules apply to the system virtual columns:

- \$TD_GROUP_BY_TIME: Always has a value of 1, since there is only one timebucket.
- \$TD_TIMECODE_RANGE: Composed of the first and last timecode values read for the group.

Note that the data being evaluated in the WHERE clauses (for example, the minimum and maximum temperature observation) must belong to the timecode value present in the same row of data. This is the expected behavior. However, this assumption can be violated when joining multiple tables together. It is possible to construct a query where the result of a join causes specific data points (for example, a temperature reading) to be present in a data row with a timecode that is not indicative of when that data point occurred. In such a scenario, it is highly likely that the results are not as expected, or are misleading. Vantage does not detect these types of queries, so you must make sure that you preserve the correlation between data points and timecodes.

DELTA_T Syntax

```
DELTA_T (
  ( 'starting_sql_condition' ),
  ( 'ending_sql_condition' )
)
```

Syntax Elements

starting_sql_condition

Any supported WHERE clause that defines the start of the time period for which you are searching.

ending_sql_condition

Any supported WHERE clause that defines the end of the time period for which you are searching.

Examples

Example: Creating the OCEAN_BUOYS_DELTA_T Table

```
/*PTI Table*/
CREATE TABLE OCEAN_BUOYS_DELTA_T(BUOYID INTEGER, SALINITY INTEGER,
TEMPERATURE INTEGER )
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2012-01-01', HOURS(1),
COLUMNS(BUOYID), nonsequenced);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 08:00:00.000000',
0, 55, 10);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 08:08:59.999999',
0, 55, );
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 08:09:59.999999',
0, 55, 99);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 08:10:00.000000',
0, 55, 10);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 08:10:01.000000',
0, 55, 100);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 09:01:25.122200',
1, 55, 77);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 09:01:25.122200',
1, 55, 70);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 09:02:25.122200',
1, 55, 78);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 09:02:25.122200',
1, 55, 71);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 09:03:25.122200',
1, 55, 79);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 09:03:25.122200',
1, 55, 72);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 21:01:25.122200',
2, 55, 80);
```

```

INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 21:02:25.122200',
2, 55, 81);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 21:03:25.122200',
2, 55, 82);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 10:00:24.000000',
44, 55, 43);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 10:00:24.333300',
44, 55, 43);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 10:00:25.122200',
44, 55, 43);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 10:00:26.122200',
44, 55, 43);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 10:01:25.122200',
44, 55, 53);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 10:01:25.122200',
44, 55, 54);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 10:02:25.122200',
44, 55, 53);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 10:02:25.122200',
44, 55, 55);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 10:03:25.122200',
44, 55, 53);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 10:03:25.122200',
44, 55, 56);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 10:12:00.000000',
44, 55, 43);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 10:32:12.122200',
44, 55, 43);
INSERT INTO OCEAN_BUOYS_DELTA_T VALUES(TIMESTAMP '2014-01-06 10:52:00.000009',
44, 55, 43);

/*Non-PTI Table*/
CREATE TABLE OCEAN_BUOYS_NONPTI_DELTA_T(TIMECODE TIMESTAMP(6), BUOYID INTEGER,
SALINITY INTEGER, TEMPERATURE INTEGER)
PRIMARY INDEX (BUOYID);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
08:00:00.000000', 0, 55, 10);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
08:08:59.999999', 0, 55, );
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
08:09:59.999999', 0, 55, 99);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
08:10:00.000000', 0, 55, 10);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06

```

```

08:10:01.000000', 0, 55, 100);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
09:01:25.122200', 1, 55, 77);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
09:01:25.122200', 1, 55, 70);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
09:02:25.122200', 1, 55, 78);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
09:02:25.122200', 1, 55, 71);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
09:03:25.122200', 1, 55, 79);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
09:03:25.122200', 1, 55, 72);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
21:01:25.122200', 2, 55, 80);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
21:02:25.122200', 2, 55, 81);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
21:03:25.122200', 2, 55, 82);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
10:00:24.000000', 44, 55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
10:00:24.333300', 44, 55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
10:00:25.122200', 44, 55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
10:00:26.122200', 44, 55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
10:01:25.122200', 44, 55, 53);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
10:01:25.122200', 44, 55, 54);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
10:02:25.122200', 44, 55, 53);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
10:02:25.122200', 44, 55, 55);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
10:03:25.122200', 44, 55, 53);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
10:03:25.122200', 44, 55, 56);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
10:12:00.000000', 44, 55, 43);
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
10:32:12.122200', 44, 55, 43);

```

```
INSERT INTO OCEAN_BUOYS_NONPTI_DELTA_T VALUES(TIMESTAMP '2014-01-06
10:52:00.000009', 44, 55, 43);
```

Example: Searching the Minimum and Maximum Observed Temperatures

This example measures the time between minimum and maximum observed temperatures every 10 minutes between 8:00 AM and 10:30 AM on each buoy.

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, t1.BUOYID, DELTA_T((WHERE
TEMPERATURE=MIN_TEMP),(WHERE TEMPERATURE=MAX_TEMP))
FROM (
    SELECT MIN(TEMPERATURE) AS MIN_TEMP, MAX(TEMPERATURE) AS MAX_TEMP, BUOYID
    FROM OCEAN_BUOYS_DELTA_T
    WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
    GROUP BY TIME(MINUTES(30) AND BUOYID)
) AS t1, OCEAN_BUOYS_DELTA_T
WHERE t1.buoyid=OCEAN_BUOYS_DELTA_T.buoyid
GROUP BY TIME(DAYS(1) and t1.buoyid)
USING TIMECODE(OCEAN_BUOYS_DELTA_T.td_timecode)
order by 2, 3;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, t1.BUOYID, DELTA_T((WHERE
TEMPERATURE=MIN_TEMP),(WHERE TEMPERATURE=MAX_TEMP))
FROM (
    SELECT MIN(TEMPERATURE) AS MIN_TEMP, MAX(TEMPERATURE) AS MAX_TEMP, BUOYID
    FROM OCEAN_BUOYS_NONPTI_DELTA_T
    WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
    GROUP BY TIME(MINUTES(30) AND BUOYID)
    USING TIMECODE(TIMECODE)
) AS t1, OCEAN_BUOYS_NONPTI_DELTA_T
WHERE t1.buoyid=OCEAN_BUOYS_NONPTI_DELTA_T.buoyid
GROUP BY TIME(DAYS(1) and t1.buoyid)
USING TIMECODE(OCEAN_BUOYS_NONPTI_DELTA_T.timecode)
order by 2, 3;
```

The results for both tables:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Delta_T(TD_ TIMECODE)
('2014-01-06 00:00:00.000000+00:00', '2014-01-07 00:00:00.000000+00:00')	737	0	('2014-01-06 08:10:00.000000', '2014-01-06 08:10:01.000000')
('2014-01-06 00:00:00.000000+00:00', '2014-01-07 00:00:00.000000+00:00')	737	1	('2014-01-06 09:01:25.122200', '2014-01-06 09:03:25.122200')
('2014-01-06 00:00:00.000000+00:00', '2014-01-07 00:00:00.000000+00:00')	737	44	('2014-01-06 10:00:26.122200', '2014-01-06 10:03:25.122200')

Buoy 44, timebucket 13, results are based on the final measurement meeting the start condition. No result row returned for buoy 44, timebucket 14, because no entries met the start and end conditions.

Example: Creating a Multiset Table to Track Elapsed Time

In this example, create a table with information about parcels sent by a delivery service.

```

/*PTI Table*/
CREATE MULTISET TABLE package_tracking_pti(ParcelNumber INTEGER,
Status VARCHAR(512))
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2012-01-01', HOURS(1),
COLUMNS(ParcelNumber), nonsequenced);
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 08:00:00', 55,
'picked up from customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 08:05:00', 59,
'picked up from customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 09:10:00', 55,
'arrived at receiving station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 09:20:00', 60,
'picked up from customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 10:00:00', 55,
'in transit to destination');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 10:45:00', 60,
'arrived at receiving station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 16:10:00', 55,
'arrived at destination station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 16:30:00', 55,
'in transit to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 17:00:00', 55,
'delivered to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 08:00:00', 75,
'picked up from customer');

```

```

INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 08:05:00', 79,
'picked up from customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 09:10:00', 75,
'arrived at receiving station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 09:20:00', 80,
'picked up from customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 10:00:00', 75,
'in transit to destination');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 10:45:00', 80,
'arrived at receiving station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 16:10:00', 75,
'arrived at destination station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 16:30:00', 75,
'in transit to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-15 17:00:00', 75,
'delivered to customer');

/*Non-PTI Table*/
CREATE MULTISET TABLE package_tracking(ParcelNumber INTEGER, CLOCK_TIME
TIMESTAMP, Status VARCHAR(512));
INSERT INTO package_tracking(55, TIMESTAMP'2016-10-15 08:00:00',
'picked up from customer');
INSERT INTO package_tracking(59, TIMESTAMP'2016-10-15 08:05:00',
'picked up from customer');
INSERT INTO package_tracking(55, TIMESTAMP'2016-10-15 09:10:00',
'arrived at receiving station');
INSERT INTO package_tracking(60, TIMESTAMP'2016-10-15 09:20:00',
'picked up from customer');
INSERT INTO package_tracking(55, TIMESTAMP'2016-10-15 10:00:00',
'in transit to destination');
INSERT INTO package_tracking(60, TIMESTAMP'2016-10-15 10:45:00',
'arrived at receiving station');
INSERT INTO package_tracking(55, TIMESTAMP'2016-10-15 16:10:00',
'arrived at destination station');
INSERT INTO package_tracking(55, TIMESTAMP'2016-10-15 16:30:00',
'in transit to customer');
INSERT INTO package_tracking(55, TIMESTAMP'2016-10-15 17:00:00',
'delivered to customer');
INSERT INTO package_tracking(75, TIMESTAMP'2016-10-15 08:00:00',
'picked up from customer');
INSERT INTO package_tracking(79, TIMESTAMP'2016-10-15 08:05:00',
'picked up from customer');
INSERT INTO package_tracking(75, TIMESTAMP'2016-10-15 09:10:00',
'arrived at receiving station');

```



```

INSERT INTO package_tracking(80, TIMESTAMP'2016-10-15 09:20:00',
'picked up from customer');
INSERT INTO package_tracking(75, TIMESTAMP'2016-10-15 10:00:00',
'in transit to destination');
INSERT INTO package_tracking(80, TIMESTAMP'2016-10-15 10:45:00',
'arrived at receiving station');
INSERT INTO package_tracking(75, TIMESTAMP'2016-10-15 16:10:00',
'arrived at destination station');
INSERT INTO package_tracking(75, TIMESTAMP'2016-10-15 16:30:00',
'in transit to customer');
INSERT INTO package_tracking(75, TIMESTAMP'2016-10-15 17:00:00',
'delivered to customer');

```

Example: Finding Time Elapsed between Shipping and Receiving an Item

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, ParcelNumber, DELTA_T(
    (WHERE Status LIKE 'picked%up%customer'),
    (WHERE Status LIKE 'delivered%customer'))
FROM package_tracking_pti
GROUP BY TIME(* AND ParcelNumber);

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, ParcelNumber, DELTA_T(
    (WHERE Status LIKE 'picked%up%customer'),
    (WHERE Status LIKE 'delivered%customer'))
FROM package_tracking
GROUP BY TIME(* AND ParcelNumber) USING TIMECODE(CLOCK_TIME);

```

The results are the same for both tables:

TIMECODE_RANGE	ParcelNumber	Delta_T(TD_TIMECODE)
('2016-10-16 08:00:00.000000+00:00', '2016-10-17 17:00:00.000000+00:00')	55	('2016-10-15 08:00:00.000000', '2016-10-15 17:00:00.000000')
('2016-10-16 08:00:00.000000+00:00', '2016-10-17 17:00:00.000000+00:00')	75	('2016-10-15 08:00:00.000000', '2016-10-15 17:00:00.000000')

Example: Tracking an Incorrectly Delivered Package

This example is for an incorrectly delivered package. The package, picked up and delivered twice, resulted in two output rows.

```

DELETE package_tracking_pti;
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 08:00:00', 65,
'picked up from customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 08:05:00', 69,
'picked up from customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 09:10:00', 65,
'arrived at receiving station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 09:20:00', 70,
'picked up from customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 10:00:00', 65,
'in transit to destination');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 10:45:00', 70,
'arrived at receiving station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 16:10:00', 65,
'arrived at destination station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 16:30:00', 65,
'in transit to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 17:00:00', 65,
'delivered to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 17:05:00', 65,
'notification of incorrect delivery');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-17 08:05:00', 65,
'picked up from customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-17 09:10:00', 65,
'arrived at receiving station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-17 10:00:00', 65,
'in transit to destination');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-17 16:30:00', 65,
'in transit to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-17 17:00:00', 65,
'delivered to customer');

DELETE package_tracking;
INSERT INTO package_tracking(65, TIMESTAMP'2016-10-16 08:00:00',
'picked up from customer');
INSERT INTO package_tracking(69, TIMESTAMP'2016-10-16 08:05:00',
'picked up from customer');
INSERT INTO package_tracking(65, TIMESTAMP'2016-10-16 09:10:00',
'arrived at receiving station');
INSERT INTO package_tracking(70, TIMESTAMP'2016-10-16 09:20:00',
'picked up from customer');
INSERT INTO package_tracking(65, TIMESTAMP'2016-10-16 10:00:00',

```

```

'in transit to destination');
INSERT INTO package_tracking(70, TIMESTAMP'2016-10-16 10:45:00',
'arrived at receiving station');
INSERT INTO package_tracking(65, TIMESTAMP'2016-10-16 16:10:00',
'arrived at destination station');
INSERT INTO package_tracking(65, TIMESTAMP'2016-10-16 16:30:00',
'in transit to customer');
INSERT INTO package_tracking(65, TIMESTAMP'2016-10-16 17:00:00',
'delivered to customer');
INSERT INTO package_tracking(65, TIMESTAMP'2016-10-16 17:05:00',
'notification of incorrect delivery');
INSERT INTO package_tracking(65, TIMESTAMP'2016-10-17 08:05:00',
'picked up from customer');
INSERT INTO package_tracking(65, TIMESTAMP'2016-10-17 09:10:00',
'arrived at receiving station');
INSERT INTO package_tracking(65, TIMESTAMP'2016-10-17 10:00:00',
'in transit to destination');
INSERT INTO package_tracking(65, TIMESTAMP'2016-10-17 16:30:00',
'in transit to customer');
INSERT INTO package_tracking(65, TIMESTAMP'2016-10-17 17:00:00',
'delivered to customer');

```

To find the results for the tables:

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, ParcelNumber, DELTA_T(
    (WHERE Status LIKE 'picked%up%customer'),
    (WHERE Status LIKE 'delivered%customer'))
FROM package_tracking_pti
GROUP BY TIME(* AND ParcelNumber);

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, ParcelNumber, DELTA_T(
    (WHERE Status LIKE 'picked%up%customer'),
    (WHERE Status LIKE 'delivered%customer'))
FROM package_tracking
GROUP BY TIME(* AND ParcelNumber) USING TIMECODE(CLOCK_TIME);

```

The results are the same for both tables:

TIMECODE_RANGE	ParcelNumber	DELTA_T(TD_TIMECODE)
('2016-10-16 08:00:00.000000+00:00', '2016-10-17 17:00:00.000000+00:00')	65	('2016-10-16 08:00:00.000000', '2016-10-16 17:00:00.000000')

TIMECODE_RANGE	ParcelNumber	DELTA_T(TD_TIMECODE)
('2016-10-16 08:00:00.000000+00:00', '2016-10-17 17:00:00.000000+00:00')	65	('2016-10-17 08:05:00.000000', '2016-10-17 17:00:00.000000')

Example: Tracking Shipping Days and Percentage of Delivery Failures

In this example, the marketing team starts a campaign to guarantee package delivery within a specified number of days. If the package is not delivered within that date range, the company pays for shipping.

To minimize shipping charges, the company analyzes recent deliveries, determining how many packages missed the delivery dates. The analysis uses DELTA_T with HAVING clause filtration.

```

/*PTI Table*/
DELETE package_tracking_pti;
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 08:00:00', 65, 'picked up
from customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 08:05:00', 69, 'picked up
from customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 09:10:00', 65, 'arrived
at receiving station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 09:20:00', 70, 'picked up
from customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 10:00:00', 65, 'in
transit to destination');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 10:45:00', 70, 'arrived
at receiving station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 16:10:00', 65, 'arrived
at destination station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 16:30:00', 65, 'in
transit to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-16 17:00:00', 65, 'delivered
to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-17 10:00:00', 70, 'in
transit to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-17 10:45:00', 69, 'arrived
at receiving station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-17 11:45:00', 69, 'in
transit to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-17 14:00:00', 69, 'delivered
to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-17 15:00:00', 70, 'delivered
to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-17 08:00:00', 71, 'picked up
from customer');

```

```

INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-18 09:10:00', 71, 'arrived
at receiving station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-20 16:30:00', 71, 'in
transit to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-20 17:00:00', 71, 'delivered
to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-21 08:00:00', 72, 'picked up
from customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-25 09:10:00', 72, 'arrived
at receiving station');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-27 16:30:00', 72, 'in
transit to customer');
INSERT INTO package_tracking_pti(TIMESTAMP'2016-10-27 17:00:00', 72, 'delivered
to customer');

```

To find how many packages took more than two days to deliver, use this example:

```

/*PTI Table*/
SELECT COUNT(*) AS FailedDeliveries FROM
(
    SELECT DELTA_T((WHERE Status LIKE 'picked%up%customer'),
                    (WHERE Status LIKE 'delivered%customer')) AS deltaT
    FROM package_tracking_pti
    GROUP BY TIME(* AND ParcelNumber)
    HAVING INTERVAL(deltaT) DAY TO SECOND > INTERVAL '2 00:00:00' DAY TO SECOND
) as t1;

```

Result:

FailedDeliveries
2

To find how many packages took more than five days to deliver, use this example:

```

/*PTI Table*/
SELECT COUNT(*) AS FailedDeliveries FROM
(
    SELECT DELTA_T((WHERE Status LIKE 'picked%up%customer'),
                    (WHERE Status LIKE 'delivered%customer')) AS deltaT
    FROM package_tracking_pti
    GROUP BY TIME(* AND ParcelNumber)
    HAVING INTERVAL(deltaT) DAY TO SECOND > INTERVAL '5 00:00:00' DAY TO SECOND
) as t1;

```

Result:

FailedDeliveries
1

Find the percentage of packages taking longer than five days to deliver:

```
/*PTI Table*/
SELECT CAST(SUM(CASE
                WHEN INTERVAL(deltaT) DAY TO SECOND > INTERVAL '5 00:00:00' DAY
                TO SECOND
                THEN 1
                ELSE 0
                END) AS REAL) /
        CAST(COUNT(*) AS REAL) * 100 (FORMAT 'zz.zz%') AS "%FailedDeliveries"
FROM
(
    SELECT DELTA_T((WHERE Status LIKE 'picked%up%customer'),
                    (WHERE Status LIKE 'delivered%customer')) AS deltaT
    FROM package_tracking_pti
    GROUP BY TIME(* AND ParcelNumber)
) as t1;
```

Result:

% FailedDeliveries
20.00%

DESCRIBE

Expands the SELECT list with aggregates of *value_expression* over time. DESCRIBE includes both fully parallel and single-threaded (when using VERBOSE) functions.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

Return Value

The return value is defined by each aggregate function that is actually invoked.

Nulls are not included in the result computation.

DISTINCT is not used with MODE when DESCRIBE VERBOSE is expanded.

The following functions are included in the result set of DESCRIBE:

- MAX
- MIN

- AVERAGE
- STDDEV_SAMP

If using the optional VERBOSE keyword, the list of aggregates also includes:

- MEDIAN
- MODE
- 25th PERCENTILE
- 50th PERCENTILE
- 75th PERCENTILE

Additional aggregates included with VERBOSE are single-threaded, and require longer processing times. PERCENTILE computations use a linear process to determine if the percentile lies between two data points. VERBOSE includes MODE, which returns duplicate rows with different values.

The HAVING clause references any aggregates included as part of the DESCRIBE function.

Usage Notes

DESCRIBE has the following restrictions:

- It can only be used in the SELECT list.
- It cannot be used in comparisons.
- It cannot be aliased.
- It cannot be used as an argument to a function.

value_expression must be a numeric data type.

DESCRIBE Syntax

```
DESCRIBE [VERBOSE] ( [ DISTINCT | ALL ] 'value_expression' )
```

Syntax Elements

VERBOSE

Includes both fully parallel and single-threaded functions.

ALL

All non-null *value_expressions*, including duplicates, in the computation.

DISTINCT

Excludes duplicates specified by *value_expression* from the computation.

value_expression

A literal or column expression for which a set of aggregate functions are computed for the series.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

Examples

Example: Using DESCRIBE to Find the Temperature

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, DESCRIBE(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 2,3,4;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, DESCRIBE(TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 2,3,4;
```

The results are the same for both tables:

TIMECODE_RANGE	GROUP BY TIME (MINUTES (10))	BUOYID	Max (TEMPER-ATURE)	Min (TEMPER-ATURE)	Average (TEMPER-ATURE)	Stddev_Samp (TEMPER-ATURE)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	99	10	55	62.9325
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	100	10	54	63.6396
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	79	70	74	3.9479

TIMECODE_ RANGE	GROUP BY TIME (MINUTES (10))	BUOYID	Max (TEMPER- ATURE)	Min (TEMPER- ATURE)	Average (TEMPER- ATURE)	Stddev_ Samp (TEMPER- ATURE)
'2014-01-06 09: 10:00. 000000+00:00')						
('2014-01-06 10: 00:00. 000000+00:00', '2014-01-06 10: 10:00. 000000+00:00')	13	44	56	43	43	5.7581
('2014-01-06 10: 10:00. 000000+00:00', '2014-01-06 10: 20:00. 000000+00:00')	14	44	43	43	50	?

Example: Using the VERBOSE Option

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
DESCRIBE VERBOSE(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 2,3,4;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
DESCRIBE VERBOSE(TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 2,3,4;

```

The results are the same for both tables:

TIMECODE_ RANGE	GROUP BY TIME (MINUTES (10))	BUOY- ID	Max (TEM- PERA- TURE)	Min (TEM- PERA- TURE)	Average (TEM- PERA- TURE)	Stddev_ Samp (TEMPER- ATURE)	Median (TEM- PERA- TURE)	Mode (TEM- PERA- TURE)	Percentile (TEMPER- ATURE, 25)	Percentile (TEMPER- ATURE,5)	Percentile (TEMPER- ATURE, 75)
('2014-01-06 10:10:00. 000000+ 00: 00', '2014-01-06 10:20:00. 000000+ 00: 00')	14	44	43	43	50	?	43	43	43	43	43
('2014-01-06 10:00:00. 000000+ 00: 00', '2014-01-06 10:10:00. 000000+ 00: 00')	13	44	56	43	43	5.7581	53	43	43	53	53.75
('2014-01-06 09:00:00. 000000+ 00: 00', '2014-01-06 09:10:00. 000000+ 00: 00')	7	1	79	70	74	3.9479	74.5	70	71.25	74.5	77.75
('2014-01-06 09:00:00. 000000+ 00: 00', '2014-01-06 09:10:00. 000000+ 00: 00')	7	1	79	70	74	3.9479	74.5	71	71.25	74.5	77.75

8: Time Series Aggregate Functions

TIMECODE_ RANGE	GROUP BY TIME (MINUTES (10))	BUOY- ID	Max (TEM- PERA- TURE)	Min (TEM- PERA- TURE)	Average (TEM- PERA- TURE)	Stddev_ Samp (TEMPER- ATURE)	Median (TEM- PERA- TURE)	Mode (TEM- PERA- TURE)	Percentile (TEMPER- ATURE, 25)	Percentile (TEMPER- ATURE,5)	Percentile (TEMPER- ATURE, 75)
('2014-01-06 09:00:00. 000000+ 00: 00', '2014-01-06 09:10:00. 000000+ 00: 00')	7	1	79	70	74	3.9479	74.5	72	71.25	74.5	77.75
('2014-01-06 09:00:00. 000000+ 00: 00', '2014-01-06 09:10:00. 000000+ 00: 00')	7	1	79	70	74	3.9479	74.5	77	71.25	74.5	77.75
('2014-01-06 09:00:00. 000000+ 00: 00', '2014-01-06 09:10:00. 000000+ 00: 00')	7	1	79	70	74	3.9479	74.5	78	71.25	74.5	77.75
('2014-01-06 09:00:00. 000000+ 00: 00', '2014-01-06 09:10:00. 000000+ 00: 00')	7	1	79	70	74	3.9479	74.5	79	71.25	74.5	77.75

8: Time Series Aggregate Functions

TIMECODE_ RANGE	GROUP BY TIME (MINUTES (10))	BUOY- ID	Max (TEM- PERA- TURE)	Min (TEM- PERA- TURE)	Average (TEM- PERA- TURE)	Stddev_ Samp (TEMPER- ATURE)	Median (TEM- PERA- TURE)	Mode (TEM- PERA- TURE)	Percentile (TEMPER- ATURE, 25)	Percentile (TEMPER- ATURE,5)	Percentile (TEMPER- ATURE, 75)
('2014-01-06 08:10:00. 000000+ 00: 00', '2014-01-06 08:20:00. 000000+ 00: 00')	2	0	100	10	54	63.6396	55	10	32.5	55	77.5
('2014-01-06 08:10:00. 000000+ 00: 00', '2014-01-06 08:20:00. 000000+ 00: 00')	2	0	100	10	54	63.6396	55	100	32.5	55	77.5
('2014-01-06 08:00:00. 000000+ 00: 00', '2014-01-06 08:10:00. 000000+ 00: 00')	1	0	99	10	55	62.9325	54.5	10	32.25	54.5	76.75
('2014-01-06 08:00:00. 000000+ 00: 00', '2014-01-06 08:10:00. 000000+ 00: 00')	1	0	99	10	55	62.9325	54.5	99	32.25	54.5	76.75

Example: Filtering Out Rows with NULL Values for STDDEV_SAMP Using the HAVING Clause

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, DESCRIBE(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
HAVING STDDEV_SAMP(TEMPERATURE) IS NOT NULL
ORDER BY 2,3,4;

```

```

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, DESCRIBE(TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
HAVING STDDEV_SAMP(TEMPERATURE) IS NOT NULL
ORDER BY 2,3,4;

```

The results are the same for both tables:

TIMECODE_RANGE	GROUP BY TIME(MINUTES (10))	BUOYID	Max (TEMPERATURE)	Min (TEMPERATURE)	Average (TEMPERATURE)	Stddev_Samp (TEMPERATURE)
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	79	70	74	4
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	56	43	43	6
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	99	10	55	63

TIMECODE_ RANGE	GROUP BY TIME(MINUTES (10))	BUOYID	Max (TEMPER- ATURE)	Min (TEMPER- ATURE)	Average (TEMPER- ATURE)	Stddev_ Samp (TEMPER- ATURE)
00', '2014-01-06 08:10:00. 000000+00: 00')						
('2014-01-06 08:10:00. 000000+00: 00', '2014-01-06 08:20:00. 000000+00: 00')	2	0	100	10	54	64

FIRST

Returns the oldest value, determined by the timecode, for each group. FIRST is a single-threaded function.

To invoke the FIRST function, use the GROUP BY TIME clause.

Return Values

The return value is the same data type as the input data type.

Nulls are not included in the result computation.

Usage Notes

FIRST is only valid on numeric data.

In the event of a tie, such as simultaneous timecode values for a particular group, all tied results are returned.

If a sequence number is present with the data, it can break a tie, assuming it is unique across identical timecode values.

FIRST Syntax

```
FIRST ( 'value_expression' )
```

Syntax Elements

value_expression

A literal or column expression to evaluate.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

Example: Using FIRST to Return the Temperature

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, FIRST(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 2,3,4;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, FIRST(TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 2,3,4;
```

The result set is the same for both examples. The ties are returned in Rows 3 and 4, below.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	First (TEMPERATURE)
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	43
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	70
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	77
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	10
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	10

KURTOSIS

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

For more information about this function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Examples: Using KURTOSIS with Time Series

Example: Calculating the Kurtosis for PTI and non-PTI Tables

The following example shows how the Kurtosis function can be used with both Primary Time Index (PTI) and non-PTI tables.

The examples use the tables and data setup in [Table and Data Definition for Time Series Aggregates Examples](#).

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, KURTOSIS(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 3,2,1;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, KURTOSIS(TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 3,2,1;
```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Kurtosis (TEMPERATURE)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	?
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	?
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	-2. 75837669094693E 000

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Kurtosis (TEMPERATURE)
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	-2. 17645648488608E 000
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	?

Example: Using the HAVING Clause to Filter Out NULL Results for the Kurtosis Function

The following example shows how the Kurtosis function can be used with the HAVING clause to filter the results.

The examples use the tables and data setup in [Table and Data Definition for Time Series Aggregates Examples](#).

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, KURTOSIS(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
HAVING KURTOSIS(TEMPERATURE) IS NOT NULL
ORDER BY 3,2,1;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, KURTOSIS(TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
HAVING KURTOSIS(TEMPERATURE) IS NOT NULL
ORDER BY 3,2,1;

```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Kurtosis (TEMPERATURE)
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	-2.75837669094693E 000
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	-2.17645648488608E 000

LAST

Returns the newest value, determined by the timecode, for each group. LAST is a single-threaded function.

To invoke the LAST function, use the GROUP BY TIME clause.

Return Values

The return value is the same data type as the input data type.

Nulls are not included in the result computation.

Usage Notes

LAST is only valid on numeric data.

In the event of a tie, such as simultaneous timecode values for a particular group, all tied results are returned. If a sequence number is present with the data, it can break a tie, assuming it is unique across identical timecode values.

LAST Syntax

```
LAST ('value_expression')
```

Syntax Elements

value_expression

A literal or column expression to evaluate.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

Example: Using LAST with Time Series

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, LAST(TEMPERATURE)
```

```

FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 2,3,4;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, LAST(TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 2,3,4;

```

The ties are returned in Rows 5-7, below.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Last (TEMPERATURE)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	99
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	10
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	100
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	72
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	79
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	53
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	56
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	43

MAXIMUM

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

For more information about this function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Examples: Using MAXIMUM with Time Series

Example: Calculating the Maximum for PTI and non-PTI Tables

The following example shows how the MAX function can be used with both Primary Time Index (PTI) and non-PTI tables.

The examples use the tables and data setup in [Table and Data Definition for Time Series Aggregates Examples](#).

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, MAX(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 3,2,1;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, MAX(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 3,2,1;
```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	MAX (TEMPERATURE)	COUNT(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	99	3
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	100	2

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	MAX (TEMPERATURE)	COUNT(*)
('2014-01-06 09:00:00. 000000+00:00', '2014-01-06 09: 10:00.000000+00:00')	7	1	79	6
('2014-01-06 10:00:00. 000000+00:00', '2014-01-06 10: 10:00.000000+00:00')	13	44	56	10
('2014-01-06 10:10:00. 000000+00:00', '2014-01-06 10: 20:00.000000+00:00')	14	44	43	1

Example: Using the HAVING Clause to Filter the MAX Function Results

The following example shows how the MAX function can be used with the HAVING clause to filter the results. In this case, the example filters out the rows which compute their result on less than three values.

The examples use the tables and data setup in [Table and Data Definition for Time Series Aggregates Examples](#).

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, MAX(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
HAVING COUNT(*) >= 3
ORDER BY 3,2,1;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, MAX(TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
HAVING COUNT(*) >= 3
ORDER BY 3,2,1;

```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	MAX (TEMPERATURE)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	99
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	79
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	56

Median Absolute Deviation (MAD)

Returns the median of the set of values defined as the absolute value of the difference between each value and the median of all values in each group.

MAD is a single-threaded function.

To invoke this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

Return Value

This function returns the REAL data type.

Nulls are not included in the result computation.

Usage Notes

The formula for computing MAD:

$$MAD = b \cdot M_i (|x_i - M_j(x_j)|)$$

b = some constant, default 1.4826

$M_j(x_j)$ = MEDIAN of the original set of values

x_i = the original set of values

M_i = MEDIAN of absolute value of difference between each value in x_i and the MEDIAN computed in $M_j(x_j)$

If the *constant_multiplier* is not provided, it uses 1.4826 as the default.

MAD is valid only for numeric data.

Median Absolute Deviation (MAD) Syntax

```
MAD ( [ 'constant_multiplier', ] 'value_expression' )
```

Syntax Elements

constant_multiplier

A Teradata literal numeric constant.

value_expression

A literal or column expression for which to compute a median absolute deviation.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

Examples

Example: Finding the Median Absolute Deviation

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, MAD(1,
TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(30) AND BUOYID)
ORDER BY 3,2,1;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, MAD(1,
TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(30) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 3,2,1;
```

The results are the same for both tables:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Mad(1, TEMPERATURE)	Count(*)
('2014-01-06 08:00:00. 000000+00:00', '2014-01-06 08:30:00.000000+00:00')	1	0	4. 4500000000000000E 001	5

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Mad(1, TEMPERATURE)	Count(*)
('2014-01-06 09:00:00. 000000+00:00', '2014-01-06 09:30:00.000000+00:00')	3	1	3. 500000000000000E 000	6
('2014-01-06 10:00:00. 000000+00:00', '2014-01-06 10:30:00.000000+00:00')	5	44	3. 000000000000000E 000	11

Example: Reporting Groups of Two or More with An Odd Number of Values

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, MAD(1,
TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(30) AND BUOYID)
HAVING COUNT(*) MOD 2 = 1 AND COUNT(*) > 2
ORDER BY 3,2,1;

```

```

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, MAD(1,
TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(30) AND BUOYID)
USING TIMECODE(TIMECODE)
HAVING COUNT(*) MOD 2 = 1 AND COUNT(*) > 2
ORDER BY 3,2,1;

```

The results are the same for both tables:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Mad(1, TEMPERATURE)	Count(*)
('2014-01-06 08:00:00. 000000+00:00', '2014-01-06 08:30:00.000000+00:00')	1	0	4. 450000000000000E 001	50
('2014-01-06 10:00:00. 000000+00:00', '2014-01-06 10:30:00.000000+00:00')	5	44	3. 000000000000000E 000	11

Example: Using the Default Constant Multiplier

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, MAD(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(30) AND BUOYID)
ORDER BY 3,2,1;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, MAD(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(30) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 3,2,1;

```

The results are the same for both tables:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Mad(1. 4826, TEMPERATURE)	Count(*)
('2014-01-06 08:00:00. 000000+00:00', '2014-01-06 08:30:00.000000+00:00')	1	0	6. 597570000000000E 001	5
('2014-01-06 09:00:00. 000000+00:00', '2014-01-06 09:30:00.000000+00:00')	3	1	5. 189100000000000E 000	6
('2014-01-06 10:00:00. 000000+00:00', '2014-01-06 10:30:00.000000+00:00')	5	44	4. 447800000000000E 000	11

MEDIAN

Returns the median of all values in each group. MEDIAN returns the average of the two middle values if the *value_expression* contains an even number of values. It is a single-threaded function.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

Return Values

This function returns the REAL data type.

Nulls are not included in the result computation.

Usage Notes

The ALL keyword includes all non-null *value_expressions*, including duplicates in the computation. The DISTINCT keyword excludes duplicates specified by *value_expression* from the computation.

- MEDIAN is valid only for numeric data.
- DISTINCT is only valid with MEDIAN when using the time series version of MEDIAN. MEDIAN DISTINCT queries return an error if not associated with a GROUP BY TIME clause.

MEDIAN Syntax

```
MEDIAN ( [ DISTINCT | ALL ] 'value_expression' )
```

Syntax Elements

ALL

All non-null *value_expressions*, including duplicates.

DISTINCT

Excludes duplicates specified by *value_expression*.

value_expression

A literal or column expression for which a median is to be computed.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

Examples

Example: Using the MEDIAN Function

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
MEDIAN(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(30) AND BUOYID)
ORDER BY 3,2,1;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
```

```

MEDIAN(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(30) AND BUOYID)
USING TIMECODE(TIMECODE)

```

The results are the same for both tables:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Median (TEMPERATURE)	Count(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:30:00.000000+00:00')	1	0	5.450000000000000E 001	5
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:30:00.000000+00:00')	3	1	7.450000000000000E 001	6
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:30:00.000000+00:00')	5	44	5.300000000000000E 001	11

Example: Reporting Groups with an Odd Number of Values Using the HAVING Clause

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
MEDIAN(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(30) AND BUOYID)
HAVING COUNT(*) MOD 2 = 1 AND COUNT(*) > 2
ORDER BY 3,2,1;

```

```

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
MEDIAN(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(30) AND BUOYID)
USING TIMECODE(TIMECODE)

```

```
HAVING COUNT(*) MOD 2 = 1 AND COUNT(*) > 2
ORDER BY 3,2,1;
```

The results are the same for both tables:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Median (TEMPERATURE)	Count(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:30:00.000000+00:00')	1	0	5.450000000000000E 001	5
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:30:00.000000+00:00')	5	44	5.300000000000000E 001	11

MINIMUM

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

For more information about this function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Example: Using MINIMUM with Time Series

Example: Calculating the Minimum for PTI and non-PTI Tables

The following example shows how the MIN function can be used with both Primary Time Index (PTI) and non-PTI tables.

The examples use the tables and data setup in [Table and Data Definition for Time Series Aggregates Examples](#).

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, MIN(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 3,2,1;
```

```
/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, MIN(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
```

```
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 3,2,1;
```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	MIN (TEMPERATURE)	COUNT(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	10	3
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	10	2
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	70	6
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	43	10
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	43	1

MODE

Returns the mode of all values in each group. In the event of a tie between two or more values from *value_expression*, a row per result is returned. MODE is a single-threaded function.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

Return Values

The return value is the same data type as the input.

Nulls are not included in the result computation.

Usage Notes

MODE is valid only for numeric data.

MODE Syntax

```
MODE ( 'value_expression' )
```

Syntax Elements

value_expression
A literal or column expression for which a median is to be computed.
The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

Example: Using MODE with Time Series

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
MODE(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 3,2,4;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
MODE(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 3,2,4;
```

The results are the same for both tables:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Mode (TEMPERATURE)	Count(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	10	3

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Mode (TEMPERATURE)	Count(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	99	3
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	10	2
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	100	2
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:30:00.000000+00:00')	7	1	70	6
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:30:00.000000+00:00')	7	1	71	6
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:30:00.000000+00:00')	7	1	72	6
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:30:00.000000+00:00')	7	1	77	6
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:30:00.000000+00:00')	7	1	78	6
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:30:00.000000+00:00')	7	1	79	6
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	43	10
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	43	1

PERCENTILE

Returns the value which represents the desired percentile from each group. PERCENTILE is a single-threaded function.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

Return Values

This function returns the REAL data type.

Nulls are not included in the result computation.

The result value is determined by the desired index (di) in an ordered list of values. The following equation is for the di :

$$di = (\text{number of values in group} - 1) * \text{percentile}/100$$

When di is a whole number, that value is the returned result. The di can also be between two data points, i and j , where $i < j$. In this case, the result is interpolated according to one of the following schemes:

- Linear interpolation. The result value is computed using the following equation:

$$\text{result} = i + (j - i) * (di/100) \text{MOD } 1$$

Specify by using the optional interpolation parameter with a value of LINEAR. LINEAR is the default scheme for interpolation.

- Low value. The result value is equal to i . Specify by using the optional interpolation parameter with a value of LOW.
- High value. the result value is equal to j . Specify by using the optional interpolation parameter with a value of HIGH.
- Nearest value. The result value is i if $(di/100) \text{MOD } 1 \leq .5$; otherwise, it is j . Specify by using the optional interpolation parameter with a value of NEAREST.
- Midpoint. The result value is equal to $(i+j)/2$. Specify by using the optional interpolation parameter with a value of MIDPOINT.

Usage Notes

PERCENTILE is valid only for numeric data.

PERCENTILE Syntax

```
PERCENTILE (
  [ DISTINCT | ALL ] 'value_expression',
  percentile
  [, { LINEAR | LOW | HIGH | NEAREST | MIDPOINT } ]
)
```

Syntax Elements

ALL

All non-null *value_expressions*, including duplicates, in the computation.

DISTINCT

Excludes duplicates specified by *value_expression* from the computation.

value_expression

A literal or column expression for which a set of aggregate functions are computed for the series.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

percentile

A Teradata literal float value that represents the desired percentile.

LINEAR

Use linear interpolation to interpolate the result value when the desired result lies between two data points.

This parameter is not case-sensitive.

LOW

Use a low value interpolation scheme to interpolate the result value when the desired result lies between two data points.

This parameter is not case-sensitive.

HIGH

Use a high value interpolation scheme to interpolate the result value when the desired result lies between two data points.

This parameter is not case-sensitive.

NEAREST

Use a nearest value interpolation scheme to interpolate the result value when the desired result lies between two data points.

This parameter is not case-sensitive.

MIDPOINT

Use a midpoint interpolation scheme to interpolate the result value when the desired result lies between two data points.

This parameter is not case-sensitive.

Examples

Example: Using PERCENTILE

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, PERCENTILE(TEMPERATURE,
25), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 3,2,1;
```

```
/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, PERCENTILE(TEMPERATURE,
25), COUNT(*)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
```

The results are the same for both tables:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Percentile (TEMPERATURE)	Count(*)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	10	3
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	10	2
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	72	6
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	43	10
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	43	1

Example: Using the LOW Interpolation Scheme

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, PERCENTILE(TEMPERATURE,
25, LOW), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 3,2,1;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, PERCENTILE(TEMPERATURE,
25, LOW), COUNT(*)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 3,2,1;

```

The results are the same for both tables:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Percentile (TEMPERATURE)	Count(*)
('2014-01-06 08:00:00. 000000+00:00', '2014-01-06 08: 10:00.000000+00:00')	1	0	10	3
('2014-01-06 08:10:00. 000000+00:00', '2014-01-06 08: 20:00.000000+00:00')	2	0	10	2
('2014-01-06 09:00:00. 000000+00:00', '2014-01-06 09: 10:00.000000+00:00')	7	1	72	6
('2014-01-06 10:00:00. 000000+00:00', '2014-01-06 10: 10:00.000000+00:00')	13	44	44	43
('2014-01-06 10:10:00. 000000+00:00', '2014-01-06 10: 20:00.000000+00:00')	14	44	43	1

RANK (ANSI)

You can use the RANK (ANSI) function in conjunction with the GROUP BY TIME clause.

For more information about this function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

The following shows an example of how to rank time series aggregated data.

Example: Using RANK(ANSI) with Time Series Data

The following example shows how to use RANK(ANSI) to rank the averages of particular timebuckets across an entire time series.

The examples use the tables and data setup in [Table and Data Definition for Time Series Aggregates Examples](#).

Perform the AVG before using RANK(ANSI). The AVG function can be used with both Primary Time Index (PTI) and non-PTI tables. The following examples get the averages from the time series data.

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00' AND BUOYID=0
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 1,2,3;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, AVG(TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 1,2,3;
```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	AVG (TEMPERATURE)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	54

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	AVG (TEMPERATURE)
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	55
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	74
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	14	50
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	14	43

Use RANK(ANSI) to rank the averages for each individual time series:

```

/*PTI Table*/
SEL TIMECODE_RANGE, TBN, BUOYID, AVG_TEMPERATURE, RANK() OVER (PARTITION BY
BUOYID ORDER BY AVG_TEMPERATURE) FROM (
    SELECT $TD_TIMECODE_RANGE as TIMECODE_RANGE, $TD_GROUP_BY_TIME as TBN,
    BUOYID, AVG(TEMPERATURE) AS AVG_TEMPERATURE
    FROM OCEAN_BUOYS
    WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
    GROUP BY TIME (MINUTES(10) AND BUOYID)
) AS NESTED_AVG_TABLE
ORDER BY 3,5;

/*Non-PTI Table*/
SEL TIMECODE_RANGE, TBN, BUOYID, AVG_TEMPERATURE, RANK() OVER (PARTITION BY
BUOYID ORDER BY AVG_TEMPERATURE) FROM (
    SELECT $TD_TIMECODE_RANGE as TIMECODE_RANGE, $TD_GROUP_BY_TIME as TBN,
    BUOYID, AVG(TEMPERATURE) AS AVG_TEMPERATURE
    FROM OCEAN_BUOYS_NONPTI
    WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
    GROUP BY TIME (MINUTES(10) AND BUOYID)
    USING TIMECODE(TIMECODE)
) AS NESTED_AVG_TABLE
ORDER BY 3,5;

```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	TBN	BUOYID	AVG (TEMPERATURE)	RANK
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	54	1
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	55	2
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	74	1
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	13	44	43	1
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	50	2

SKEW

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

For more information about this function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Example: Using SKEW with Time Series

For a PTI table:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, SKEW(TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 3,2,1;
```

For a non-PTI table:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, SKEW(TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 3,2,1;
```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Skew(TEMPERATURE)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	?
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	?
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	0.000000000000000E 000
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	-3.83596236981183E-001
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	?

STDDEV_POP

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

For more information about this function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Example: Using STDDEV_POP with Time Series

For a PTI table:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
STDDEV_POP(CAST(TEMPERATURE AS REAL))
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 3,2,1;
```

For a non- PTI table:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
STDDEV_POP(CAST(TEMPERATURE AS REAL))
```

```

FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10)) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 3,2,1;

```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	STDDEV_ POP (TEMPERATURE)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	4.450000000000000E 001
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	4.500000000000000E 001
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	3.59397644214130E 000
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	5.46260011349907E 000
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	0.000000000000000E 000

STDDEV_SAMP

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

For more information about this function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Example: Using STDDEV_SAMP with Time Series

For a PTI table:

```

SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
STDDEV_SAMP(CAST(TEMPERATURE AS REAL))
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'

```



```
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 3,2,1;
```

For a non-PTI table:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
STDDEV_SAMP(CAST(TEMPERATURE AS REAL))
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 3,2,1;
```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Stddev_ Samp (TEMPERATURE)
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	?
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	5.75808610178378E 000
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	3.93700393700591E 000
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	6.36396103067893E 001
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	6.29325035256027E 001

Example: Filtering Out NULL Results with the HAVING Clause

For a PTI table:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
STDDEV_SAMP(CAST(TEMPERATURE AS REAL))
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
```

```
HAVING STDDEV_SAMP(TEMPERATURE) IS NOT NULL
ORDER BY 3,2,1;
```

For a non-PTI table:

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
STDDEV_SAMP(CAST(TEMPERATURE AS REAL))
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
HAVING STDDEV_SAMP(TEMPERATURE) IS NOT NULL
ORDER BY 3,2,1;
```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Stddev Samp (TEMPERATURE)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	6.29325035256027E 001
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	6.36396103067893E 001
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	3.93700393700591E 000
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	5.75808610178378E 000

SUM

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

For more information about this function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Example: Using SUM with Time Series

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, SUM(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 3,2,1;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, SUM(TEMPERATURE), COUNT(*)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 3,2,1;

```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID		
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	109	3
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	110	2
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	447	6
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	496	10
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	43	1

TOP

Returns the largest *number_of_values* in the *value_expression* for each group, with or without ties. TOP is a single-threaded function.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

Syntax

```
TOP [ WITH TIES ] ('number_of_values', 'value_expression')
```

Syntax Elements

number_of_values

Teradata integer value representing the number of values to return.

value_expression

A literal or column expression from which the top values are taken.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

Return Value

The return value is the same as the input data type.

Nulls are not included in the computation.

Usage Notes

TOP is valid only for numeric data.

WITH TIES implies the rows returned by the query include the specified number of rows in the ordered set for each time bucket, plus any additional rows where the sort key value is the same as the value of the sort key in the last row satisfying the specified number or percentage of rows. If this clause is omitted and ties are found, the earliest value in terms of timecode is returned.

Examples

Example: Using TOP Without TIES

```
/*PTI Table*/SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID,
TOP(2, TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 10:00:00' AND TIMESTAMP
'2014-01-06 10:06:00' AND BUOYID=44
```

```

GROUP BY TIME (MINUTES(2) AND BUOYID) FILL(NULLS)
ORDER BY 1, 4, 3;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, TOP(2, TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 10:00:00' AND TIMESTAMP
'2014-01-06 10:06:00' AND BUOYID=44
GROUP BY TIME (MINUTES(2) AND BUOYID)
USING TIMECODE(TIMECODE)
FILL(NULLS)
ORDER BY 1, 4, 3;

```

The results are the same for both tables:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	TEMPERATURE
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:02:00.000000+00:00')	1	44	?
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:02:00.000000+00:00')	1	44	?
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:02:00.000000+00:00')	1	44	53
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:02:00.000000+00:00')	1	44	54
('2014-01-06 10:02:00.000000+00:00', '2014-01-06 10:04:00.000000+00:00')	2	44	?
('2014-01-06 10:02:00.000000+00:00', '2014-01-06 10:04:00.000000+00:00')	2	44	?
('2014-01-06 10:02:00.000000+00:00', '2014-01-06 10:04:00.000000+00:00')	2	44	55
('2014-01-06 10:02:00.000000+00:00', '2014-01-06 10:04:00.000000+00:00')	2	44	56
('2014-01-06 10:04:00.000000+00:00', '2014-01-06 10:06:00.000000+00:00')	3	44	?
('2014-01-06 10:04:00.000000+00:00', '2014-01-06 10:06:00.000000+00:00')	3	44	?
('2014-01-06 10:04:00.000000+00:00', '2014-01-06 10:06:00.000000+00:00')	3	44	?
('2014-01-06 10:04:00.000000+00:00', '2014-01-06 10:06:00.000000+00:00')	3	44	?

Example: Using TOP With TIES

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, TOP WITH
TIES(2, TEMPERATURE)
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 10:00:00' AND TIMESTAMP
'2014-01-06 10:06:00' AND BUOYID=44
GROUP BY TIME (MINUTES(2) AND BUOYID) FILL(NULLS)
ORDER BY 1, 4, 3;

```

```

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, TOP WITH
TIES(2, TEMPERATURE)
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 10:00:00' AND TIMESTAMP
'2014-01-06 10:06:00' AND BUOYID=44
GROUP BY TIME (MINUTES(2) AND BUOYID)
USING TIMECODE(TIMECODE)
FILL(NULLS)
ORDER BY 1, 4, 3;

```

The results are the same for both tables:

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	TEMPERATURE
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:02:00.000000+00:00')	1	44	53
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:02:00.000000+00:00')	1	44	54
('2014-01-06 10:02:00.000000+00:00', '2014-01-06 10:04:00.000000+00:00')	2	44	55
('2014-01-06 10:02:00.000000+00:00', '2014-01-06 10:04:00.000000+00:00')	2	44	56
('2014-01-06 10:04:00.000000+00:00', '2014-01-06 10:06:00.000000+00:00')	3	44	?
('2014-01-06 10:04:00.000000+00:00', '2014-01-06 10:06:00.000000+00:00')	3	44	?

VAR_POP

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

For more information about this function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Example: Using VAR_POP with Time Series

```
/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, VAR_POP(CAST(TEMPERATURE
AS REAL))
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 3,2,1;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, VAR_POP(CAST(TEMPERATURE
AS REAL))
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 3,2,1;
```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Var_ Pop (TEMPERATURE)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	1.980250000000000E 003
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	2.025000000000000E 003
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	1.291666666666667E 001

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Var_ Pop (TEMPERATURE)
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	2.984000000000000E 001
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	0.000000000000000E 000

VAR_SAMP

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see [GROUP BY TIME Clause](#).

For more information about this function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Example: Using VAR_SAMP with Time Series

```

/*PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, VAR_SAMP(CAST(TEMPERATURE
AS REAL))
FROM OCEAN_BUOYS
WHERE TD_TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
ORDER BY 3,2,1;

/*Non-PTI Table*/
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, BUOYID, VAR_SAMP(CAST(TEMPERATURE
AS REAL))
FROM OCEAN_BUOYS_NONPTI
WHERE TIMECODE BETWEEN TIMESTAMP '2014-01-06 08:00:00' AND TIMESTAMP
'2014-01-06 10:30:00'
GROUP BY TIME (MINUTES(10) AND BUOYID)
USING TIMECODE(TIMECODE)
ORDER BY 3,2,1;

```

Result:

Note:

The results of both queries are identical so only one result set is shown.

TIMECODE_RANGE	GROUP BY TIME (MINUTES(10))	BUOYID	Var_ Pop (TEMPERATURE)
('2014-01-06 08:00:00.000000+00:00', '2014-01-06 08:10:00.000000+00:00')	1	0	3.960500000000000E 003
('2014-01-06 08:10:00.000000+00:00', '2014-01-06 08:20:00.000000+00:00')	2	0	4.050000000000000E 003
('2014-01-06 09:00:00.000000+00:00', '2014-01-06 09:10:00.000000+00:00')	7	1	1.550000000000000E 001
('2014-01-06 10:00:00.000000+00:00', '2014-01-06 10:10:00.000000+00:00')	13	44	3.315555555555556E 001
('2014-01-06 10:10:00.000000+00:00', '2014-01-06 10:20:00.000000+00:00')	14	44	?

Tools and Utilities for PTI Tables

Most Teradata tools and utilities support PTI tables as they support non-PTI tables, with the following exceptions:

Load and Unload Utilities

- Teradata MultiLoad cannot be used to load PTI tables.
- Teradata Parallel Transporter, Teradata FastLoad, Teradata FastExport, and Teradata TPump support loading data to, and unloading data from PTI tables.
 - To load data using an INSERT statement, follow the same rules used for MERGE WHEN NOT MATCH INSERT (see [MERGE](#)).
 - For a single row update/insert on a PTI table, follow the [UPDATE \(Upsert Form\)](#) guidelines.

Note:

The TD_TIMEBUCKET column is automatically generated and managed by Vantage as necessary for PTI tables. It is invisible to users and DML. Therefore, never attempt to load or insert data that includes values for the TD_TIMEBUCKET column of a PTI table.

Archive and Restore

You can use DSA to archive, restore, and copy PTI tables. If a PTI table is copied or restored from one system to another, use ALTER TABLE REVALIDATE to revalidate the PTI tables after restoring.

Notation Conventions

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 [<i>delimiter</i> ...]	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

Character Shorthand Notation Used in This Document

This document uses the Unicode naming convention for characters. For example, the lowercase character 'a' is more formally specified as either LATIN CAPITAL LETTER A or U+0041. The U+xxxx notation refers to a particular code point in the Unicode standard, where xxxx stands for the hexadecimal representation of the 16-bit value defined in the standard.

In parts of the document, it is convenient to use a symbol to represent a special character, or a particular class of characters. This is particularly true in discussion of the following Japanese character encodings:

- KanjiEBCDIC
- KanjiEUC
- KanjiShift-JIS

These encodings are further defined in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Character Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

Symbol	Encoding	Meaning
a-z A-Z 0-9	Any	Any single byte Latin letter or digit.
<u>a-z</u> <u>A-Z</u> <u>0-9</u>	Any	Any fullwidth Latin letter or digit.

Symbol	Encoding	Meaning
<	KanjiEBCDIC	Shift Out [SO] (0x0E). Indicates transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEBCDIC	Shift In [SI] (0x0F). Indicates transition from multibyte to single byte KanjiEBCDIC.
T	Any	Any multibyte character. The encoding depends on the current character set. For KanjiEUC, code set 3 characters are always preceded by <code>ss3</code> .
!	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by <code>ss2</code> , forming an individual multibyte character.
<u>Δ</u>	Any	Represents the graphic pad character.
Δ	Any	Represents a single or multibyte pad character, depending on context.
ss 2	KanjiEUC	Represents the EUC code set 2 introducer (0x8E).
ss 3	KanjiEUC	Represents the EUC code set 3 introducer (0x8F).

For example, string “TEST”, where each letter is intended to be a fullwidth character, is written as **TEST**. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set:

LMN<TEST>QRS

is represented as:

D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2

Pad Characters

The following table lists the pad characters for the various character data types.

Server Character Set	Pad Character Name	Pad Character Value
LATIN	SPACE	0x20
UNICODE	SPACE	U+0020
GRAPHIC	IDEOGRAPHIC SPACE	U+3000
KANJISJIS	ASCII SPACE	0x20
KANJI1	ASCII SPACE	0x20

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