

Teradata Vantage™ - SQL Date and Time Functions and Expressions

Release 17.10

July 2021

Copyright and Trademarks

Copyright © 2019 - 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 Vantage SQL Date and Time Functions and Expressions	6
Changes and Additions	6
Chapter 2: Business Calendars	7
About the Business Calendars	7
Computation	8
About Business Calendar Macros	11
About the Business Calendar Tables	16
About the Business Calendar Views	18
Business Calendar Functions	22
td_week_begin	36
td_week_end	37
td_sunday	38
td_monday	40
td_tuesday	42
td_wednesday	43
td_thursday	45
td_friday	47
td_saturday	48
DayNumber_Of_Week	50
td_month_begin	52
td_month_end	53
DayNumber_Of_Month	55
DayOccurrence_Of_Month	57
WeekNumber_Of_Month	59
td_year_begin	60
td_year_end	62
DayNumber_Of_Year	63
WeekNumber_Of_Year	65
MonthNumber_Of_Year	67
td_quarter_begin	68
td_quarter_end	70
WeekNumber_Of_Quarter	71
MonthNumber_Of_Quarter	73
QuarterNumber_Of_Year	75
DayNumber_Of_Calendar	76
WeekNumber_Of_Calendar	78
MonthNumber_Of_Calendar	79
QuarterNumber_Of_Calendar	81

YearNumber_Of_Calendar	83
Chapter 3: Calendar Functions	85
td_day_of_week/DayOfWeek	85
td_day_of_month	86
td_day_of_year	87
td_weekday_of_month	88
td_week_of_month	90
td_week_of_year	91
td_week_of_calendar	92
td_month_of_quarter	94
td_month_of_year	95
td_month_of_calendar	96
td_quarter_of_year	97
td_quarter_of_calendar	99
td_year_of_calendar	100
Chapter 4: DateTime and Interval Functions and Expressions	102
ANSI DateTime Data Types	102
ANSI DateTime and Interval Data Type Assignment Rules	102
Scalar Operations on ANSI SQL:2011 DateTime and Interval Values	105
ANSI DateTime Expressions	106
ANSI Interval Expressions	116
Arithmetic Operators and ANSI DateTime and Interval Data Types	124
Aggregate Functions and ANSI DateTime and Interval Data Types	126
Scalar Operations and DateTime Functions	127
Teradata Date and Time Expressions	128
Scalar Operations on Teradata DATE Values	129
YEAR/MONTH/DAYOFMONTH/HOUR/MINUTE/SECOND	130
WEEK	131
LAST_DAY	132
NEXT_DAY	134
MONTHS_BETWEEN	137
ADD_MONTHS	139
OADD_MONTHS	145
TRUNC(Date)	147
ROUND(Date)	151
EXTRACT	154
GetTimeZoneDisplacement	158
Chapter 5: Period Functions and Operators	167
Period Value Constructor	167
Arithmetic Operators	169
Comparison of Period Types	171
BEGIN	173

CONTAINS	175
END	178
EQUALS	180
IS UNTIL_CHANGED/IS NOT UNTIL_CHANGED	182
IS UNTIL_CLOSED/IS NOT UNTIL_CLOSED	185
IMMEDIATELY PRECEDES	187
IMMEDIATELY SUCCEEDS	189
INTERVAL	191
LAST	195
MEETS	196
NEXT	199
OVERLAPS	200
P_INTERSECT	209
PRECEDES	211
PRIOR	214
LDIFF	215
RDIFF	217
SUCCEEDS	220
TD_NORMALIZE_OVERLAP	222
TD_NORMALIZE_MEET	225
TD_NORMALIZE_OVERLAP_MEET	227
TD_SUM_NORMALIZE_OVERLAP	229
TD_SUM_NORMALIZE_MEET	232
TD_SUM_NORMALIZE_OVERLAP_MEET	234
TD_SEQUENCED_SUM	237
TD_SEQUENCED_AVG	239
TD_SEQUENCED_COUNT	242
Appendix A: Notation Conventions	245
Appendix B: Additional Information	248

Introduction to Teradata Vantage SQL Date and Time Functions and Expressions

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.

This document describes SQL date and time functions and expressions, including those for calendars, business calendars, DateTime and intervals, and periods.

Changes and Additions

Date	Description
July 2021	Minor edits.

Business Calendars

The following sections describe the system-defined business calendars, the macros you can use to manipulate them, the business calendar tables and views, and the embedded services system functions that support the system-defined business calendars.

About the Business Calendars

A business calendar defines business and non-business days. The significance of a day being a business day or a non-business day is user-determined. For example, a business day could be a work day, and a non-business day could be either a non-working day, a weekend day, a holiday, or a vacation day. You can define different week patterns (weekdays and weekends) and exceptions (holidays and business open and closed days) for the system-defined calendars.

There are three Teradata system-defined business calendars you can set for your session:

- Teradata
- ISO
- COMPATIBLE

All three calendars are based on the Gregorian calendar. The Gregorian calendar has 365 days in most years and 366 days in a leap year. The calendars differ in how they define weeks and whether they allow partial weeks. You can use macros to specify weekday/weekend patterns and exceptions to the patterns.

The calendars support January 1, 1900, to December 31, 2100. The default session calendar is Teradata. Each calendar defaults to all business days. You can change that pattern using a macro. See [About Business Calendar Macros](#).

Calendar Differences

For This Calendar...	The First Full Week Begins...
Teradata	on Sunday. The days of the year before Sunday belong to Week 0. For example, if the year starts on January 1, 2004 (a Thursday), then Week 0 is from January 1 to January 3. Week 1 begins on Sunday, January 4.
ISO	on Monday. The first week of the year is the first week that has at least 4 days. If a week has fewer than 4 days, it belongs to the last week of the previous year. There are no partial weeks. This calendar follows the ISO and European standard. For example, if the year starts on January 1, 2008 (a Tuesday) and the week start is Monday, week 1 of 2008 is from December 31, 2007, to January 6, 2008.
COMPATIBLE	on January 1. regardless of what day of the week that is.

For This Calendar...	The First Full Week Begins...
	<p>There can be partial weeks with 1 day (for most years) or 2 days (for leap years) at the end of the year. The day the week begins can change from year to year.</p> <p>This calendar is Oracle-compatible. For example, if January 1, 2011, is a Saturday, the first week of the year is from Saturday, January 1, 2011, through Friday, January 7, 2011.</p>

Computation

About ISO Computation

The way in which a week, month, quarter, and year are defined in the ISO calendar is different from the Teradata and COMPATIBLE calendars. ISO has only complete weeks. All the business calendar functions and views use the following rules for ISO computation.

Week

An ISO week always has 7 days. There are no partial weeks. The ISO week always starts on Monday and ends on Sunday.

Year

Each year has 52 or 53 complete weeks. The year start or end is determined by applying the ISO Thursday rule to the week at the year border. If Thursday falls in the old year in the Gregorian calendar, the border week becomes part of the old year in the ISO calendar. If Thursday falls in the new year in the Gregorian calendar, the border week becomes part of the new year in the ISO calendar. For example, December 31, 2009, is a Thursday. Therefore, in the ISO calendar the week of Monday, December 28, 2009, through Sunday, January 3, 2010, is the last week in 2009. The first week of 2010 in the ISO calendar begins on Monday, January 4, 2010. In contrast, the Teradata calendar for 2010 begins on Friday, January 1.

2009 Dec last week ISO

Dec 2009 Calendar						
S	M	T	W	T	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

2010 Jan first week ISO

Jan 2010 Calendar						
S	M	T	W	T	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

Month

Each month has 4 or 5 complete weeks. The month start or end is determined by applying the ISO Thursday rule to the week at the month border. For example, in the ISO calendar, the month of October

2012 ends on Sunday, October 28, because Thursday in the border week falls on November 1. The new week that begins on Monday, October 29, becomes part of November.

2012 Oct last week ISO

Oct 2012 Calendar						
S	M	T	W	Th	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3

2012 Nov first week ISO

Nov 2012 Calendar						
S	M	T	W	Th	F	S
	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

Quarter

The ISO year is divided into 4 quarters with 13 weeks per quarter. There are 14 weeks in the last quarter if the year has 53 weeks. Quarters are divided into 3 months with a pattern of 4,4,5 or 4,5,4 or 5,4,4 weeks. If the year has 53 weeks, the last quarter has 3 months with a pattern of 4,5,5 or 5,4,5 or 5,5,4 weeks. The quarter start or end is determined by applying the ISO Thursday rule to the week at the month border in

the last month of the quarter. The week at the month border belongs to the month that has Thursday of that week. For example, Friday, January 1, 2016, is part of a week in which Thursday falls in the last week and last quarter of 2015. Therefore, Friday, January 1, 2016, belongs to last quarter of 2015 in the ISO calendar and the first quarter of 2016 in the Teradata calendar.

2015 Dec last week of quarter ISO

Dec 2015 Calendar						
S	M	T	W	Th	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3						

2016 Jan first week of quarter ISO

Jan 2016 Calendar						
S	M	T	W	Th	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

About Business Calendar Macros

System business macros enable administrators and users to configure the system business calendars (Teradata, ISO, and COMPATIBLE) by:

- Creating the pattern for the calendar.
- Creating exceptions for the calendar.
- Deleting existing exceptions from the calendar.

These macros are created when the DIP script is executed and reside in the DBC database.

Note:

To view any of the macros, use the DIP script.

Macro Name	Purpose
CreateBusinessCalendarPattern	Creates the pattern for a business calendar.
CreateException	Defines an exception list for a specific business calendar. The list identifies all of the days of the calendar that are exceptions to the pattern defined for the calendar.
DeleteException	Deletes a particular exception from a specific business calendar.
DeleteAllExceptions	Deletes all of the exceptions from a specific business calendar.

CreateBusinessCalendarPattern

You can use this macro to create the pattern for a business calendar. The pattern is the template for business days and non-business days for the entire calendar and the basic structural unit on which the calendar is based (for example, week, month, quarter or year).

Note:

Currently, only the weekly pattern is supported. You can define each day of the week as a business day or non-business day. To make particular business days non-business days (for example, a holiday), you can use the CreateException macro (see [CreateException](#)).

When you execute the macro, the pattern for the calendar is created and the information about the pattern is stored in the DBC.BusinessCalendarPattern table.

CreateBusinessCalendarPattern Required Parameters

Parameter	Data Type	Format	Description
CalendarName	VARCHAR(256) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL	X(128)	Name of the business calendar. For the system business calendars, it must be Teradata, ISO, or COMPATIBLE.
DayName	CHAR(10)	X(10)	Name of the calendar day. Must be SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, or SATURDAY.
Pattern	CHAR(5)	X(5)	Pattern for the calendar day. Must be ON (working day) or OFF (non-working day).
PatternComment	VARCHAR(2048)	X(1024)	Reason or rationale for the pattern for the calendar day (ON or OFF).

Example: Setting Non-Working Days

In the following, Saturday and Sunday of each week of the Teradata business calendar are defined as non-working days (non-working days have the OFF status) and this information is added to the DBC.BusinessCalendarPattern table.

Note:

By default, each day of week for the Teradata business calendar is a working day.

```
EXEC DBC.CreateBusinessCalendarPattern
  ('TERADATA', 'SATURDAY', 'OFF', 'Non-work Day');
EXEC DBC.CreateBusinessCalendarPattern
  ('TERADATA', 'SUNDAY', 'OFF', 'Non-work Day');
```

Example: Designating Working Days

The following modifies the ISO business calendar by designating SUNDAY and MONDAY as working days:

```
EXEC DBC.CreateBusinessCalendarPattern
  ('ISO', 'SUNDAY', 'ON', 'Work Day');
```

```
EXEC DBC.CreateBusinessCalendarPattern
('ISO', 'MONDAY', 'ON', 'Work Day');
```

CreateException

You can use this macro to define an exception list for a specific business calendar. The list identifies all of the days (by date) of the calendar that are exceptions to the pattern defined for the calendar.

When you execute the macro, the exceptions in the list are inserted into the calendar and into the DBC.BusinessCalendarException table.

If you try to define an exception for a particular date that is already an exception, an error is returned. To replace the existing exception, you must first delete the existing exception then redefine a new one for the date.

CreateException Required Parameters

Parameter	Format	Data Type	Description
CalendarName	VARCHAR(256) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL	X(128)	Name of the business calendar. For the system business calendars, it must be Teradata, ISO, or COMPATIBLE.
ExceptionIndicator	CHAR(5)	X(5)	Type of day for the exception. Must be ON (business day) or OFF (non-business day).
ExceptionDate	DATE	YYYY-MM-DD	Date of the exception (not the day it was created).
PatternComment	VARCHAR(2048)	X(1024)	Reason or rationale for the pattern for the calendar day (ON or OFF).

CreateException Usage Notes

If you try to insert exceptions that are beyond the CalendarPeriod boundaries, the insertion is aborted.

If you try to insert an exception for a day that is already defined as an exception, you must delete the existing exception for that day and replace it with the new exception.

Example: Defining OFF and ON Exceptions

In this example, an OFF and an ON exception are defined for the Teradata business calendar and the information about them is added to the DBC.BusinessCalendarException table. Days with the OFF status are non-business days. Days with the ON status are business days.

```
EXEC DBC.CreateException
  ('Teradata', 'OFF', DATE '2008-03-03', 'Holiday');
EXEC DBC.CreateException
  ('Teradata', 'ON', DATE '2008-06-07', 'Holiday make up day');
```

The two exceptions are inserted into the Teradata business calendar. This first exception indicates that March 3, 2008 is now a non-business day (noted in the comment as a holiday). The second exception indicates that June 7, 2008 is now a business day (noted in the comment as a day to make up for a previous holiday).

The CalendarPeriod of the Teradata calendar is from January 1, 1900 to December 31, 2100. If you try to insert holidays beyond the CalendarPeriod boundary, the insertion is aborted.

DeleteException

You can use this macro to delete a particular exception from a specific business calendar. When you delete an exception, the status for that exception day automatically reverts to the status defined for it in the pattern for the calendar.

When you execute the macro, the exception is removed from the calendar and from the DBC.BusinessCalendarException table.

If you need to delete all of the exceptions from a calendar, use the DeleteAllExceptions macro (see [DeleteAllExceptions](#)).

DeleteException Required Parameters

Parameter	Data Type	Format	Description
CalendarName	VARCHAR(256) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL	X(128)	The name of the business calendar. For the system business calendars, it must be Teradata, ISO, or COMPATIBLE.
ExceptionDate	DATE	YY/MM/DD	The date of the exception (not the day it was created).

Example: Deleting Exceptions

In this example, the exception with the date of January 1, 2009 is deleted from the ISO business calendar and from the DBC.BusinessCalendarException table.

```
EXEC DBC.DeleteException('ISO', DATE '2009-01-01');
```

DeleteAllExceptions

You can use this macro to delete all of the exceptions from a specific business calendar. When you delete all exceptions, the status for all exception days automatically reverts to the status defined for the days in the pattern for the calendar.

When you execute the macro, all exceptions are removed from the calendar and from the DBC.BusinessCalendarException table.

If you need to delete a particular exception from a calendar, use the DeleteException macro (see [DeleteException](#)).

DeleteAllExceptions Required Parameter

Parameter	Data Type	Format	Description
CalendarName	VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL	X(128)	The name of the business calendar. For the system business calendars, it must be Teradata, ISO, or COMPATIBLE.

Example: Removing Exceptions from the ISO Business Calendar

In this example, all of the exceptions are removed from the ISO business calendar and from the DBC.BusinessCalendarException table.

```
EXEC DBC.DeleteAllExceptions('ISO');
```

About the Business Calendar Tables

You can query business calendar tables to see calendar information relevant to your business.

DBC.BusinessCalendarPattern Table

This table records calendar information about patterns of business and non-business days.

Column Name	Column Description
CalendarName	Business calendar name.
PatternType	Identifies the pattern as weekly, monthly, quarterly, or yearly. Currently, only the weekly pattern is supported. Pattern "W" indicates a weekly pattern.
DayNumber	The days in the week are numbered from 1 to 7, starting with Sunday as 1 and ending with Saturday as 7. There is one row in the table for each day, and the ON/OFF status for the day is stored in the Pattern column.
Pattern	A weekly pattern describes 7 rows, each set to 1 or 0. For each DayNumber, this column stores the status, whether it is a business day (1) or a non-business day (0).
PatternComment	A description of the pattern.
CreatorName	The name of the user who created or modified the pattern. For system-defined business calendars, this is DBC.
LastModified	The timestamp of the last pattern change.

DBC.BusinessCalendarException Table

This table contains a list of days that are exceptions to the pattern of working and non-working days.

Column Name	Column Description
CalendarName	Business calendar name to which the exception applies.
ExceptionIndicator	A value of 0 indicates that an ExceptionDate is a non-business day, and a value of 1 indicates that an ExceptionDate is a business day.
ExceptionDate	An ExceptionDate must fall within the Teradata calendar (January 1, 1900, to December 31, 2100). If the ExceptionDate is outside that period, it is not accepted.
ExceptionReason	The reason for the exception. For example, December 25, 2012 (a Tuesday) is normally a working day, but it is also the Christmas holiday, so the ExceptionReason is "Christmas." The ExceptionReason may be NULL.
CreatorName	The name of the user who created the exception.
CreationTime	The timestamp of the exception creation.

There are two types of exceptions:

- An OFF exception: A typical working day is a day off (for example, Monday, September 5, 2011, is the Labor Day holiday).
- An ON exception: A typical day off is a working day (for example, everyone at your job is required to work on Saturday, July 30, 2011).

You do not need to validate whether an OFF-exception falls on a business day and an ON-exception falls on a non-business day. Just set the exception, and specify an ExceptionReason. For example, Monday,

July 4, is a holiday, but the Barbeque Bonanza store is open and your ExceptionReason is Independence Day Sale.

About the Business Calendar Views

You can query business calendar views to retrieve calendar information relevant to your business, such as the first business day of a week.

Sys_Calendar.BusinessCalendarExceptions

This view provides information about exceptions defined for a calendar, such as a holiday on Monday, which is normally a working day. You can query the view to check the exceptions.

View Column Name	Description	Data Type	Format
CalendarName	The name of the calendar.	VARCHAR(128)	X(128)
ExceptionIndicator	Returns ON for a working day and OFF for a nonworking day.	VARCHAR(3)	X(3)
ExceptionDate	The exception date set for the calendar.	DATE	YY/MM/DD
ExceptionReason	The reason for the exception.	VARCHAR(1024)	X(1024)
CreatorName	The user who created the exception.	VARCHAR(128)	X(128)
CreationTime	The timestamp of exception creation.	TIMESTAMP(0)	YYYY-MM-DDBHH:MI:SS

Sys_Calendar.BusinessCalendarPatterns

This view provides information about patterns defined for a calendar, such as a weekday/weekend pattern.

View Column Name	Description	Data Type	Format
CalendarName	The name of the calendar.	VARCHAR(128)	X(128)
DayName	The day name, for example, Friday	VARCHAR(9)	X(9)
Pattern	The ON or OFF pattern settings.	VARCHAR(3)	X(3)
PatternComment	The comment for each day in the pattern.	VARCHAR(1024)	X(1024)
CreatorName	The user who created the pattern.	VARCHAR(128)	X(128)
LastModified	The timestamp of last pattern change.	TIMESTAMP(0)	YYYY-MM-DD:HH:MI:SS

Sys_Calendar.BusinessCalendar

This Sys_Calendar view provides business functionality for the three system-defined business calendars.

View Column Name	Description	Data Type	Format
Calendar_Date	The date.	DATE	YY/MM/DD
Day_of_Week	An integer value that ranges from 1 to 7.	INTEGER	-(10)9
Day_of_Month	A integer value that ranges from 1 to 31.	INTEGER	-(10)9
Day_of_Year	An integer value that ranges from 1 to 366.	INTEGER	-(10)9
Day_of_Calendar	The number of days since the beginning of the calendar.	INTEGER	-(10)9
Weekday_of_Month	The <i>n</i> th occurrence of the weekday in the month (1-5).	INTEGER	-(10)9
Week_of_Month	The week number of the month, ranging from 0 to 5.	INTEGER	-(10)9
Week_of_Quarter	The week number of the quarter, ranging from 0 to 14.	INTEGER	-(10)9
Week_of_Year	The week number of the year, ranging from 0 to 53.	INTEGER	-(10)9
Week_of_Calendar	For a given date, the number of the week in the calendar in which it occurs.	INTEGER	-(10)9
Month_of_Quarter	For a given date, the number of the month in the quarter in which it occurs.	INTEGER	-(10)9
Month_of_Year	For a given date, the number of the month in the year in which it occurs.	INTEGER	-(10)9
Month_of_Calendar	For a given date, the number of the month in the calendar in which it occurs.	INTEGER	-(10)9
Quarter_of_Year	For a given date, the quarter of the year in which it occurs.	INTEGER	-(10)9
Quarter_of_Calendar	For a given date, the quarter number of the calendar in which it occurs.	INTEGER	-(10)9
Year_of_Calendar	For a given date, the year number of the calendar in which it occurs.	INTEGER	-(10)9
WeekEnd	The end of the week for the given date.	DATE	YY/MM/DD
WeekBegin	The beginning of the week for the given date.	DATE	YY/MM/DD
MonthBegin	The beginning of the month for the given date.	DATE	YY/MM/DD
MonthEnd	The end of the month for the given date.	DATE	YY/MM/DD
QuarterBegin	The beginning of the quarter for the given date.	DATE	YY/MM/DD

View Column Name	Description	Data Type	Format
QuarterEnd	The end of the quarter for the given date.	DATE	YY/MM/DD
YearBegin	The beginning of the year for the given date.	DATE	YY/MM/DD
YearEnd	The end of the year for the given date.	DATE	YY/MM/DD
IsBusinessDay	Whether or not the given day is a business day.	BYNET	-(3)9
BusinessWeekBegin	The first working day of the week in which the given date occurs.	DATE	YY/MM/DD
BusinessWeekEnd	The last working day of the week in which the given date occurs.	DATE	YY/MM/DD
BusinessMonthBegin	The first working day of the month in which the given date occurs.	DATE	YY/MM/DD
BusinessMonthEnd	The last working day of the month in which the given date occurs.	DATE	YY/MM/DD
BusinessQuarterBegin	The first working day of the quarter in which the given date occurs.	DATE	YY/MM/DD
BusinessQuarterEnd	The last working day of the quarter in which the given date occurs.	DATE	YY/MM/DD
BusinessYearBegin	The first working day of the year in which the given date occurs.	DATE	YY/MM/DD
BusinessYearEnd	The last working day of the year in which the given date occurs.	DATE	YY/MM/DD

Example: Querying for Day of the Week Using the ISO Calendar

This query returns the day of the week using the ISO calendar.

```

Set session calendar = iso;
Sel day_of_week from Sys_Calendar.Calendar where calendar_date =
date '2011-01-01';
day_of_week
-----
6

```

Examples

Example: Querying for Day of the Week Using the ISO Calendar

This query returns the day of the week using the ISO calendar:

```

Set session calendar = iso;
Sel day_of_week from Sys_Calendar.Calendar where calendar_date =
date '2011-01-01';
day_of_week
-----
6

```

Example: Querying for Day of the Week Using the COMPATIBLE Calendar

This query returns the day of the week using the COMPATIBLE calendar:

```

Set session calendar = compatible;
Sel day_of_week from Sys_Calendar.Calendar where calendar_date =
date '2011-01-01';
day_of_week
-----
1

```

Example: Querying for Beginning of the Week Using the ISO Calendar

This query returns the beginning of the week using the ISO calendar:

```

Set session calendar = iso;
Sel weekBegin from Sys_Calendar.BusinessCalendar where calendar_date =
date '2011-01-01';
WeekBegin
-----
10/12/27

```

Example: Querying for Beginning of the Week Using the COMPATIBLE calendar

This query returns the beginning of the week using the COMPATIBLE calendar:

```

Set session calendar = compatible;
Sel weekBegin from Sys_Calendar.BusinessCalendar where calendar_date =
date '2011-01-01';
WeekBegin
-----
11/01/01

```

Example: Adding an OFF Exception Day

By default, Monday is a working day in the ISO calendar. The following query adds an OFF exception for January 1, 2007, which is a Monday.

```
Exec DBC.CreateException('ISO', 'OFF', date '2007-01-01', 'Newyear Day');
```

The following query returns the next working day after January 1, 2007:

```
Sel BusinessWeekBegin from Sys_Calendar.BusinessCalendar where calendar_date =
date '2007-01-01';
BusinessWeekBegin
-----
07/01/02
```

Example: Querying for Beginning of the Week and the Week Number

This query returns the beginning of the week and the week number for 2009, assuming that no calendar is set for the session. The session uses the default business calendar “Teradata.”

```
SEL WeekBegin, week_of_year FROM Sys_Calendar.BusinessCalendar WHERE
calendar_date = DATE '2009-03-15';
Week_begin          week_of_year
-----
09/03/15            11
```

Business Calendar Functions

About the Business Calendar Functions

You can specify business calendar functions in an SQL statement wherever you specify UDFs, for example, in an INSERT, UPDATE, DELETE, MERGE, or SELECT statement.

The following facts apply to the business calendar functions:

- If you do not specify a calendar name, the calendar defaults to the session calendar.
- All functions related to a year, for example MonthNumber_Of_Year, are calculated relative to January 1 of that year.
- All functions related to a calendar, for example, DayNumber_Of_Calendar, are calculated relative to the beginning of the calendar, 1900-01-01.

- You can specify these functions anywhere in a DML statement, in CHECK CONSTRAINTS in the DDL statements, and all other places in an SQL statement where a UDF can be specified.
- All the definitions of the business calendar functions are stored in the TD_SYSFNLIB database. The format and title of the business calendar functions used in a SELECT statement follow the regular UDF style.
- These functions are available to all Teradata users and do not require any privileges.
- All the computations inside the functions are in UTC if the input is of type TIMESTAMP or TIMESTAMP WITH TIME ZONE.

Prerequisites for Using the Business Calendar Functions

Before you can use these functions, you must run the Database Initialization Program (DIP) utility and execute the DIPSYFNC script. The DIPALL or DIPSYFNC script will create the calendar functions in the TD_SYSFNLIB database. For more information about the DIP utility, see *Teradata Vantage™ - Database Utilities*, B035-1102.

If you have a UDF with the same name as a business calendar function, you must remove that UDF from the normal UDF search path before you can invoke the business calendar function. If the business calendar function is not found in the current database, Vantage searches for the function in the TD_SYSFNLIB database. Alternatively, you may invoke the calendar function by using the fully qualified syntax, `TD_SYSFNLIB.calendar_function_name`.

Example: Cumulative Sales During First Week of All Quarters

This example finds cumulative sales of an item during first week of all quarters. Results are shown for each calendar setting.

```
SEL Item_Code, SUM(Sale_Amt) FROM Sales_Tbl
WHERE WeekNumber_Of_Quarter(Sale_Date) = 1
GROUP BY Item_Code;
```

SET SESSION calendar = Teradata;

Item_Code	SUM(Sale_Amt)
101	225
102	120

SET SESSION calendar = Compatible

Item_Code	SUM(Sale_Amt)
101	245
102	100

SET SESSION calendar = ISO;

Item_Code	SUM(Sale_Amt)
101	135
102	30

Sales_Tbl		
Item_Code	Sale_Amt	Sale_Date
101	10	2008-12-30
101	15	2008-12-31
101	20	2009-01-01
101	25	2009-01-02
101	30	2009-01-03
101	35	2009-01-04
101	40	2009-01-05
101	45	2009-01-06
101	50	2009-01-07
101	55	2009-01-08
102	10	2009-01-01
102	20	2009-01-03
102	30	2009-01-05
102	40	2009-01-07
102	50	2009-01-09

Jan 2009 Calendar						
S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Examples

Example: Cumulative Sales During First Week of All Quarters - Teradata Calendar

This example shows how cumulative sales for the first week of all quarters are totaled using the Teradata calendar.

```
SET SESSION calendar = Teradata;
SEL Item_Code, SUM(Sale_Amt) FROM Sales_Tbl
WHERE WeekNumber_Of_Quarter(Sale_Date) = 1
GROUP BY Item_Code;
```

SET SESSION calendar = Teradata;

Item_Code	SUM(Sale_Amt)
101	225
102	120

Sales_Tbl		
Item_Code	Sale_Amt	Sale_Date
101	10	2008-12-30
101	15	2008-12-31
101	20	2009-01-01
101	25	2009-01-02
101	30	2009-01-03
101	35	2009-01-04
101	40	2009-01-05
101	45	2009-01-06
101	50	2009-01-07
101	55	2009-01-08
102	10	2009-01-01
102	20	2009-01-03
102	30	2009-01-05

102	40	2009-01-07
102	50	2009-01-09

Jan 2009 Calendar						
S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Example: Cumulative Sales During First Week of All Quarters - ISO Calendar

This example shows how cumulative sales for the first week of all quarters are totaled using the ISO calendar.

```
SET SESSION calendar = ISO;
SEL Item_Code, SUM(Sale_Amt) FROM Sales_Tbl
WHERE WeekNumber_Of_Quarter(Sale_Date) = 1
GROUP BY Item_Code;
```

SET SESSION calendar = ISO;

Item_Code	SUM(Sale_Amt)
101	135
102	30

Sales_Tbl		
Item_Code	Sale_Amt	Sale_Date
101	10	2008-12-30
101	15	2008-12-31
101	20	2009-01-01
101	25	2009-01-02
101	30	2009-01-03

101	35	2009-01-04
101	40	2009-01-05
101	45	2009-01-06
101	50	2009-01-07
101	55	2009-01-08
102	10	2009-01-01
102	20	2009-01-03
102	30	2009-01-05
102	40	2009-01-07
102	50	2009-01-09

Jan 2009 Calendar						
S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Example: Cumulative Sales During First Week of All Quarters - Compatible Calendar

This example shows how cumulative sales for the first week of all quarters are totaled using the Compatible calendar.

```
SET SESSION calendar = Compatible;
SEL Item_Code, SUM(Sale_Amt) FROM Sales_Tbl
WHERE WeekNumber_Of_Quarter(Sale_Date) = 1
GROUP BY Item_Code;
```

SET SESSION calendar = Compatible;

Item_Code	SUM(Sale_Amt)
101	245
102	100

Sales_Tbl		
Item_Code	Sale_Amt	Sale_Date
101	10	2008-12-30
101	15	2008-12-31
101	20	2009-01-01
101	25	2009-01-02
101	30	2009-01-03
101	35	2009-01-04
101	40	2009-01-05
101	45	2009-01-06
101	50	2009-01-07
101	55	2009-01-08
102	10	2009-01-01
102	20	2009-01-03
102	30	2009-01-05
102	40	2009-01-07
102	50	2009-01-09

Jan 2009 Calendar						
S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Example: Cumulative Weekday Sales During First Week of All Quarters - Teradata Calendar

This example finds cumulative sales of an item during weekdays of the first week of all quarters.

```

SET SESSION calendar = Teradata;
/* Setting SUNDAY and SATURDAY as weekend days. */
exec dbc.createbusinesscalendarpattern('teradata','SUNDAY','OFF',);
exec dbc.createbusinesscalendarpattern('teradata','SATURDAY','OFF',);
SEL Item_Code, SUM(Sale_Amt) FROM Sales_Tbl,
     Sys_Calendar.BusinessCalendar
WHERE WeekNumber_Of_Quarter(Sale_Date) = 1
AND calendar_date = Sale_Date
AND IsBusinessDay = 1
GROUP BY Item_Code
ORDER BY 1;

```

SET SESSION calendar = Teradata;

Item_Code	SUM(Sale_Amt)
101	190
102	120

Sales_Tbl		
Item_Code	Sale_Amt	Sale_Date
101	10	2008-12-30
101	15	2008-12-31
101	20	2009-01-01
101	25	2009-01-02
101	30	2009-01-03
101	35	2009-01-04
101	40	2009-01-05
101	45	2009-01-06
101	50	2009-01-07
101	55	2009-01-08
102	10	2009-01-01
102	20	2009-01-03
102	30	2009-01-05
102	40	2009-01-07

102	50	2009-01-09
-----	----	------------

Jan 2009 Calendar						
S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Example: Show Week 1 of September 2011

This example shows week 1 of September, 2011, for each business calendar setting.

```

SEL calendar_date, week_of_month FROM Sys_Calendar.BusinessCalendar
WHERE week_of_month = 1
AND month_of_year = 9
AND year_of_calendar = 2011
ORDER BY 1;
SET SESSION calendar = ISO;

```

Sep-2011 Calendar

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

```

calendar_date  week_of_month
-----
11/08/29      1
11/08/30      1
11/08/31      1
11/09/01      1

```

```

11/09/02      1
11/09/03      1

```

11/09/04 1

SET SESSION calendar = Teradata;

Sep-2011 Calendar

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

```

calendar_date  week_of_month
-----
11/09/04      1
11/09/05      1
11/09/06      1
11/09/07      1
11/09/08      1
11/09/09      1
11/09/10      1
SET SESSION calendar = Compatible;

```

Sep-2011 Calendar

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

```

calendar_date  week_of_month
-----
11/09/01      1
11/09/02      1

```

11/09/03	1
11/09/04	1
11/09/05	1
11/09/06	1
11/09/07	1

Example: Inventory Status at the Beginning of Each Week in a Month

The following example shows the inventory status at the beginning of each week for the month of September, 2011 for each business calendar setting.

```

SEL Item, BEGIN(week_duration) AS WeekBegin, Inventory
FROM Inventory_Tbl
EXPAND ON Duration AS week_duration
BY ANCHOR WEEK_BEGIN;

```

Inventory_Tbl

Item	Duration	Inventory
101	2011-09-01, 2011-09-05	12
101	2011-09-05, 2011-09-12	18
101	2011-09-12, 2011-09-18	15
101	2011-09-18, 2011-09-21	11

Sep-2011 Calendar

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

SET SESSION CALENDAR = Teradata;

Item	WeekBegin	Inventory
101	2011-09-04	12
101	2011-09-11	18

101	2011-09-18	11
-----	------------	----

SET SESSION CALENDAR = ISO;		
Item	WeekBegin	Inventory
101	2011-09-05	18
101	2011-09-12	15
101	2011-09-19	11

SET SESSION CALENDAR = Compatible;		
Item	WeekBegin	Inventory
101	2011-09-03	12
101	2011-09-10	18
101	2011-09-17	15

Example: Number of Weeks in Each Quarter in 2004

The following example shows the number of weeks in each quarter for 2004.

```
SEL quarter_of_year, MAX(week_of_quarter)
FROM Sys_Calendar.BusinessCalendar
WHERE year_of_calendar=2004
GROUP BY 1 ORDER BY 1;
```

SET SESSION CALENDAR = Teradata;	
Quarter_of_Year	Maximum(Week_of_Quarter)
1	13
2	13
3	13
4	13

SET SESSION CALENDAR = ISO;	
Quarter_of_Year	Maximum(Week_of_Quarter)
1	13

2	13
3	13
4	14

SET SESSION CALENDAR = COMPATIBLE;	
Quarter_of_Year	Maximum(Week_of_Quarter)
1	13
2	13
3	14
4	14

Example: Sales by Quarter

The following example shows how sales are calculated by quarter for each business calendar setting.

```

SEL QuarterNumber_Of_Year(Sale_Date) Quarter_No,
Item_Code, SUM(Sale_Amt) FROM Sales_Tbl
GROUP BY 1,2
ORDER BY 1,2;
SET SESSION calendar = ISO;

```

Quarter_No	Item_Code	SUM(Sale_Amt)
1	101	325
1	102	150

```
SET SESSION calendar = Teradata;
```

Quarter_No	Item_Code	SUM(Sale_Amt)
1	101	300
1	102	150
4	101	25

```
SET SESSION calendar = COMPATIBLE;
```

Quarter_No	Item_Code	SUM(Sale_Amt)
1	101	300
1	102	150
4	101	25

Sales_Tbl		
Item_Code	Sale_Amt	Sale_Date
101	10	2008-12-30
101	15	2008-12-31
101	20	2009-01-01
101	25	2009-01-02
101	30	2009-01-03
101	35	2009-01-04
101	40	2009-01-05
101	45	2009-01-06
101	50	2009-01-07
101	55	2009-01-08
102	10	2009-01-01
102	20	2009-01-03
102	30	2009-01-05
102	40	2009-01-07
102	50	2009-01-09

Jan 2009 Calendar						
S	M	T	W	T	F	S
	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

td_week_begin

Returns the week that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

ANSI Compliance

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

Result Type

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

td_week_begin Syntax

```
[TD_SYSFNLIB.] td_week_begin (
  expression_1 [, { calendar_name | NULL } [, expression_2 ] ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Week Before a DATE or TIMESTAMP

The following SQL statement:

```
SELECT TD_WEEK_BEGIN(TIMESTAMP'2012-01-15 12:00:00+02:00', 'ISO', TIME'05:00:00');
```

returns 2012-01-09 07:00:00+02:00.

td_week_end

Returns the week end that falls immediately after the DATE or TIMESTAMP specified in *expression_1*.

ANSI Compliance

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

Syntax

```
[TD_SYSFNLIB.] td_week_end (
  expression_1 [, { calendar_name | NULL } [, expression_2 ] ]
)
```

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Week End After a DATE or TIMESTAMP

The following SQL statement:

```
SELECT TD_WEEK_END(timestamp'2012-05-08 10:00:00');
```

returns 2012-05-12 23:59:59.

where no calendar specified indicates the session calendar, which in this example is TERADATA.

td_sunday

Returns the Sunday that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

Result Type

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

td_sunday Syntax

```
[TD_SYSFNLIB.] td_sunday (
  expression_1 [, { calendar_name | NULL } [, expression_2 ] ]
)
```

ANSI Compliance

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

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Sunday Before a DATE or TIMESTAMP

The following SQL statement:

```
SELECT TD_SUNDAY(timestamp'2012-05-08 10:00:00');
```

returns 2012-05-06 00:00:00.

where no calendar specified indicates the session calendar, which in this case is TERADATA.

td_monday

Returns the Monday that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

ANSI Compliance

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

Result Type

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

td_monday Syntax

```
[TD_SYSFNLIB.] td_monday (
  expression_1 [, { calendar_name | NULL } [, expression_2 ] ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR

- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Monday Before a DATE or TIMESTAMP

The following SQL statement:

```
TD_MONDAY(Date, 'TERADATA');
```

returns 12/05/07.

td_tuesday

Returns the Tuesday that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

ANSI Compliance

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

Result Type

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

td_tuesday Syntax

```
[TD_SYSFNLIB.] td_tuesday (
  expression_1 [, { calendar_name | NULL } [, expression_2 ] ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Tuesday Before a DATE or TIMESTAMP

The following SQL statement:

```
SELECT TD_TUESDAY(TIMESTAMP'2012-05-08 10:00:00');
```

returns 2012-05-08 00:00:00.

where no calendar specified indicates the session calendar, which in this case is TERADATA.

td_wednesday

Returns the Wednesday that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

ANSI Compliance

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

Result Type

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

td_wednesday Syntax

```
[TD_SYSFNLIB.] td_wednesday (
    expression_1 [, { calendar_name | NULL } [, expression_2 ] ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Wednesday Before the DATE or TIMESTAMP

The following SQL statement:

```
SELECT TD_WEDNESDAY(TIMESTAMP'2012-05-08 10:00:00+01:00', 'ISO');
```

returns 2012-05-02 01:00:00+01:00.

td_thursday

Returns the Thursday that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

ANSI Compliance

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

Result Type

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

td_thursday Syntax

```
[TD_SYSFNLIB.] td_thursday (
    expression_1 [, { calendar_name | NULL } [, expression_2 ] ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME WITH TIME ZONE
- NULL

Example: Querying the Thursday Before a DATE or TIMESTAMP

The following SQL statement:

```
SELECT TD_THURSDAY(TIMESTAMP '2012-05-08 10:00:00', NULL, NULL);
```

returns 2012-05-03 00:00:00.

where NULL indicates the session calendar, which in this case is TERADATA.

td_friday

Returns the Friday that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

ANSI Compliance

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

Result Type

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

td_friday Syntax

```
[TD_SYSFNLIB.] td_friday (
    expression_1 [, { calendar_name | NULL } [, expression_2 ] ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Friday Before a DATE or TIMESTAMP

The following SQL statement:

```
SELECT TD_FRIDAY(TIMESTAMP'2012-05-08 10:00:00',NULL,TIME'05:00:00');
```

returns 2012-05-04 05:00:00.

where NULL indicates the session calendar, which in this case is TERADATA.

td_saturday

Returns the Saturday that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

ANSI Compliance

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

Result Type

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

td_saturday Syntax

```
[TD_SYSFNLIB.] td_saturday (
  expression_1 [, { calendar_name | NULL } [, expression_2 ] ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Saturday Before a DATE or TIMESTAMP

The following SQL statement:

```
SELECT TD_SATURDAY(TIMESTAMP '2012-05-08 10:00:00', NULL, TIME '05:00:00+03:00');
```

returns 2012-05-05 02:00:00.

where NULL indicates the session calendar, which in this case is TERADATA.

DayNumber_Of_Week

Returns the number of days from the beginning of the week to the specified date.

The DayNumber_Of_Week function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value between 1 and 7, representing the day of the week, where the first day of the week = 1 and the last day of the week = 7. The first day of the week is defined by the business calendar

the session is using. For example, in the ISO calendar, Monday = 1 and in the Teradata business calendar Monday = 2.

DayNumber_Of_Week Syntax

```
[TD_SYSFNLIB.] DayNumber_Of_Week (
    expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME WITH TIME ZONE

- NULL

Example: Querying Employee Names

The following query returns the names of employees who joined the company on the first day of the week:

```
SELECT empname from emp
WHERE DAYNUMBER_OF_WEEK (date_of_join)= 1;
```

Related Information

For more information about the CALENDAR system view, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

td_month_begin

Returns the month that begins immediately before the DATE or TIMESTAMP specified in *expression_1*.

ANSI Compliance

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

Result Type

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

td_month_begin Syntax

```
[TD_SYSFNLIB.] td_month_begin (
  expression_1 [, { calendar_name | NULL } ] [, expression_2 ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the Month Beginning Immediately Before the DATE or TIMESTAMP

The following SQL statement:

```
SELECT TD_MONTH_BEGIN(DATE '2012-01-15');
```

returns 12/01/01.

where no calendar specified indicates the session calendar, which in this case is Teradata.

td_month_end

Returns the month that ends immediately after the DATE or TIMESTAMP specified in *expression_1*.

ANSI Compliance

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

Result Type

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

td_month_end Syntax

```
[TD_SYSFNLIB.] td_month_end (
    expression_1 [, { calendar_name | NULL } ] [, expression_2 ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the Month Ending Immediately After the DATE or TIMESTAMP

The following SQL statement:

```
SELECT TD_MONTH_END(DATE '2012-01-15', NULL);
```

returns 12/01/31.

where NULL indicates the session calendar, which in this case is Teradata.

DayNumber_Of_Month

Returns the number of days from the beginning of the month to the specified date.

The DayNumber_Of_Month function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value from 1 through 31.

DayNumber_Of_Month Syntax

```
[TD_SYSFNLIB.] DayNumber_Of_Month (
    expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME WITH TIME ZONE
- NULL

Examples: Querying the Number of Days from the Month Beginning

In both of the following examples, no business calendar is set for the session, so the query uses the system-defined business calendar Teradata.

The following query gives the day number of the month as 31 instead of 1 because the input is converted to UTC 2009-12-31 19:00:00.

```
SET TIME ZONE INTERVAL '05:00' HOUR TO MINUTE;
SELECT DAYNUMBER_OF_MONTH (TIMESTAMP'2010-01-01 00:00:00');
```

The following query gives the day number of the month as 31 instead of 1 because the input is converted to UTC 2009-12-31 21:00:00.

```
SEL DAYNUMBER_OF_MONTH (TIMESTAMP'2010-01-01 00:00:00+03:00');
```

Related Information

For more information about the CALENDAR system view, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

DayOccurrence_Of_Month

Returns the *n*th occurrence of the weekday in the month for the specified date.

The DayOccurrence_of_Month function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value between 1 and 5, representing the *n*th occurrence of the weekday in the month.

DayOccurrence_Of_Month Syntax

```
[TD_SYSFNLIB.] DayOccurrence_Of_Month (
    expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIMEZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME WITH TIME ZONE
- NULL

Example: Querying the nth Occurrence of the Weekday in the Month

If the current date is May 1, 2010, the following queries return 1 because May 1, 2010, falls on the first Saturday of the month:

```
SELECT TD_SYSFNLIB.DAYOCCURRENCE_OF_MONTH(CURRENT_DATE);
SELECT TD_SYSFNLIB.DAYOCCURRENCE_OF_MONTH(DATE '2010-05-01');
```

Related Information

For more information, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

WeekNumber_Of_Month

Returns the number of weeks from the beginning of the month to the specified date.

The WeekNumber_Of_Month function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value between 1 and 5, representing the *n*th occurrence of the week in the month. The value 0 means the partial week, Week0.

WeekNumber_Of_Month Syntax

```
[TD_SYSFNLIB.] WeekNumber_Of_Month (
    expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIMEZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Number of Weeks from the Month Beginning

If the current date is January 5, 1901, the following query returns 0 because January 5, 1901, is in week 0 of January in the Teradata calendar:

```
SELECT TD_SYSFNLIB.WEekNUMBER_OF_MONTH (CURRENT_DATE);
```

The following query returns 1 because January 5, 1901, is in week 1 of January in the ISO calendar:

```
SELECT TD_SYSFNLIB.WEekNUMBER_OF_MONTH (DATE '1901-01-05', 'ISO');
```

td_year_begin

Returns the year that begins immediately before the DATE or TIMESTAMP specified in *expression_1*.

ANSI Compliance

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

Result Type

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

td_year_begin Syntax

```
[TD_SYSFNLIB.] td_year_begin (
  expression_1 [, { calendar_name | NULL } ] [, expression_2 ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Year That Begins Immediately Before the DATE or TIMESTAMP

The following SQL statement:

```
SELECT TD_YEAR_BEGIN(TIMESTAMP'2012-01-15 12:00:00', 'ISO', NULL);
```

returns 2012-01-02 00:00:00.

td_year_end

Returns the year that ends immediately after the DATE or TIMESTAMP specified in *expression_1*.

ANSI Compliance

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

Result Type

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

td_year_end Syntax

```
[TD_SYSFNLIB.] td_year_end (
    expression_1 [, { calendar_name | NULL } ] [, expression_2 ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Year Ending After the DATE or TIMESTAMP

The following SQL statement:

```
SELECT TD_YEAR_END(TIMESTAMP'2012-01-15 10:00:00','ISO');
```

returns 2012-12-30 23:59:59.

DayNumber_Of_Year

Returns the number of days from the beginning of the year to the specified date.

The DayNumber_Of_Year function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value from 1 through 366.

DayNumber_Of_Year Syntax

```
[TD_SYSFNLIB.] DayNumber_Of_Year (
    expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIMEZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME WITH TIME ZONE
- NULL

Example: Querying the Year Ending Immediately After the DATE or TIMESTAMP

If the current date is February 10, 2010, the following queries return 41 because February 10, 2010, is the 41st day since the beginning of the Teradata calendar year:

```
SELECT TD_SYSFNLIB.DAYNUMBER_OF_YEAR(CURRENT_DATE);
SELECT TD_SYSFNLIB.DAYNUMBER_OF_YEAR(DATE '2010-02-10');
```

Related Information

For more information about the CALENDAR system view, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

WeekNumber_Of_Year

Returns the number of weeks from the beginning of the year to the specified date.

The WeekNumber_Of_Year function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value from 0 through 53, where 1 is the first week of the year and 53 is the last week of the year. The value 0 means that there is a partial week in the year, Week0.

WeekNumber_Of_Year Syntax

```
[TD_SYSFNLIB.] WeekNumber_Of_Year (
    expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The first argument, expression, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, calendar_name, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Number of Weeks from the Year Beginning

If the current date is January 5, 1901, the following query returns 0 because January 5, 1901, is in week 0 of 1901 in the Teradata calendar:

```
SELECT TD_SYSFNLIB.WEekNUMBER_OF_YEAR (CURRENT_DATE);
```

The following query returns 1 because January 5, 1901, is in week 1 of 1901 in the COMPATIBLE calendar:

```
SELECT TD_SYSFNLIB.WEekNUMBER_OF_YEAR (DATE '1901-01-05', 'COMPATIBLE');
```

Related Information

For more information, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

MonthNumber_Of_Year

Returns the number of months from the beginning of the year to the specified date.

The MonthNumber_Of_Year function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value between 1 and 12, representing the n th month of the year.

MonthNumber_Of_Year Syntax

```
[TD_SYSFNLIB.] MonthNumber_Of_Year (
    expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Number of Months from the Year Beginning

If the current date is March 15, 2011, the following query returns 3 because March 15, 2011, is in the third month of the Teradata calendar:

```
SELECT TD_SYSFNLIB.MONTHNUMBER_OF_YEAR(CURRENT_DATE);
```

The following query returns 3 because March 15, 2011, is in the third month of the ISO calendar:

```
SELECT TD_SYSFNLIB.MONTHNUMBER_OF_YEAR(DATE '2011-03-15', 'ISO');
```

Related Information

For more information about the CALENDAR system view, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

td_quarter_begin

Returns the quarter that begins immediately before the DATE or TIMESTAMP specified in *expression_1*.

ANSI Compliance

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

Result Type

The return value is one of the following data types:

- DATE

- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

td_quarter_begin Syntax

```
[TD_SYSFNLIB.] td_quarter_begin (
    expression_1 [, { calendar_name | NULL } ] [, expression_2 ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Quarter Beginning Immediately Before the DATE or TIMESTAMP

The following SQL statement:

```
SELECT TD_QUARTER_BEGIN(DATE '2012-01-15', 'COMPATIBLE');
```

returns 12/01/01.

td_quarter_end

Returns the quarter that ends immediately after the DATE or TIMESTAMP specified in *expression_1*.

ANSI Compliance

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

Result Type

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

td_quarter_end Syntax

```
[TD_SYSFNLIB.] td_quarter_end (
  expression_1 [, { calendar_name | NULL } ] [, expression_2 ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

expression_2

A value of TIME, TIME WITH TIME ZONE, or NULL. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Quarter Ending Immediately After the DATE or TIMESTAMP

The following SQL statement:

```
SELECT TD_QUARTER_END(TIMESTAMP'2012-01-15 10:00:00', 'COMPATIBLE');
```

returns 2012-03-31 23:59:59.

WeekNumber_Of_Quarter

Returns the number of weeks from the beginning of the quarter to the specified date.

ANSI Compliance

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

Result Type

The result is an INTEGER value between 1 and 14, representing the *n* th week of the quarter. The value 0 means the partial week, Week0.

WeekNumber_Of_Quarter Syntax

```
[TD_SYSFNLIB.] WeekNumber_Of_Quarter (
    expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR

- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Number Of Weeks from the Beginning of the Quarter

If the current date is April 20, 2011, the following query returns 3 because April 20, 2011, is in the third week of the quarter in the Teradata calendar:

```
SELECT TD_SYSFNLIB.WEekNUMBER_OF_Quarter(CURRENT_DATE);
```

The following query returns 3 because April 20, 2011, is in the third week of the quarter in the ISO calendar:

```
SELECT TD_SYSFNLIB.WEekNUMBER_OF_Quarter (DATE '2011-04-20', 'ISO');
```

MonthNumber_Of_Quarter

Returns the number of months from the beginning of the quarter to the specified date.

The MonthNumber_Of_Quarter function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value between 1 to 3, representing the n th month of the quarter.

MonthNumber_Of_Quarter Syntax

```
[TD_SYSFNLIB.] MonthNumber_Of_Quarter (
  expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The first argument, expression, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, calendar_name, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Number of Months from the Quarter Beginning

If the current date is April 20, 2011, the following query returns 1 because April 20, 2011, is in the first month of the quarter in the Teradata calendar:

```
SELECT TD_SYSFNLIB.MONTHNUMBER_OF_QUARTER(CURRENT_DATE);
```

The following query returns 1 because April 20, 2011, is in the first month of the quarter in the ISO calendar:

```
SELECT TD_SYSFNLIB.MONTHNUMBER_OF_QUARTER(DATE '2011-04-20', 'ISO');
```

Related Information

For more information, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

QuarterNumber_Of_Year

Returns the number of quarters from the beginning of the year to the specified date.

The QuarterNumber_Of_Year function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value between 1 and 4, representing the *n* th quarter of the year.

QuarterNumber_Of_Year Syntax

```
[TD_SYSFNLIB.] QuarterNumber_Of_Year (
    expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The first argument, expression, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, calendar_name, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Number of Quarters from the Year Beginning

If the current date is December 20, 2010, the following query returns 4 because December 20, 2010, is in the fourth quarter in the Teradata calendar:

```
SELECT TD_SYSFNLIB.QUARTERNUMBER_OF_YEAR(CURRENT_DATE);
```

The following query returns 4 because December 10, 2010, is in the fourth quarter in the ISO calendar:

```
SELECT TD_SYSFNLIB.QUARTERNUMBER_OF_YEAR(DATE '2010-12-20', 'ISO');
```

Related Information

For more information about the CALENDAR system view, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

DayNumber_Of_Calendar

Returns the number of days from the beginning of the business calendar to the specified date.

The DayNumber_Of_Calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value representing the number of days since and including the beginning of the business calendar.

DayNumber_Of_Calendar Syntax

```
[TD_SYSFNLIB.] DayNumber_Of_Calendar (
    expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Number of Days from the Beginning of the Business Calendar

If the current date is January 05, 1901, the following queries return 370 since January 05, 1901, is the 370th day since January 1, 1900:

```
SELECT TD_SYSFNLIB.DAYNUMBER_OF_CALENDAR(CURRENT_DATE);
SELECT TD_SYSFNLIB.DAYNUMBER_OF_CALENDAR(DATE '1901-01-05');
```

Related Information

For more information about the CALENDAR system view, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

WeekNumber_Of_Calendar

Returns the number of weeks from the beginning of the business calendar to the specified date.

The WeekNumber_Of_Calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value. The value 0 means there is a partial week in the year, which is Week0.

WeekNumber_Of_Calendar Syntax

```
[TD_SYSFNLIB.] WeekNumber_Of_Calendar (
    expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The first argument, expression, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, calendar_name, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Number of Weeks from the Beginning of the Business Calendar

If the current date is January 5, 1901, the following query returns 52 because January 5, 1901, is in week 52 since the start of the Teradata calendar on 1900-01-01:

```
SELECT TD_SYSFNLIB.WEekNUMBER_OF_CALENDAR (CURRENT_DATE);
```

The following query returns 53 because January 5, 1901, is in week 53 since the start of the COMPATIBLE calendar on 1900-01-01:

```
SELECT TD_SYSFNLIB.WEekNUMBER_OF_CALENDAR (DATE '1901-01-05', 'COMPATIBLE');
```

MonthNumber_Of_Calendar

Returns the number of months from the beginning of the calendar to the specified date.

The MonthNumber_Of_Calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value representing the number of months since and including the beginning of the calendar.

MonthNumber_Of_Calendar Syntax

```
[TD_SYSFNLIB.] MonthNumber_Of_Calendar (
    expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR

- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying Number of Months from the Beginning of the Calendar

If the current date is May 5, 1901, the following query returns 17 because May 5, 1901, is in the 17th month since the start of the Teradata calendar on 1900-01-01:

```
SELECT TD_SYSFNLIB.MONTHNUMBER_OF_CALENDAR(CURRENT_DATE);
```

The following query returns 17 because May 5, 1901, is in the 17th month since the start of the COMPATIBLE calendar on 1900-01-01:

```
SELECT TD_SYSFNLIB.MONTHNUMBER_OF_CALENDAR(DATE '1901-05-05', 'COMPATIBLE');
```

Related Information

For more information, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

QuarterNumber_Of_Calendar

Returns the number of quarters from the beginning of the calendar to the specified date.

The QuarterNumber_Of_Calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value, representing the number of quarters since and including the beginning of the calendar.

QuarterNumber_Of_Calendar Syntax

```
[TD_SYSFNLIB.] QuarterNumber_Of_Calendar (
  expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Query the Number of Quarters from the Beginning of the Calendar

If the current date is May 5, 1901, the following query returns 6 because May 5, 1901, is in the 6th quarter since 1900-01-01, the beginning of the Teradata calendar:

```
SELECT TD_SYSFNLIB.QUARTERNUMBER_OF_CALENDAR(CURRENT_DATE);
```

The following query returns 6 because May 5, 1901, is in the 6th quarter since 1900-01-01, the beginning of the COMPATIBLE calendar:

```
SELECT TD_SYSFNLIB.QUARTERNUMBER_OF_CALENDAR (DATE '1901-05-05', 'COMPATIBLE');
```

Related Information

For more information about the CALENDAR system view, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

YearNumber_Of_Calendar

Returns the year of the specified date.

The YearNumber_Of_Calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value representing the year in 4-digit format.

YearNumber_Of_Calendar Syntax

Syntax

```
[TD_SYSFNLIB.] YearNumber_Of_Calendar (
    expression [,] [ calendar_name | NULL ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

NULL

The business calendar for the session.

Argument Types

The first argument, expression, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, calendar_name, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying the Year of the Specified Date

If the current date is June 14, 1906, the following queries return 1906:

```
SELECT TD_SYSFNLIB.YEARNUMBER_OF_CALENDAR(CURRENT_DATE);  
SELECT TD_SYSFNLIB.YEARNUMBER_OF_CALENDAR (DATE '1906-06-14', 'COMPATIBLE');
```

Related Information

For more information, see:

- *Teradata Vantage™ - Data Dictionary*, B035-1092

Calendar Functions

The following sections describe the functions that provide support for DateTime operations that use calendar attributes.

td_day_of_week/DayOfWeek

Returns the day of the week on which the specified date falls.

The `td_day_of_week` function provides improved performance compared to using the `Sys_Calendar.Calendar` system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value from 1 through 7, representing the day of the week, where Sunday = 1 and Saturday = 7.

td_day_of_week/DayOfWeek Syntax

```
{ DayOfWeek | [SYSLIB.] td_day_of_week } ( expression )
```

Syntax Elements

SYSLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Argument Types

`td_day_of_week` is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the Day of the Week on Which the Specified Date Falls

If the current date is October 18, 2010, which is a Monday, the following queries return the value 2 as the result since Monday is the 2nd day of the week.

```
SELECT td_day_of_week(CURRENT_DATE);
SELECT td_day_of_week(DATE '2010-10-18');
```

```
SELECT DAYOFWEEK(CURRENT_DATE);
SELECT DAYOFWEEK(DATE '2010-10-18');
```

td_day_of_month

Returns the number of days from the beginning of the month through the specified date.

The `td_day_of_month` function provides improved performance compared to using the `Sys_Calendar.Calendar` system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value from 1 through 31.

td_day_of_month Syntax

```
[SYSLIB.] td_day_of_month ( expression )
```

Syntax Elements

SYSLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Argument Types

td_day_of_month is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the Number of Days from the Beginning of the Month

If the current date is May 27, 2010, the following queries return the value 27 as the result since May 27, 2010 is the 27th day from the beginning of the month of May.

```
SELECT td_day_of_month(CURRENT_DATE);
SELECT td_day_of_month(DATE '2010-05-27');
```

Related Information

- For more information on overloaded functions, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For more information about the CALENDAR system view, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

td_day_of_year

Returns the number of days from the beginning of the year (January 1st) to the specified date.

The td_day_of_year function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value from 1 through 366.

td_day_of_year Syntax

```
[SYSLIB.] td_day_of_year ( expression )
```

Syntax Elements

SYSLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Argument Types

td_day_of_year is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the Number of Days from the Beginning of the Year

If the current date is February 10, 2010, the following queries return the value 41 as the result since February 10, 2010 is the 41st day from the beginning of the year.

```
SELECT td_day_of_year(CURRENT_DATE);
SELECT td_day_of_year(DATE '2010-02-10');
```

Related Information

- For more information about function name overloading, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147
- *Teradata Vantage™ - Data Dictionary*, B035-1092

td_weekday_of_month

Returns the *n*th occurrence of the weekday in the month for the specified date.

The `td_weekday_of_month` function provides improved performance compared to using the `Sys_Calendar.Calendar` system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value from 1 to 5, representing the n th occurrence of the weekday in the month.

td_weekday_of_month Syntax

```
[SYSLIB.] td_weekday_of_month ( expression )
```

Syntax Elements

SYSLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Argument Types

`td_weekday_of_month` is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the nth Occurrence of the Weekday in the Month

If the current date is May 01, 2010, the following queries return the value 1 as the result since May 01, 2010 falls on the first Saturday of the month.

```
SELECT td_weekday_of_month(CURRENT_DATE);
SELECT td_weekday_of_month(DATE '2010-05-01');
```

Related Information

- For more information about function name overloading, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147
- For more information about the CALENDAR system view, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

td_week_of_month

Returns the n th full week from the beginning of the month to the specified date.

The `td_week_of_month` function provides improved performance compared to using the `Sys_Calendar.Calendar` system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value from 0 to 5, representing the n th full week from the beginning of the month, where the first partial week is 0.

td_week_of_month Syntax

```
[SYSLIB.] td_week_of_month ( expression )
```

Syntax Elements

SYSLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Argument Types

`td_week_of_month` is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the *n*th Full Week from the Beginning of the Month

If the current date is May 01, 2010, the following queries return the value 0 as the result since May 01, 2010 falls on the first partial week of May.

```
SELECT td_week_of_month(CURRENT_DATE);
SELECT td_week_of_month(DATE '2010-05-01');
```

Related Information

- For more information about function name overloading, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147
- For more information about the CALENDAR system view, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

td_week_of_year

Returns the *n*th full week from the beginning of the year (January 1st) to the specified date.

The `td_week_of_year` function provides improved performance compared to using the `Sys_Calendar.Calendar` system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value from 0 to 53, representing the *n*th full week from the beginning of the year, where the first partial week is 0.

td_week_of_year Syntax

```
[SYSLIB.] td_week_of_year ( expression )
```

Syntax Elements

SYSLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Argument Types

td_week_of_year is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the nth Full Week from the Beginning of the Year

If the current date is May 04, 2010, the following queries return the value 18 as the result since May 04, 2010 falls on the 18th week of the year.

```
SELECT td_week_of_year(CURRENT_DATE);
SELECT td_week_of_year(DATE '2010-05-04');
```

Related Information

- For more information about function name overloading, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147
- *Teradata Vantage™ - Data Dictionary*, B035-1092

td_week_of_calendar

Returns the number of weeks from the beginning of the calendar starting on 01/01/1900 to the specified date.

The td_week_of_calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value representing the number of full weeks since and including the week of 01/01/1900, where the first partial week is 0.

td_week_of_calendar Syntax

```
[SYSLIB.] td_week_of_calendar ( expression )
```

Syntax Elements

SYSLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Argument Types

td_week_of_calendar is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the Number of Weeks from the Beginning of the Calendar

If the current date is January 10, 1901, the following queries return the value 53 as the result since January 10, 1901 falls on the 53rd week since January 01, 1900.

```
SELECT td_week_of_calendar(CURRENT_DATE);
SELECT td_week_of_calendar(DATE '1901-01-10');
```

Related Information

- For more information about function name overloading, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147

- *Teradata Vantage™ - Data Dictionary, B035-1092*

td_month_of_quarter

Returns the number of months from the beginning of the quarter to the specified date.

The `td_month_of_quarter` function provides improved performance compared to using the `Sys_Calendar.Calendar` system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value from 1 to 3.

td_month_of_quarter Syntax

```
[SYSLIB.] td_month_of_quarter ( expression )
```

Syntax Elements

SYSLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Argument Types

`td_month_of_quarter` is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the Number of Months from the Beginning of the Quarter

If the current date is June 12, 2010, the following queries return the value 3 as the result because June 12, 2010 falls on the 3rd month of the 2nd quarter.

```
SELECT td_month_of_quarter(CURRENT_DATE);
SELECT td_month_of_quarter(DATE '2010-06-12');
```

Related Information

For more information, see:

- *Teradata Vantage™ - SQL External Routine Programming*, B035-1147
- *Teradata Vantage™ - Data Dictionary*, B035-1092

td_month_of_year

Returns the number of months from the beginning of the year (January 1st) to the specified date.

The `td_month_of_year` function provides improved performance compared to using the `Sys_Calendar.Calendar` system view to obtain similar results.

Result Type

The result is an INTEGER value from 1 to 12.

td_month_of_year Syntax

```
[SYSLIB.] td_month_of_year ( expression )
```

Syntax Elements

SYSLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Argument Types

`td_month_of_year` is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the Number of Months from the Beginning of the Year

If the current date is August 29, 2010, the following queries return the value 8 as the result because August 29, 2010 falls on the 8th month of the year.

```
SELECT td_month_of_year(CURRENT_DATE);
SELECT td_month_of_year(DATE '2010-08-29');
```

Related Information

- For more information about function name overloading, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147
- *Teradata Vantage™ - Data Dictionary*, B035-1092

td_month_of_calendar

Returns the number of months from the beginning of the calendar starting on 01/01/1900 to the specified date.

The `td_month_of_calendar` function provides improved performance compared to using the `Sys_Calendar.Calendar` system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value representing the number of months since and including January, 1900.

td_month_of_calendar Syntax

```
[SYSLIB.] td_month_of_calendar ( expression )
```


Syntax Elements

SYSLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Argument Types

td_month_of_calendar is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the Number of Months from the Beginning of the Calendar Year

If the current date is August 29, 1901, the following queries return the value 20 as the result since August 29, 1901 falls on the 20th month since January 01, 1900.

```
SELECT td_month_of_calendar(CURRENT_DATE);
SELECT td_month_of_calendar(DATE '1901-08-29');
```

Related Information

- For more information about function name overloading, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147
- *Teradata Vantage™ - Data Dictionary*, B035-1092

td_quarter_of_year

Returns the quarter number of the year for the specified date.

The td_quarter_of_year function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value from 1 to 4, representing the quarter number from the beginning of the year, where 1 = first quarter (Jan/Feb/Mar) and 4 = fourth quarter (Oct/Nov/Dec).

td_quarter_of_year Syntax

```
[SYSLIB.] td_quarter_of_year ( expression )
```

Syntax Elements

SYSLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Argument Types

td_quarter_of_year is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the Quarter Number of the Year

If the current date is November 14, 1983, the following queries return the value 4 as the result since November 14, 1983 falls on the 4th quarter of the year.

```
SELECT td_quarter_of_year(CURRENT_DATE);
SELECT td_quarter_of_year(DATE '1983-11-14');
```

Related Information

- For more information about function name overloading, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147
- *Teradata Vantage™ - Data Dictionary*, B035-1092

td_quarter_of_calendar

Returns the number of quarters from the beginning of the calendar starting on 01/01/1900 to the specified date.

The `td_quarter_of_calendar` function provides improved performance compared to using the `Sys_Calendar.Calendar` system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value representing the number of quarters since and including the first quarter of 1900.

td_quarter_of_calendar Syntax

```
[SYSLIB.] td_quarter_of_calendar ( expression )
```

Syntax Elements

SYSLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Argument Types

`td_quarter_of_calendar` is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the Number of Quarters from the Beginning of the Calendar

If the current date is November 14, 1901, the following queries return the value 8 as the result since November 14, 1901 falls on the 8th quarter since January 01, 1900.

```
SELECT td_quarter_of_calendar(CURRENT_DATE);
SELECT td_quarter_of_calendar(DATE '1901-11-14');
```

Related Information

- For more information about function name overloading, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147
- *Teradata Vantage™ - Data Dictionary*, B035-1092

td_year_of_calendar

Returns the year of the specified date.

The `td_year_of_calendar` function provides improved performance compared to using the `Sys_Calendar.Calendar` system view to obtain similar results.

ANSI Compliance

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

Result Type

The result is an INTEGER value in 4 digit format representing the year of the specified date.

td_year_of_calendar Syntax

```
[SYSLIB.] td_year_of_calendar ( expression )
```

Syntax Elements

SYSLIB.

Name of the database where the function is located.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Argument Types

td_year_of_calendar is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Example: Querying for the Year of the Specified Date

If the current date is November 14, 1977, the following queries return the value 1977 as the result, which is the year of the specified date.

```
SELECT td_year_of_calendar(CURRENT_DATE);
SELECT td_year_of_calendar(DATE '1977-11-14');
```

Related Information

- For more information about function name overloading, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For more information about the CALENDAR system view, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

DateTime and Interval Functions and Expressions

The following sections describe functions and expressions that operate on ANSI DateTime and Interval values, as well as functions and expressions that operate on Teradata DATE values, which are extensions to the ANSI SQL:2011 standard.

ANSI DateTime Data Types

ANSI DateTime data types include:

- DATE
- TIME
- TIME WITH TIME ZONE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Interval Data Types

There are two categories of ANSI Interval data types:

- Year-Month Intervals, which include:
 - YEAR
 - YEAR TO MONTH
 - MONTH
- Day-Time Intervals, which include:
 - DAY
 - DAY TO HOUR
 - DAY TO MINUTE
 - DAY TO SECOND
 - HOUR
 - HOUR TO MINUTE
 - HOUR TO SECOND
 - MINUTE
 - MINUTE TO SECOND
 - SECOND

ANSI DateTime and Interval Data Type Assignment Rules

Data Type Compatibility and Conversion

The following rules apply to assignments involving ANSI DateTime or Interval data types:

Source Type	Target Type	Description
DATE	DATE	The types are compatible and assignments do not require conversion. For compatibility with existing Teradata assignments, non-ANSI operations such as assigning a DATE to an INTEGER or an INTEGER to a DATE (with validity checking) follow existing Teradata assignment rules.
TIME	TIME	The types are compatible and assignments do not require conversion. The Teradata system value TIME is encoded as a REAL and is not compatible with ANSI TIME or TIME WITH TIME ZONE.
TIMESTAMP	TIMESTAMP	The types are compatible and assignments do not require conversion.
Year-Month INTERVAL	Year-Month INTERVAL	
Day-Time INTERVAL	Day-Time INTERVAL	
Numeric	DATE	Vantage performs implicit type conversion before the assignment. For more information, see <i>Teradata Vantage™ - Data Types and Literals</i> , B035-1143.
DATE	<ul style="list-style-type: none"> • Character • Numeric • TIMESTAMP 	
Character	<ul style="list-style-type: none"> • DATE • TIME • TIMESTAMP 	
TIME	TIMESTAMP	
TIMESTAMP	<ul style="list-style-type: none"> • DATE • TIME 	
Interval	Exact Numeric	
Note: The INTERVAL type must have only one field. For example, INTERVAL YEAR.		

Source Type	Target Type	Description
Exact Numeric	Interval	

For all other source/target data type combinations in assignments involving ANSI DateTime or Interval data types, the types must be explicitly converted.

To perform explicit conversions on ANSI DateTime or Interval data types, use the CAST function.

CAST Syntax

```
CAST ( expression AS { ansi_sql_data_type | data_definition_list } )
```

Syntax Elements

expression

An expression with known data type to be cast as a different data type.

ansi_sql_data_type

The new data type for expression.

data_definition_list

The new data type or data attributes or both for expression.

Interval Data Type Assignment Rules

The following rules apply to Year-Month INTERVAL assignments.

WHEN ...	THEN ...
the types match	assignment is straightforward.
the source is INTERVAL YEAR and the target is INTERVAL YEAR TO MONTH	the value for MONTH in the target is set to zero.
the source is INTERVAL MONTH and the target is INTERVAL YEAR TO MONTH	the source is extended to include the YEAR field initialized to zero, and the resulting interval is normalized. For example, if the source is '15' then the extended source is '0-15', normalized to '1-03'.
the target is INTERVAL MONTH and the source is either INTERVAL YEAR or INTERVAL YEAR TO MONTH	the source is converted to INTERVAL MONTH before assignment. For example, if the source is '2-11', it is converted to '35'.

WHEN ...	THEN ...
the least significant field of the source is lower than that of the target	the values of fields in the source with precision lower than the least significant field of the target are truncated. For example, if a source of INTERVAL '32' MONTH is assigned to a target column of type INTERVAL YEAR, the value stored is '2'.

The following rules apply to Day-Time INTERVAL assignments.

WHEN ...	THEN ...
the types match	assignment is straightforward.
the target is of lower significance than the least significant field of the source	values for those fields are set to zero. For example, if the source is INTERVAL '49:30' HOUR TO MINUTE and it is assigned to a target column of type INTERVAL HOUR(4) TO SECOND(2), the value stored is '49:30:00.00'.
the target has fields of higher significance than the most significant field of the source	the source type is extended to match the target type, setting the new fields to zeros, and normalizing the content as the final step. For example, if the source is INTERVAL '49:30' HOUR TO MINUTE and it is assigned to a target column of type INTERVAL DAY TO MINUTE, the value stored is '2 1:30'.
the least significant field of the source is lower than that of the target	the values of fields in the source with precision lower than the least significant field of the target are truncated. For example, if the source is INTERVAL '10:12:58' HOUR TO SECOND and it is assigned to a target column of type INTERVAL HOUR TO MINUTE, the value stored is '10:12'.

Scalar Operations on ANSI SQL:2011 DateTime and Interval Values

Teradata SQL defines a set of permissible scalar operations for ANSI DateTime and Interval values.

Scalar operations include:

Operation	Description
DateTime Expressions	Expressions providing a result that is a DateTime value. DateTime expressions have arguments that are also DateTime or Interval expressions.
Interval Expressions	Expressions providing a result that is an Interval. Interval expressions may include components that are Interval, DateTime, or Numeric expressions.

Data Type Compatibility

The database convention of performing implicit conversions to resolve expressions of mixed data types is not supported for operations that include ANSI DateTime or Interval values.

To convert ANSI DateTime or Interval expressions, use the CAST function. See *Teradata Vantage™ - Data Types and Literals*, B035-1143.

The following restrictions apply to the values appearing in all DateTime and Interval scalar operations:

IF ...	THEN ...
two DateTime values appear in the same DateTime expression	both values must use the same type. You cannot mix DATE, TIME, and TIMESTAMP values across type.
a DateTime and Interval values appear in the same DateTime expression	the Interval value must contain only DateTime fields that are also contained within the DateTime value.
two Interval values appear in the same Interval expression	both values must use the same type. You cannot mix Year-Month with Day-Time intervals.

ANSI DateTime Expressions

Perform a computation on a DATE, TIME, or TIMESTAMP value (or value expression) and return a single value of the same type.

A DateTime expression is any expression that returns a result that is a DATE, TIME, or TIMESTAMP value.

date_time_expression Syntax

```
{ date_time_term |
  interval_expression + date_time_term |
  date_time_expression {+|-} interval_term
}
```

date_time_term

```
date_time_primary [
  AT { LOCAL | [ TIME ZONE ] { expression | time_zone_string } }
]
```

date_time_expression

An expression that evaluates to a DATE, TIME, or TIMESTAMP value.

The form of the expression is one of the following:

- a single *date_time_term*.
- the sum of an *interval_expression* and a *date_time_term* expression.
- the sum or difference of a *date_time_expression* and an *interval_term*.

date_time_term

A single *date_time_primary* or a *date_time_primary* with a time zone specifier of AT LOCAL, AT [TIME ZONE] expression, or AT [TIME ZONE] *time_zone_string*.

interval_expression

One of the following:

- a single *interval_term*.
- an *interval_term* added to or subtracted from an *interval_expression*.
- the difference between a *date_time_expression* and a *date_time_term* (enclosed by parentheses) preceding a start TO end phrase.

date_time_primary

One of the following elements, any of which must have the appropriate DateTime type:

- Column reference
- DateTime literal value
- DateTime function reference

For example, the result of a CASE expression or CAST function or DateTime built-in function such as CURRENT_DATE or CURRENT_TIME.

- Scalar function reference
- Aggregate function reference
- (table_expression)

A scalar subquery.

- (date_time_timestamp_expression)

AT LOCAL

Use the default time zone displacement based on the current session time zone. The current session time zone may be specified as a time zone string or a time zone displacement expressed as an Interval data type that defines the local time zone offset.

AT [TIME ZONE] expression

Use the time zone displacement defined by expression.

The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE.

AT [TIME ZONE] *time_zone_string*

time_zone_string determines the time zone displacement.

AT LOCAL and AT TIME ZONE Time Zone Specifiers

A *date_time_primary* can include an AT LOCAL or AT [TIME ZONE] clause only if the *date_time_primary* evaluates to a TIME or TIMESTAMP value or is the built-in function CURRENT_DATE or DATE.

The effect is to adjust *date_time_term* to be in accordance with the specified time zone displacement.

The *expression* that specifies the time zone displacement in an AT [TIME ZONE] clause is implicitly converted, as needed and if allowed, to a time zone displacement or time zone string depending on its data type as defined in the following table.

Data type of <i>expression</i>	Implicit Conversion
INTERVAL HOUR(<i>n</i>) TO MINUTE where <i>n</i> is not 2	CAST(<i>expression</i> AS INTERVAL HOUR(2) TO MINUTE)
INTERVAL HOUR INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL HOUR INTERVAL HOUR TO SECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL SECOND	CAST(<i>expression</i> AS INTERVAL HOUR(2) TO MINUTE)
BYTEINT SMALLINT INTEGER BIGINT DECIMAL/NUMERIC if the fractional precision is 0	CAST(CAST(<i>expression</i> AS INTERVAL HOUR(2)) AS INTERVAL HOUR(2) TO MINUTE)
DECIMAL/NUMERIC if the fractional precision is greater than 0	CAST(CAST((<i>expression</i>)*60 AS INTERVAL MINUTE(4)) AS INTERVAL HOUR(2) TO MINUTE)
Character with CHARACTER SET UNICODE	CAST(CAST(<i>expression</i> AS INTERVAL HOUR(2)) AS INTERVAL HOUR(2) TO MINUTE) If an error occurs for the above CAST statement, Vantage attempts the following: CAST(<i>expression</i> AS INTERVAL HOUR(2) TO MINUTE)

Data type of expression	Implicit Conversion
	If an error occurs for this CAST statement also, Vantage treats the character value as a time zone string.
Character that is not CHARACTER SET UNICODE	TRANSLATE(<i>expression</i> USING <i>source_repertoire_name</i> _TO_Unicode) where <i>source_repertoire_name</i> is the server character set of <i>expression</i> . The translated value is then processed as above for a character value with CHARACTER SET UNICODE.
other	An error is returned.

Note:

There is a general restriction that in Numeric-to-Interval conversions, the INTERVAL type must have only one DateTime field. However, this restriction is not an issue when implicitly converting the *expression* of an AT clause because the conversion is done with two CAST statements.

If the conversion to INTERVAL HOUR(2) TO MINUTE results in a value that is not between INTERVAL '-12:59' HOUR TO MINUTE and INTERVAL '14:00' HOUR TO MINUTE, an error is returned.

You can specify two kinds of time zone strings in the AT [TIME ZONE] *time_zone_string* clause:

- Time zone strings that do not follow separate daylight savings time (DST) and standard time zone displacements from Coordinated Universal Time (UTC) time.
- Time zone strings that follow different DST and standard time zone displacements from UTC time.

The following time zone strings are supported.

Strings that do not follow separate DST and standard time zone displacements		
<ul style="list-style-type: none"> • 'GMT' • 'GMT+1' • 'GMT+10' • 'GMT+11' • 'GMT+11:30' • 'GMT+12' • 'GMT+13' • 'GMT+14' • 'GMT+2' • 'GMT+3' • 'GMT+3:30' 	<ul style="list-style-type: none"> • 'GMT+4' • 'GMT+4:30' • 'GMT+5' • 'GMT+5:30' • 'GMT+5:45' • 'GMT+6' • 'GMT+6:30' • 'GMT+7' • 'GMT+8' • 'GMT+8:45' • 'GMT+9' • 'GMT+9:30' 	<ul style="list-style-type: none"> • 'GMT-1' • 'GMT-10' • 'GMT-11' • 'GMT-2' • 'GMT-3' • 'GMT-4' • 'GMT-5' • 'GMT-6' • 'GMT-6:30' • 'GMT-7' • 'GMT-8'

Strings that follow different DST and standard time zone displacements		
<ul style="list-style-type: none"> • 'Africa Egypt' 	<ul style="list-style-type: none"> • 'Asia Gaza' 	<ul style="list-style-type: none"> • 'Australia Central'

Strings that follow different DST and standard time zone displacements

<ul style="list-style-type: none"> • 'Africa Morocco' • 'Africa Namibia' • 'America Alaska' • 'America Aleutian' • 'America Argentina' • 'America Atlantic' • 'America Brazil' • 'America Central' • 'America Chile' • 'America Cuba' • 'America Eastern' • 'America Mountain' • 'America Newfoundland' • 'America Pacific' • 'America Paraguay' • 'America Uruguay' 	<ul style="list-style-type: none"> • 'Asia Iran' • 'Asia Iraq' • 'Asia Irkutsk' • 'Asia Israel' • 'Asia Jordan' • 'Asia Kamchatka' • 'Asia Krasnoyarsk' • 'Asia Lebanon' • 'Asia Magadan' • 'Asia Omsk' • 'Asia Syria' • 'Asia Vladivostok' • 'Asia West Bank' • 'Asia Yakutsk' • 'Asia Yekaterinburg' 	<ul style="list-style-type: none"> • 'Australia Eastern' • 'Australia Western' • 'Europe Central' • 'Europe Eastern' • 'Europe Kaliningrad' • 'Europe Moscow' • 'Europe Samara' • 'Europe Western' • 'Indian Mauritius' • 'Mexico Central' • 'Mexico Northwest' • 'Mexico Pacific' • 'Pacific New Zealand' • 'Pacific Samoa'
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Vantage resolves the time zone string and calculates the time zone displacement for the session or requested query.

Note:

Vantage will automatically adjust the time zone displacement to account for the start or end of daylight saving time only if you specify a time zone using a time zone string that follows different DST and standard time zone displacements. GMT format strings represent time zone strings that follow only one standard time and does not have a separate daylight saving time. For example, the time zone string 'GMT+5:30' can be used for India in order to use the displacement interval 5:30, which is applicable all year around.

Vantage resolves the time zone string based on the rules and time zone displacement information stored in the system UDF (user-defined function), GetTimeZoneDisplacement.

If the time zone strings provided by Teradata do not meet your requirements, you may add new time zone strings or modify the existing time zone strings by modifying or adding new rules to the GetTimeZoneDisplacement UDF.

You can also use the AT clause to explicitly specify a time zone in the following cases:

- With the following built-in functions:
 - "CURRENT_DATE".
 - "CURRENT_TIME".
 - "CURRENT_TIMESTAMP".
 - "DATE".
 - "TIME".

Note:

If you specify these built-in functions with an AT LOCAL clause, the value returned depends on the setting of the DBS Control flag TimeDateWZControl.

- When converting DateTime data types using the CAST function or Teradata conversion syntax. You can specify the time zone used for the CAST or conversion as the source time zone, a specific time zone displacement or time zone string, or the current session time zone.
- With the EXTRACT function to specify a time zone for the source expression before extracting the fields.

Related Information

- For more information about setting session time zones, see SET TIME ZONE, CREATE USER, and MODIFY USER in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For more information about system time zone settings, see *Teradata Vantage™ - Database Utilities*, B035-1102.
- For more information about automatic adjustment of the system time to account for DST, see the information about the SDF file and Teradata Locale Definition Utility (tdlocaledef) in *Teradata Vantage™ - Database Utilities*, B035-1102.
- For details on DateTime literals, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For details, see [GetTimeZoneDisplacement](#).
- For more information about data type conversions, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information about time zones, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Gregorian Calendar Rules

DateTime expressions always operate within the rules of the Gregorian calendar.

When an evaluation results in a value outside the permissible range for any contained field or results in a value impermissible according to the natural rules for DATE and TIME values, then an error is returned.

For example, the following operation returns an error because it evaluates to a date that is not valid ('1996-09-31').

```
SELECT DATE '1996-08-31' + INTERVAL '1' MONTH;
```

The desired result is obtained with a slight rephrasing of the second operand.

```
SELECT DATE '1996-08-31' + INTERVAL '30' DAY;
```

This operation returns the desired result, '1996-09-30'. No error is returned.

Evaluation Types

Expressions involving DateTime values evaluate to a DateTime type, with DATE being the least significant type and TIMESTAMP the most significant.

DateTime expressions involving ...	Evaluate to a ...
Dates	date.
Times	time.
Timestamps	timestamp.

Adding and Subtracting Interval Values

DateTime expressions formed by adding an Interval to a DateTime value or by subtracting an Interval from a DateTime value are performed by adding or subtracting values of the appropriate component fields and carrying overflow from lower precision fields with the appropriate modulo to represent proper arithmetic in terms of the calendar and clock.

An *interval_expression* or *interval_term* may only contain DateTime fields that are contained in the corresponding *date_time_expression* or *date_time_term*.

When an Interval value is added to or subtracted from a TIME or TIMESTAMP value, the time zone displacement value associated with the result is identical to that associated with the TIME or TIMESTAMP value.

Computations With Time Zones

If you perform arithmetic on DateTime expressions containing time zones, the results are computed in the following way.

Call the DateTime value of the expression DV and the time zone value component (normalized to UTC) TZ.

The result is computed as $DV - TZ$.

Examples

Example: date_time_primary

In this example, the *date_time_primary* is a built-in time function.

```
SELECT CURRENT_TIME;
```


Example: `date_time_term` With an Integer Numeric Time Zone Specifier

In this example, the *date_time_primary* is a built-in time function and the time zone displacement is specified by the negative integer numeric value for Pacific Standard Time.

```
SELECT CURRENT_TIME AT TIME ZONE -8;
```

Example: `date_time_term` With a Scaled Decimal Time Zone Specifier

In this example, the *date_time_primary* is a built-in time function and the time zone displacement is specified by the scaled decimal value for Venezuela's time zone.

```
SELECT CURRENT_TIME AT TIME ZONE -4.5;
```

Example: `date_time_term` With an 'hh:mm' String Time Zone Specifier

In this example, the *date_time_primary* is a built-in time function and the time zone displacement is specified by the negative 'hh:mm' string for Venezuela's time zone.

```
SELECT CURRENT_TIME AT TIME ZONE -'4:30';
```

Example: `date_time_term` With an Interval Column Time Zone Specifier

In this example, the *date_time_term* is a *date_time_primary* column value named f1.

TS.f1 is a value of type TIME or TIMESTAMP and intrvl.a is a column interval value of type INTERVAL HOUR(2) TO MINUTE.

```
SELECT f1 AT TIME ZONE intrvl.a
FROM TS;
```

Example: `date_time_term` With an Interval Literal Time Zone Specifier

In this example, the *date_time_term* is a *date_time_primary* column value named f1.

The specified interval is an interval literal value of type INTERVAL HOUR TO MINUTE.

```
SELECT f1 AT TIME ZONE INTERVAL '01:00' HOUR TO MINUTE
FROM TS;
```

Example: `date_time_term` With a Time Zone String Time Zone Specifier

In this example, the *date_time_term* is a *date_time_primary* column value named `f1`.

`TS.f1` is a value of type `TIME` or `TIMESTAMP` and the time zone displacement is based on the time zone string `'America Pacific'`.

```
SELECT f1 AT TIME ZONE 'America Pacific'
FROM TS;
```

Example: `date_time_expression`

In this example, the *date_time_expression* is an *interval_expression* added to a *date_time_term*. Note that you can only add these terms—subtraction of a *date_time_term* from an *interval_expression* is not permitted.

```
SELECT INTERVAL '20' YEAR + CURRENT_DATE;
```

Example: `date_time_expression` With Addition

In this example, the *date_time_expression* comprises another *date_time_expression* added to an *interval_term*.

The columns *subscribe_date* and *subscription_interval* are typed `DATE` and `INTERVAL MONTH(4)`, respectively.

```
SUBSCRIBE_DATE + SUBSCRIPTION_INTERVAL
```

Example: `date_time_expression` With Subtraction

You can also subtract an *interval_term* from a *date_time_expression*.

In this example, an *interval_term* is subtracted from the *date_time_expression*.

The columns *expiration_date* and *subscription_interval* are typed `DATE` and `INTERVAL MONTH(4)`, respectively.

```
EXPIRATION_DATE - SUBSCRIPTION_INTERVAL
```

Time Zone Sort Order

Time zones are ordered chronologically, using the same time zone.

Examples

Consider the following examples using ordered SELECT statements on a table having a column with type `TIMESTAMP(0) WITH TIME ZONE`.

The identical ordering demonstrated in these ORDER BY SELECTs applies to all time zone comparison operations.

```
SELECT f1 TIMESTAMPFIELD
FROM timestwz
ORDER BY f1;
```

This statement returns the following results table.

```
TIMESTAMPFIELD
-----
1997-10-07 15:43:00+08:00
1997-10-07 15:43:00-00:00
1997-10-07 15:47:52-08:00
```

Note how the values are displayed with the stored time zone information, but that the ordering is not immediately evident.

Now note how normalizing the time zones by means of a CAST function indicates chronological ordering explicitly.

```
SELECT CAST(f1 AS TIMESTAMP(0)) TIMESTAMP_NORMALIZED
FROM timestwz
ORDER BY f1;
```

This statement returns the following results table.

```
TIMESTAMP_NORMALIZED
-----
1997-10-06 23:43:00
1997-10-07 07:43:00
1997-10-07 15:45:52
```

While the ordering is the same as for the previous query, the display of `TIMESTAMP` values has been normalized to the time zone in effect for the session, which is `'-08:00'`.

A different treatment of the time zones, this time to reflect local time, indicates the same chronological ordering but from a different perspective.

```
SELECT f1 AT LOCAL LOCALIZED
FROM timestwz
ORDER BY f1;
```

This statement returns the following results table.

```
LOCALIZED
-----
1997-10-06 23:43:00-08:00
1997-10-07 07:43:00-08:00
1997-10-07 15:45:52-08:00
```

Related Information

- For more information about setting session time zones, see SET TIME ZONE, CREATE USER, and MODIFY USER in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For more information about system time zone settings, see *Teradata Vantage™ - Database Utilities*, B035-1102.
- For more information about automatic adjustment of the system time to account for DST, see the information about the SDF file and Teradata Locale Definition Utility (tdlocaledef) in *Teradata Vantage™ - Database Utilities*, B035-1102.
- For details on DateTime literals, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For details, see [GetTimeZoneDisplacement](#).
- For more information about data type conversions, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information about time zones, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

ANSI Interval Expressions

An interval expression is any expression that returns a result that is an INTERVAL value. It performs a computation on an Interval value (or value expression) and returns a single value of the same type.

ANSI Interval Expression Syntax

```
{ interval_term |
  interval_expression {+|-} interval_term |
  ( date_time_expression [-] date_time_term ) start [ TO end ]
}
```

interval_term

```
{ {+|-} interval_primary |
  interval_term {*/}/} numeric_factor |
  numeric_term * interval_factor
}
```

start

```
{ YEAR | MONTH | DAY | HOUR | MINUTE | SECOND }
[ ( precision [, fractional_seconds_precision ] ) ]
```

end

```
{ MONTH | HOUR | MINUTE | SECOND } ( fractional_seconds_precision )
```

Syntax Elements***interval_term***

One of the following:

- A single *interval_factor*
- An *interval_term* multiplied or divided by a *numeric_factor*
- The product of a *numeric_term* and an *interval_factor*

interval_expression

One of the following:

- a single *interval_term*.
- an *interval_term* added to or subtracted from an *interval_expression*.
- the difference between a *date_time_expression* and a *date_time_term* (enclosed by parentheses) preceding a start TO end phrase.

date_time_expression

An expression that evaluates to a DATE, TIME, or TIMESTAMP value.

The form of the expression is one of the following:

- a single *date_time_term*.
- the sum of an *interval_expression* and a *date_time_term* expression.
- the sum or difference of a *date_time_expression* and an *interval_term*.

start

Defines the beginning of a date or time interval.

If you specify MONTH or SECOND, you must omit TO *end*.

end

[Optional] Defines the end of a date or time interval.

The value for *end* must be less significant than the value for *start*.

If *start* is a YEAR value, then *end* must be a MONTH value.

interval_primary

One of the following elements, any of which must have the appropriate INTERVAL type:

- Column reference
- Interval literal value

For details on interval literals, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

- Scalar function reference
- Aggregate function reference
- (*table_expression*)
- Scalar subquery: (*interval_expression*)

numeric_factor

A signed *numeric_primary*.

numeric_term

A *numeric_factor* or a *numeric_term* multiplied or divided by a *numeric_factor*.

interval_factor

A signed *interval_primary*.

precision

[Optional] Specifies the permitted range of digits, in range [0, 4].

Default: 2

fractional_seconds_precision

[Optional] Specifies the fractional precision for values of SECOND, in range [0, 6].

Default: 6

numeric_primary

One of the following elements, any of which must have the appropriate numeric type:

- Column reference
- Numeric literal value
- Scalar function reference
- Aggregate function reference
- (*table_expression*)
- Scalar subquery: (*numeric_expression*)

Usage Notes

Rules

The following rules apply to Interval expressions.

- Expressions involving intervals are evaluated by converting the operands to integers, evaluating the resulting arithmetic expression, and then converting the result back to the appropriate interval.
- The data type of both an *interval_expression* and an *interval_primary* is INTERVAL.
- An *interval_expression* must contain either year-month interval components or day-time interval components. Mixing of INTERVAL types is not permitted.
- Expressions involving intervals always evaluate to an interval, even if the expressions contain DateTime or Numeric expressions.

IF an <i>interval_expression</i> contains ...	THEN the result ...
only one component of type INTERVAL	is of the same INTERVAL type.
a single DateTime value or a <i>start</i> TO <i>end</i> phrase	contains the DateTime fields specified for the DateTime or <i>start</i> TO <i>end</i> phrase values.
more than one component of type INTERVAL	is of an INTERVAL type including all the DateTime fields of the INTERVAL types of the component fields.

Normalization of Intervals with Multiple Fields

Because of the way the Parser normalizes multiple field INTERVAL values, the defined precision for an INTERVAL value may not be large enough to contain the value once it has been normalized.

For example, inserting a value of '99-12' into a column defined as INTERVAL YEAR(2) TO MONTH causes an overflow error because the Parser normalizes the value to '100-00'. When an attempt is made to insert that value into a column defined to have a 2-digit YEAR field, it fails because it is a 3-digit year.

Here is an example that returns an overflow error because it violates the permissible range values for the type.

First define the table.

```
CREATE TABLE BillDateTime
(column_1 INTERVAL YEAR
,column_2 INTERVAL YEAR(1) TO MONTH
,column_3 INTERVAL YEAR(2) TO MONTH
,column_4 INTERVAL YEAR(3) TO MONTH );
```

Now insert the value INTERVAL '999-12' YEAR TO MONTH using this INSERT statement.

```
INSERT BillDateTime (column_1, column_4)
VALUES ( INTERVAL '40' YEAR, INTERVAL '999-12' YEAR TO MONTH );
```

The result is an overflow error because the valid range for INTERVAL YEAR(3) TO MONTH values is '-999-11' to '999-11'.

You might expect the value '999-12' to work, but it fails because the Parser normalizes it to a value of '1000-00' YEAR TO MONTH. Because the value for year is then four digits, an overflow occurs and the operation fails.

Examples

Examples of Interval Expression Components and Their Processing

The following examples illustrate the components of an interval expression and describe how those components are processed.

Example of interval_term

The definition for *interval_term* can be expressed in four forms:

- *interval_factor*
- *interval_term* * *numeric_factor*
- *interval_term* / *numeric_factor*
- *numeric_term* * *interval_factor*

This example uses the second definition.

```
SELECT (INTERVAL '3-07' YEAR TO MONTH) * 4;
```

The *interval_term* in this operation is INTERVAL '3-07' YEAR TO MONTH.

The *numeric_factor* is 4.

The processing involves the following stages:

1. The interval is converted into 43 months as an INTEGER value.
2. The INTEGER value is multiplied by 4, giving the result 172 months.
3. The result is converted to '14-4'.

Example of numeric_factor

This example uses a *numeric_factor* with an INTERVAL YEAR TO MONTH typed value.

```
SELECT INTERVAL '10-02' YEAR TO MONTH * 12/5;
```

The *numeric_factor* in this operation is the integer 12.

The processing involves the following stages:

1. The interval is multiplied by 12, giving the result as an interval.
2. The interval result is divided by 5, giving '24-04'.

Note that very different results are obtained by using parentheses to change the order of evaluation as follows.

```
SELECT INTERVAL '10-02' YEAR TO MONTH * (12/5);
```

The *numeric_factor* in this operation is (12/5).

The processing involves the following stages:

1. The *numeric_factor* is computed, giving the result 2.4, which is truncated to 2 because the value is an integer by default.
2. The interval is multiplied by 2, giving '20-04'.

Example of interval_term / numeric_factor

The following example uses an *interval_term* value divided by a *numeric_factor* value.

```
SELECT INTERVAL '10-03' YEAR TO MONTH / 3;
```

The *interval_term* is INTERVAL '10-03' YEAR TO MONTH.

The *numeric_factor* is 3.

The processing involves the following stages:

1. The interval value is decomposed into a value of months.
Ten years and three months evaluate to 123 months.

2. The interval total is divided by the *numeric_factor* 3, giving '3-05'.

The next example is similar to the first except that it shows how truncation is used in integer arithmetic.

```
SELECT INTERVAL '10-02' YEAR TO MONTH / 3;
```

The *interval_term* is INTERVAL '10-02' YEAR TO MONTH.

The *numeric_factor* is 3.

The processing involves the following stages:

1. The interval value is decomposed into a value of months.
Ten years and two months evaluate to 122 months.
2. The interval total is divided by the *numeric_factor* 3, giving 40.67 months, which is truncated to 40 because the value is an integer.
3. The interval total is converted back to the appropriate format, giving INTERVAL '3-04'.

Example of *numeric_term* * *interval_primary*

In this format, the value for *numeric_term* can include instances of multiplication and division.

```
SELECT 12/5 * INTERVAL '10-02' YEAR TO MONTH;
```

The *numeric_term* is 12/5.

The *interval_primary* is INTERVAL '10-02' YEAR TO MONTH.

The processing involves the following stages:

1. The *numeric_term* 12/5 is evaluated, giving 2.4, which is truncated to 2 because the value is an integer by default.
2. The *interval_primary* is multiplied by 2, giving '20-04'.

Example of *numeric_term* * \pm *interval_primary*

This example multiplies a negative *interval_primary* by a *numeric_term* and adds the negative result to an *interval_term*.

```
SELECT (RACE_DURATION + (2 * INTERVAL -'30' DAY));
```

The *numeric_term* in this case is the *numeric_primary* 2.

The *interval_primary* is INTERVAL -'30' DAY.

RACE_DURATION is an *interval_term*, with type INTERVAL DAY TO SECOND.

The processing involves the following stages:

1. The *interval_primary* is converted to an exact numeric, or 60 days.
2. The operations indicated in the arithmetic are performed on the operands (which are both numeric at this point), producing an exact numeric result having the appropriate scale and precision.

In this example, 60 days are subtracted from RACE_DURATION, which is an INTERVAL type of INTERVAL DAY TO SECOND.

3. The numeric result is converted back into the indicated INTERVAL type, DAY TO SECOND.

Example of interval_expression

The definition for *interval_expression* can be expressed in three forms:

- *interval_term*
- *interval_expression* + *interval_term*
- (*date_time_expression* - *date_time_term*) *start* TO *end*

This example uses the second definition.

```
SELECT (CAST(INTERVAL '125' MONTH AS INTERVAL YEAR(2) TO MONTH))
+ INTERVAL '12' YEAR;
```

The *interval_expression* is INTERVAL '125' MONTH.

The *interval_term* is INTERVAL '12' YEAR.

The processing involves the following stages:

1. The CAST function converts the *interval_expression* value of 125 months to 10 years and 5 months.
2. The *interval_term* amount of 12 years is added to the *interval_expression* amount, giving 22 years and 5 months.
3. The result is converted to the appropriate data type, which is INTERVAL YEAR(2) TO MONTH, giving '22-05'.

This example uses the third definition for *interval_expression*.

You must ensure that the values for *date_time_expression* and *date_time_term* are comparable.

```
SELECT (TIME '23:59:59.99' - CURRENT_TIME(2)) HOUR(2) TO SECOND(2);
```

The *date_time_expression* is TIME '23:59:59.99'.

The *date_term* is the *date_time_primary* - CURRENT_TIME(2).

The processing involves the following stages:

1. Assume that the current system time is 18:35:37.83.
2. The HOUR(2) TO SECOND(2) time interval 18:35:37.83 is subtracted from the TIME value 23:59:59.99, giving the result '5:24:22.16'.

Here is another example that uses the third definition for *interval_expression* to find the difference in minutes between two `TIMESTAMP` values. First define a table:

```
CREATE TABLE BillDateTime
(start_time TIMESTAMP(0)
,end_time TIMESTAMP(0));
```

Now, determine the difference in minutes:

```
SELECT (end_time - start_time) MINUTE(4)
FROM BillDateTime;
```

The processing involves the following stages:

1. The `start_time` `TIMESTAMP` value is subtracted from the `end_time` `TIMESTAMP` value, giving an interval result.
2. The `MINUTE(4)` specifies an interval unit of minutes with a precision of four digits, which allows for a maximum of 9999 minutes, or approximately one week.

Arithmetic Operators and ANSI DateTime and Interval Data Types

Operations on ANSI DateTime and Interval values can include the scalar arithmetic operators `+`, `-`, `*`, and `/`. However, the operators are only valid on specific combinations of DateTime and Interval values.

Arithmetic Operators and Result Types

The following arithmetic operations are permitted for DateTime and Interval data types.

First Value Type	Operator	Second Value Type	Result Type
DateTime	-	DateTime	Interval
DateTime	+	Interval	DateTime
DateTime	-	Interval	DateTime
Interval	+	DateTime	DateTime
Interval	+	Interval	Interval
Interval	-	Interval	Interval
Interval	*	Number	Interval
Interval	/	Number	Interval
Number	*	Interval	Interval

Adding or Subtracting Numbers from DATE

Teradata SQL extends the ANSI SQL:2011 standard to allow the operations of adding or subtracting a number of days from an ANSI DATE value.

Teradata SQL treats the number as an INTERVAL DAY value.

Calculating the Difference Between Two DateTime Values

Vantage calculates the interval difference between two DATE, TIME or TIMESTAMP values according to the ANSI SQL standard. Units smaller than the unit of the result are ignored when calculating the interval value.

For example, when computing the difference in months for two DATE values, the day values in each of the two operands are ignored. Similarly when computing the difference in hours for two TIMESTAMP values, the minutes and the seconds values of the operands are ignored.

Examples

Example: Calculating the Difference in Days Between DATE Values

The following query calculates the difference in days between the two DATE values.

```
SELECT (DATE '2007-05-10' - DATE '2007-04-28') DAY;
```

The result is the following:

```
(2007-05-10 - 2007-04-28) DAY
-----
12
```

The following query calculates the difference in months between the two DATE values.

```
SELECT (DATE '2007-05-10' - DATE '2007-04-28') MONTH;
```

The result is the following:

```
(2007-05-10 - 2007-04-28) MONTH
-----
1
```

There is a difference of 12 days between the two dates, which does not constitute one month. However, Vantage ignores the day values during the calculation and only considers the month values, so the result is an interval of one month indicating the difference between April and May.

Example: Adding Interval to DATE

The following example adds an Interval value to a DateTime value:

```
CREATE TABLE Subscription
(id CHARACTER(13)
,subscribe_date DATE
,subscribe_interval INTERVAL MONTH(4));

INSERT Subscription (subscribe_date, subscribe_interval)
VALUES (CURRENT_DATE, INTERVAL '24' MONTH);

SELECT subscribe_date + subscribe_interval FROM Subscription;
```

The result is a DateTime value.

Related Information

For more information, see [DATE and Date Arithmetic](#).

Aggregate Functions and ANSI DateTime and Interval Data Types

The following aggregate functions are valid for ANSI SQL:2011 DateTime types.

For this function ...	The result is ...
AVG(<i>arg</i>)	the type of the argument.
MAX(<i>arg</i>)	the type of the argument, based on the comparison rules for DateTime types.
MIN(<i>arg</i>)	
COUNT(<i>arg</i>)	INTEGER, if the mode is Teradata. DECIMAL(<i>n</i> ,0), if the mode is ANSI, where: <ul style="list-style-type: none"> • <i>n</i> is 15 if MaxDecimal in DBSControl is 0, 15, or 18. • <i>n</i> is 38 if MaxDecimal in DBSControl is 38.

For more information about these functions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Interval Data Types

The following aggregate functions are valid for Interval types.

For this function ...	The result is ...
AVG(<i>arg</i>)	the type of the argument.
COUNT(<i>arg</i>)	INTEGER, if the mode is Teradata.
	DECIMAL(<i>n</i> ,0), if the mode is ANSI, where: <ul style="list-style-type: none"> • <i>n</i> is 15 if MaxDecimal in DBSControl is 0, 15, or 18. • <i>n</i> is 38 if MaxDecimal in DBSControl is 38.
MAX(<i>arg</i>)	the type of the argument, based on the comparison rules for DateTime types.
MIN(<i>arg</i>)	
SUM(<i>arg</i>)	the type of the argument.

For more information about these functions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Scalar Operations and DateTime Functions

DateTime functions are those functions that operate on either DateTime or Interval values and provide a DateTime value as a result.

The supported DateTime functions are:

- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- EXTRACT

To avoid any synchronization problems, operations among any of these functions are guaranteed to use identical definitions for DATE, TIME, or TIMESTAMP so that the following are always true:

- CURRENT_DATE = CURRENT_DATE
- CURRENT_TIME = CURRENT_TIME
- CURRENT_TIMESTAMP = CURRENT_TIMESTAMP
- CURRENT_DATE and CURRENT_TIMESTAMP always identify the same DATE
- CURRENT_TIME and CURRENT_TIMESTAMP always identify the same TIME

The values reflect the time when the request started and do not change during the duration of the request.

Example: Using the CURRENT_DATE DateTime Function

The following example uses the CURRENT_DATE DateTime function:

```
SELECT INTERVAL '20' YEAR + CURRENT_DATE;
```

Related Information

For more information, see:

- For more information about CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.
- For more information about EXTRACT, see [EXTRACT](#).

Teradata Date and Time Expressions

Teradata SQL provides a data type for DATE values and stores TIME values as encoded numbers with type REAL. This is a Teradata extension of the ANSI SQL:2011 standard and its use is strongly deprecated.

Since both DATE and TIME are encoded values, not simple integers or real numbers, arithmetic operations on these values are restricted.

ANSI DATE and TIME values are stored using appropriate DateTime types and have their own set of rules for DateTime assignment and expressions. For information, see [ANSI DateTime and Interval Data Type Assignment Rules](#) and [Scalar Operations on ANSI SQL:2011 DateTime and Interval Values](#).

DATE and Integer Arithmetic

The following arithmetic functions can be performed with date and an integer (INTEGER is interpreted as a number of days):

- DATE + INTEGER
- INTEGER + DATE
- DATE - INTEGER

These expressions are not processed as simple addition or subtraction, but rather as explained in the following process:

1. The encoded date value is converted to an intermediate value which is the number of days since some system-defined fixed date.
2. The integer value is then added or subtracted, forming another value as number of days, since the fixed base date.
3. The result is converted back to a date, valid in the Gregorian calendar.

DATE and Date Arithmetic

The DATE - DATE expression is not processed as a simple subtraction, but rather as explained in the following process:

1. The encoded date values are converted to intermediate values which are each the number of days since a system-defined fixed date.

- The second of these values is then subtracted from the first, giving the number of days between the two dates.
- The result is returned as if it were in the ANSI SQL:2011 form INTERVAL DAY, though the value itself is an integer.

Other arithmetic operations on date values may provide results, but those results are not meaningful.

Example: DATE/2 Integer Result

DATE/2 provides an integer result, but the value has no meaning.

There are no simple arithmetic operations that have meaning for time values. The reason is that a time value is simply a real number with time encoded as:

```
(HOUR*10000 + MINUTE*100 + SECOND)
```

where SECOND may include a fractional value.

Scalar Operations on Teradata DATE Values

The operations of addition and subtraction are allowed as follows, where integer values represent the number of days:

Argument 1	Operation	Argument 2	Result
DATE	+	INTEGER	DATE
DATE	-	INTEGER	DATE
INTEGER	+	DATE	DATE
DATE	-	DATE	INTEGER

Adding 90 days, for example, is not identical to adding 3 months, because of the varying number of days in months.

Also, adding multiples of 365 days is not identical to adding years because of leap years.

Note that scalar operations on Teradata DATE expressions are performed using ANSI SQL:2011 data types, so an expression of the type *date_expression - numeric_expression* is treated as if the *numeric_expression* component were typed as INTERVAL DAY.

ANSI SQL:2011 DateTime and Interval values have their own set of scalar operations and with the exception of the scalar operations defined here for DATE, do not support the implicit conversions to resolve expressions of mixed data types.

ADD_MONTHS Function

The ADD_MONTHS function provides for adding or subtracting months or years, handling the variable number of days involved. For details, see [ADD_MONTHS](#).

EXTRACT Function

Use the EXTRACT function to get the year, month, or day from a date. The result has INTEGER data type. For details, see [EXTRACT](#).

YEAR/MONTH/DAYOFMONTH/HOUR/MINUTE/SECOND

Returns the year, month, day of month, hour, minute, and second field from any DateTime or Interval value, converting it to an exact numeric.

ANSI Compliance

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

Result Type

For information about result type and attributes, see [EXTRACT](#).

YEAR/MONTH/DAYOFMONTH/HOUR/MINUTE/SECOND Syntax

```
{ YEAR | MONTH | DAYOFMONTH | HOUR | MINUTE | SECOND } ( expression )
```

Syntax Elements

expression

The expression that results in a DateTime, Interval, or UDT value.

For information about argument types, see [EXTRACT](#).

Examples

Usage examples:

```
SELECT HOUR(CURRENT_TIMESTAMP(6));
```

```
SELECT MINUTE(CURTIME());
```

```
SELECT SECOND('12:34:25');
```

```

sel dayofmonth(current_timestamp);
sel dayofmonth(date '2017-08-21');
sel dayofmonth(timestamp '2017-08-21 16:48:27.770000+00:00');
sel dayofmonth('2017-08-21 16:48:27.770000+00:00' (timestamp));
sel dayofmonth(date '1900-01-01');
sel dayofmonth(date '2018-02-22' - interval '300' year);
sel dayofmonth(date '2018-02-22' - interval '6' month);
sel dayofmonth(date '2018-02-22' - interval '15' day);

```

```

sel week(date);
sel week(current_date);
sel week(date '1900-01-01');
sel week(date '1700-05-31');
sel week(date - interval '300' year);
sel week(date - interval '6' month);
sel week(date - interval '150' day);

```

```

sel month(date);
sel month(current_date);
sel month(date '1900-01-01');
sel month(date '1899-12-31');
sel month(date '1700-05-31');
sel month(date - interval '300' year);
sel month(date - interval '6' month);
sel month(date - interval '150' day);

```

```

sel year(date);
sel year(current_date);
sel year(date '1900-01-01');
sel yearmonth(date '1899-12-31');
sel year(date - interval '300' year);
sel year(date - interval '6' month);

```

WEEK

Returns the number of weeks from the beginning of the year to the specified date.

ANSI Compliance

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

Result Type

The result is an integer value from 1 through 53, where 1 is the first week of the year and 53 is the last week of the year.

WEEK Syntax

```
WEEK ( expression )
```

Syntax Elements***expression***

The expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Examples: SELECT WEEK

Examples of SELECT WEEK usage:

```
select week(date);
select week(current_date);
select week(date '1900-01-01');
select week(date '1899-12-31');
select week(date '1700-05-31');
select week(date - interval '300' year);
select week(date - interval '6' month);
select week(date - interval '150' day);
```

LAST_DAY

Returns the date of the last day of month that contains *date_timestamp_value*.

ANSI Compliance

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

Result Type

The return data type is DATE.

LAST_DAY Syntax

```
[TD_SYSFNLIB.] LAST_DAY ( date_timestamp_value )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

date_timestamp_value

A date or timestamp argument.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Since TIMESTAMP values are stored in UTC time within the database and lack a time zone, the session time zone is used to interpret the time stamp value within the function. For TIMESTAMP WITH TIME ZONE values, the time zone component is used to interpret the time stamp value within the function.

Examples

Example: Querying for DATE for the Last Day of the Month

The following query:

```
SELECT LAST_DAY(DATE '2009-12-20');
```

returns the result 09/12/31.

Example: Querying for TIMESTAMP for the Last Day of the Month

The following query:

```
SELECT LAST_DAY(TIMESTAMP '2009-08-25 10:14:59');
```

returns the result 09/08/31.

Example: Querying for TIMESTAMP WITH TIME ZONE for the Last Day of the Month

The following query:

```
SELECT LAST_DAY(TIMESTAMP '2009-06-15 10:14:59-8:00');
```

returns the result 09/06/30.

Related Information

For details, see the information about CAST in explicit data type conversions in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

NEXT_DAY

Returns the date of the first weekday (*day_value*) that is later than the date specified by *date_timestamp_value*.

ANSI Compliance

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

Result Type

The return data type is DATE.

NEXT_DAY Syntax

```
[TD_SYSFNLIB.] NEXT_DAY ( date_timestamp_value, day_value )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

date_timestamp_value

A date or timestamp argument.

day_value

A character argument.

day_value can be any day of the week or its 3-character abbreviation.

Valid values are:

- 'SUNDAY' or 'SUN'
- 'MONDAY' or 'MON'
- 'TUESDAY' or 'TUE'
- 'WEDNESDAY' or 'WED'
- 'THURSDAY' or 'THU'
- 'FRIDAY' or 'FRI'

- 'SATURDAY' or 'SAT'

The day of week is not case-sensitive. Any characters that follow a valid abbreviation are ignored.

If this syntax element is NULL, NULL is returned.

Note:

While all systems are shipped only supporting the above English values for *day_value*, the Database Administrator can extend valid values to properly handle localized day names by including them in the ShortDays or LongDays of the Specification for Data Formatting (SDF) file supplied to the tdlocaledef utility.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *date_timestamp_value* = DATE or TIMESTAMP(0)
- *day_value* = VARCHAR

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Since TIMESTAMP values are stored in UTC time within the database and lack a time zone, the session time zone is used to interpret the time stamp value within the function. For TIMESTAMP WITH TIME ZONE values, the time zone component is used to interpret the time stamp value within the function.

Example

Example: Querying for the First Tuesday After a Specified Date

The following query:

```
SELECT NEXT_DAY(DATE '2009-12-20', 'TUESDAY');
```

returns the result 09/12/22 since 12/22 is the next Tuesday after Sunday, Dec 20, 2009.

Example: Querying for the First Friday After a Specified Date

The following query:

```
SELECT NEXT_DAY(DATE '2009-12-20', 'FRIDAY');
```

returns the result 09/12/25 since 12/25 is the next Friday after Sunday, Dec 20, 2009.

Example: Querying for the First Friday After a Different Specified Date

The following query:

```
SELECT NEXT_DAY(DATE '2009-12-25', 'FRIDAY');
```

returns the result 10/01/01 since Jan 1, 2010 is the next Friday after Friday, Dec 25, 2009.

Example: Non-English day_values

NEXT_DAY (*date_timestamp_value*, *day_value*) works with the Specification for Data Formatting. This is an example of the Japanese tdlocaledef.txt and how NEXT_DAY works with those definitions.

```
// DBS System Formatting Data
// Day and month names
ShortDays {
"\u65e5";
"\u6708";
"\u706b";
"\u6c34";
"\u6728";
"\u91d1";
"\u571f"
}
LongDays {
"\u65e5\u66dc\u65e5";
"\u6708\u66dc\u65e5";
"\u706b\u66dc\u65e5";      /*火曜日*/
"\u6c34\u66dc\u65e5";
"\u6728\u66dc\u65e5";
"\u91d1\u66dc\u65e5";
"\u571f\u66dc\u65e5"
}
ShortMonths {
"1\u6708";
"2\u6708";
"3\u6708";
```



```
"4\u6708";
"5\u6708";
"6\u6708";
"7\u6708";
"8\u6708";
"9\u6708";
"10\u6708";
"11\u6708";
"12\u6708"
}
```

The following SQL fails with an error:

```
select next_day(to_date('2015/06/22','YYYY/MM/DD'),'Tue') AS "日付";
```

Instead, use the Japanese day_value defined in the tdlocaledef.txt as follows:

```
select next_day(to_date('2015/06/22','YYYY/MM/DD'),'火曜日') AS "日付" ;
日付
-----
15/06/23
```

Note:

"\u706b\u66dc\u65e5"; indicates "Tuesday" in Japanese as demonstrated below.

```
Se1 _UNICODE U& '#706b#66dc#65e5' UESCAPE '#';
' 火曜日 '
-----
火曜日
```

MONTHS_BETWEEN

Returns the number of months between *date_timestamp_value1* and *date_timestamp_value2*.

ANSI Compliance

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

Result Type

The return data type is NUMBER.

MONTHS_BETWEEN Syntax

```
[TD_SYSFNLIB.] MONTHS_BETWEEN ( date_timestamp_value1, date_timestamp_value2 )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

date_timestamp_value1

A date or timestamp argument.

date_timestamp_value2

A date or timestamp argument.

If *date_timestamp_value2* is NULL, NULL is returned.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE

The two argument to MONTHS_BETWEEN can be one of the above three data types, but the two arguments must be the same data type. For example, you cannot pass a DATE as the first parameter and a TIMESTAMP as the second parameter.

Since TIMESTAMP values are stored in UTC time within the database and lack a time zone, the session time zone is used to interpret the time stamp value within the function. For TIMESTAMP WITH TIME ZONE values, the time zone component is used to interpret the time stamp value within the function.

If *date_timestamp_value2* is:

- Later than *date_timestamp_value1*, the result is negative.
- Earlier than *date_timestamp_value1*, the result is positive.

If *date_timestamp_value1* and *date_timestamp_value2* are either the same days of the month or both last days of months, the result is always an integer. Otherwise, MONTHS_BETWEEN calculates the fractional portion of the result based on a 31-day month and considers the difference in time components of *date_timestamp_value1* and *date_timestamp_value2*. The Gregorian calendar is supported.

Timestamps without a time zone are compared in the local time zone. This matters solely for the purpose of determining if two timestamps reside on the same day or are both the last day of a month. Timestamps with a time zone are converted to UTC before being compared in order to ensure the times are in the same zone.

Examples

Example: Querying for the Number of Months Between Two Dates

The following query:

```
SELECT MONTHS_BETWEEN(DATE '1995-02-02', DATE '1995-01-01');
```

returns the result 1.03225806451612903225806451612903225806. The MONTHS_BETWEEN function calculates the fractional portion of the result based on a 31-day month and considers the difference in time components of *date_timestamp_value1* and *date_timestamp_value2*.

Example: Querying for the Number of Months for One Date

The following query:

```
SELECT MONTHS_BETWEEN(DATE '2008-09-25', DATE '2008-09-25');
```

returns the result 0.0 since *date_timestamp_value1* and *date_timestamp_value2* are both the same date.

ADD_MONTHS

Adds an integer number of months to a DATE or TIMESTAMP expression and normalizes the result.

ANSI Compliance

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

ADD_MONTHS Syntax

```
[TD_SYSFNLIB.] ADD_MONTHS (
  { date_expression | timestamp_expression },
  integer_expression
)
```

Syntax Elements

date_expression

timestamp_expression

integer_expression

Usage Notes

Rules

ADD_MONTHS observes the following rules:

- If either argument of ADD_MONTHS is NULL, then the result is NULL.
- If the result is not in the range '0000-01-01' to '9999-12-31', then an error is reported.
- Results of an ADD_MONTHS function that are not valid dates are normalized to ensure that all reported dates are valid.

Support for UDTs

Vantage performs implicit type conversion if the UDT has an implicit cast that casts between the UDT and any of the following predefined types.

UDT Argument	Predefined Type
<i>date_expression</i>	<ul style="list-style-type: none"> • Character • Date • Timestamp
<i>timestamp_expression</i>	
<i>integer_expression</i>	Numeric

See the information about implicit type conversions in *Teradata Vantage™ - Data Types and Literals*, B035-1143. To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including ADD_MONTHS, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

Scalar Arithmetic on Months Issues

Consistent handling of a target month having fewer days than the month in the source date is an important issue for scalar arithmetic on month intervals because the concept of a month has no fixed definition.

All scalar function operations on dates use the Gregorian calendar. Peculiarities of the Gregorian calendar ensure that arithmetic operations such as adding 90 days (to represent three months) or 730 days (to

represent two years) to a DATE value generally do not provide the desired result. For more information, see [Gregorian Calendar Rules](#).

The ADD_MONTHS function uses an algorithm that lets you add or subtract a number of months to a *date_expression* or *timestamp_expression* and to obtain consistently valid results.

When deciding whether to use the Teradata SQL ADD_MONTHS function or ANSI SQL:2011 DateTime interval arithmetic, you are occasionally faced with choosing between returning a result that is valid, but probably neither desired nor expected, or not returning any result and receiving an error message.

A third option that does not rely on system-defined functions is to use the Teradata-defined Calendar view for date arithmetic.

Normalization Behavior of ADD_MONTHS

The standard approach to interval month arithmetic is to increment MONTH and YEAR values as appropriate and retain the source value for DAY. This is a problem for the case when the target DAY value is smaller than the source DAY value from the source date.

For example, what approach should be taken to handle the result of adding one MONTH to a source DATE value of '1999-01-31'? Using the standard approach, the answer would be '1999-02-31', but February 31 is not a valid date.

The behavior of ADD_MONTHS is equivalent to that of the ANSI SQL:2011 compliant operations DATE \pm INTERVAL 'n' MONTH and TIMESTAMP \pm INTERVAL 'n' MONTH with one important difference.

The difference between these two scalar arithmetic operations is their behavior when a non valid date value is returned by the function.

- ANSI SQL:2011 arithmetic returns an error.
- ADD_MONTHS arithmetic makes normative adjustments and returns a valid date.

Definition of Normalization

The normalization process is explained more formally as follows.

When the DAY field of the source *date_expression* or *timestamp_expression* is greater than the resulting target DAY field, ADD_MONTHS sets DD equal to the last day of the month + *n* to normalize the reported date or timestamp.

Define *date_expression* as 'YYYY-MM-DD' for simplicity.

For a given *date_expression*, you can then express the syntax of ADD_MONTHS as follows.

```
ADD_MONTHS('YYYY-MM-DD' , n)
```

Recalling that *n* can be negative, and substituting 'YYYY-MM-DD' for *date_expression*, you can redefine ADD_MONTHS in terms of ANSI SQL:2011 dates and intervals as follows.

```
ADD_MONTHS('YYYY-MM-DD', n) = 'YYYY-MM-DD' ± INTERVAL 'n' MONTH
```

The equation is true unless a non valid date such as 1999-09-31 results, in which case the ANSI expression traps the non valid date exception and returns an error.

ADD_MONTHS, on the other hand, processes the exception and returns a valid, though not necessarily expected, date. The algorithm ADD_MONTHS uses to produce its normalized result is as follows, expressed as pseudocode.

```
WHEN
DD > last_day_of_the_month(MM+n)
THEN SET
DD = last_day_of_the_month(MM+n)
```

This property is also true for the date portion of any *timestamp_expression*.

Note that normalization produces valid results for leap years.

Non-Intuitive Results of ADD_MONTHS

Because of the normalization made by ADD_MONTHS, many results of the function are not intuitive, and their inversions are not always symmetrical. For example, compare the results of [Example: Querying for the Current Date Plus One Month](#) with the results of [Example: Querying for the Month Prior to the Current Date](#).

This is because the function always produces a valid date, but not necessarily an *expected* date. Correctness in the case of interval month arithmetic is a relative term. Any definition is arbitrary and cannot be generalized, so the word 'expected' is a better choice for describing the behavior of ADD_MONTHS.

The following SELECT statements return dates that are both valid and expected:

```
SELECT ADD_MONTHS ('1999-08-15' , 1);
```

This statement returns 1999-09-15.

```
SELECT ADD_MONTHS ('1999-09-30' , -1);
```

This statement returns 1999-08-30.

The following SELECT statement returns a valid date, but its 'correctness' depends on how you choose to define the value 'one month.'

```
SELECT ADD_MONTHS ('1999-08-31' , 1);
```

This statement returns 1999-09-30, because September has only 30 days and the non-normalized answer of 1999-09-31 is not a valid date.

ADD_MONTHS Summarized

ADD_MONTHS returns a new *date_expression* or *timestamp_expression* with YEAR and MONTH fields adjusted to provide a correct date, but a DAY field adjusted only to guarantee a valid date, which might not be a date you expect intuitively.

If this behavior is not acceptable for your application, use ANSI SQL:2011 DateTime interval arithmetic instead. For more information, see [ANSI Interval Expressions](#).

Remember that ADD_MONTHS changes the DAY value of the result *only* when a non valid *date_expression* or *timestamp_expression* would otherwise be reported.

Examples

Example: Intuitive Examples

The examples are simple, intuitive examples of the ADD_MONTHS function. All results are both valid and expected.

Example: Querying the Number of Months in Years to a DATE Expression

This statement returns the current date plus 13 years.

```
SELECT ADD_MONTHS (CURRENT_DATE, 12*13);
```

Example: Querying for the Current Date Plus Six Months

This statement returns the date 6 months ago.

```
SELECT ADD_MONTHS (CURRENT_DATE, -6);
```

Example: Querying for the Current Date Plus Four Months

This statement returns the current TIMESTAMP plus four months.

```
SELECT ADD_MONTHS (CURRENT_TIMESTAMP, 4);
```

Example: Querying for the Current Date Plus Nine Months

This statement returns the TIMESTAMP nine months from January 1, 1999. Note the literal form, which includes the keyword TIMESTAMP.

```
SELECT ADD_MONTHS (TIMESTAMP '1999-01-01 23:59:59', 9);
```

Example: Querying for the Current Date Plus One Month

This statement adds one month to January 30, 1999.

```
SELECT ADD_MONTHS ('1999-01-30', 1);
```

The result is 1999-02-28.

Example: Non-Intuitive Examples

The following examples illustrate how the results of an ADD_MONTHS function are not always what you might expect them to be when the value for DAY in *date_expression* or the date component of *timestamp_expression* is 29, 30, or 31.

All examples use a *date_expression* for simplicity. In every case, the function behaves as designed.

Example: Querying When the Date is Invalid

The result of the SELECT statement in this example is a date in February, 1996. The result would be February 31, 1996 if that were a valid date, but because February 31 is not a valid date, ADD_MONTHS normalizes the answer.

That answer, because the DAY value in the source date is greater than the last DAY value for the target month, is the last valid DAY value for the *target* month.

```
SELECT ADD_MONTHS ('1995-12-31', 2);
```

The result of this example is 1996-02-29.

Note that 1996 was a leap year. If the interval were 14 months rather than 2, the result would be '1997-02-28'.

Example: Querying for the Month Prior to the Current Date

This statement performs the converse of the ADD_MONTHS function in [Example: Querying for the Current Date Plus One Month](#).

You might expect it to return '1999-01-30', which is the source date in that example, but it does not.

```
SELECT ADD_MONTHS ('1999-02-28' , -1);
```

ADD_MONTHS returns the result 1999-01-28.

The function performs as designed and this result is not an error, though it might not be what you would expect from reading [Example: Querying for the Current Date Plus One Month](#).

Example: Querying for the Current Date Plus One Month for Month Ending on the 28th

You might expect the following statement to return '1999-03-31', but it does not.

```
SELECT ADD_MONTHS ( '1999-02-28' , 1);
```

ADD_MONTHS returns the result 1999-03-28.

Example: Querying for the Current Date Plus One Month for Month Ending on the 30th

You might expect the following statement to return '1999-03-31', but it does not.

```
SELECT ADD_MONTHS ( '1999-04-30' , -1);
```

ADD_MONTHS returns the result 1999-03-30.

Example: Querying for the Current Date Plus One Month for Month Ending on the 30th

You might expect the following statement to return '1999-05-31', but it does not.

```
SELECT ADD_MONTHS ( '1999-04-30' , 1);
```

ADD_MONTHS returns the result 1999-05-30.

Related Information

- Information about CALENDAR view in *Teradata Vantage™ - Data Dictionary*, B035-1092.
- *Teradata Vantage™ - Database Utilities*, B035-1102.

OADD_MONTHS

Adds a specified *date_timestamp_value* to a specified *num_months* and returns the resulting date.

ANSI Compliance

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

Result Type

OADD_MONTHS is a scalar function whose return data type is DATE.

OADD_MONTHS Syntax

```
[TD_SYSFNLIB.] OADD_MONTHS ( date_timestamp_value, num_months )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

date_timestamp_value

A date or timestamp argument.

If this syntax element is NULL, NULL is returned.

num_months

A numeric argument.

If this syntax element is NULL, NULL is returned.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *date_timestamp_value* = DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE
- *num_months* = INTEGER

For the *num_months* argument, you can also pass values with data types that can be converted to INTEGER using the implicit data type conversion rules that apply to UDFs. Implicit type conversion is not supported for the *date_timestamp_value* argument.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Since TIMESTAMP values are stored in UTC time within the database and lack a time zone, the session time zone is used to interpret the time stamp value within the function. For TIMESTAMP WITH TIME ZONE values, the time zone component is used to interpret the time stamp value within the function.

If the day component of *date_timestamp_value* is the last day of the month, or if the resulting month has fewer days than the day component of *date_timestamp_value*, OADD_MONTHS returns the last day

of the resulting month. Otherwise, OADD_MONTHS returns a value that has the same day component as *date_timestamp_value*.

The difference between OADD_MONTHS and ADD_MONTHS is that if a month is added to an end-of-month date in OADD_MONTHS, the function always returns an end-of-month date. The following queries illustrate the difference between ADD_MONTHS and OADD_MONTHS:

- This query:

```
SELECT ADD_MONTHS ('2008-02-29', 1);
```

returns '08/03/29'

- This query:

```
SELECT OADD_MONTHS ('2008-02-29', 1);
```

returns '08/03/31'

Examples

Example: Adding a Month to a Specified Date

The following query:

```
SELECT OADD_MONTHS (DATE '2008-02-15', 1);
```

returns the result 08/03/15.

Example: Adding Two Month to a Specified Date

The following query:

```
SELECT OADD_MONTHS (DATE '2008-02-20', 2);
```

returns the result 08/04/20.

Example: Adding a Month to a Specified Date When the Month Ends on the 29th

The following query:

```
SELECT OADD_MONTHS (DATE '2008-02-29', 1);
```

returns the result 08/03/31.

Since 29 is the last day in February, March 31 is returned since 31 is the last day in March.

TRUNC(Date)

Truncates *date_value* based on the format specified by *fmt*.

ANSI Compliance

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

Result Type

TRUNC is a scalar function whose return data type is DATE.

If the data type of *date_value* is TIMESTAMP or TIMESTAMP WITH TIME ZONE, the return data type is DATE by default. If you want this function to return a TIMESTAMP value when you pass in TIMESTAMP as input, you must set the TruncRoundReturnTimestamp DBS Control field to TRUE.

TRUNC(Date) Syntax

```
[TD_SYSFNLIB.] TRUNC ( date_value [, fmt ] )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

date_value

A date or timestamp argument.

fmt

A character expression.

If this syntax element is NULL, NULL is returned.

If *fmt* is omitted, *date_value* is truncated to the nearest day. Sunday is considered the first day of the week. For ISO years, Monday is considered the first day of the week.

If this syntax element is not valid, an error is returned.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- *date_value* = DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE
- *fmt* = VARCHAR

For the *fmt* argument, you can also pass values with data types that can be converted to VARCHAR using the implicit data type conversion rules that apply to UDFs. Implicit type conversion is not supported for the *date_value* argument.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

Date Formats

Element	Description
CC SCC	One year greater than the first two digits of a 4-digit year.
YYYY YYYY YEAR SYEAR YYY YY Y	Year. Returns a value of 1, the first day of the year.
IYYY IYY IY I	ISO year
MONTH MON MM RM	Month. Returns a value of 1, the first day of the month.
Q	Quarter. Returns a value of 1, the first day of the quarter.
WW	Same day of the week as the 1st day of the year.
IW	Same day of the week as the first day of the ISO year.
W	Same day of the week as the first day of the month.
DDD DD J	Day.
DAY DY	Starting day of the week.

Element	Description
D	
HH HH12 HH24	Hour.
MI	Minute.

Examples

Example: Returning a Date Value for the First Day of the Week

The following query:

```
SELECT TRUNC(CAST('2003/09/17' AS DATE), 'D') (FORMAT 'yyyy-mm-dd');
```

returns the result 2003-09-14. The date was truncated to the first day of that week.

Example: Returning a Date Value for the Beginning of the Month

The following queries:

```
SELECT TRUNC(CAST('2003/09/15' AS DATE), 'MONTH') (FORMAT 'yyyy-mm-dd');
```

```
SELECT TRUNC(CAST('2003/09/30' AS DATE), 'MON') (FORMAT 'yyyy-mm-dd');
```

```
SELECT TRUNC(CAST('2003/09/17' AS DATE), 'MM') (FORMAT 'yyyy-mm-dd');
```

```
SELECT TRUNC(CAST('2003/09/17' AS DATE), 'RM') (FORMAT 'yyyy-mm-dd');
```

all return the result 2003-09-01. The date was truncated to the beginning of the month.

Example: Returning a TIMESTAMP Value for TIMESTAMP Input

In this example, the TruncRoundReturnTimestamp DBS Control field is set to TRUE; therefore, TRUNC returns a TIMESTAMP value when the input data type is TIMESTAMP.

```
SELECT TRUNC(cast('2000-08-04 22:00:00-05:00' as timestamp with time zone), 'DD');
```

Result:

```
TRUNC( '2000-08-04 22:00:00-05:00', 'DD' )
-----
2000-08-04 00:00:00.000000-05:00
```

Related Information

For more information, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

ROUND(Date)

Returns *date_value* with the time portion of the day rounded to the unit specified by *fmt*.

ANSI Compliance

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

Result Type

ROUND is a scalar function with a return data type of DATE.

If the data type of *date_value* is TIMESTAMP or TIMESTAMP WITH TIME ZONE, the return data type is DATE by default. If you want this function to return a TIMESTAMP value when you pass in TIMESTAMP as input, you must set the TruncRoundReturnTimestamp DBS Control field to TRUE.

ROUND(Date) Syntax

```
[TD_SYSFNLIB.] ROUND ( date_value [, fmt ] )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

date_value

A date or timestamp argument.

fmt

A character expression.

If this syntax element is NULL, NULL is returned.

If *fmt* is omitted, *date_value* is truncated to the nearest day. Sunday is considered the first day of the week. For ISO years, Monday is considered the first day of the week.

If this syntax element is not valid, an error is returned.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- *date_value* = DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE
- *fmt* = VARCHAR

For the *fmt* argument, you can also pass values with data types that can be converted to VARCHAR using the implicit data type conversion rules that apply to UDFs. Implicit type conversion is not supported for the *date_value* argument.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

Date Formats

Element	Description
CC SCC	One year greater than the first two digits of a 4-digit year.
YYYY YYYY YEAR SYEAR YYY YY Y	Year. Returns a value of 1, the first day of the year.
IYYY IYY IY I	ISO year
MONTH MON MM RM	Month. Returns a value of 1, the first day of the month.
Q	Quarter. Returns a value of 1, the first day of the quarter.
WW	Same day of the week as the 1st day of the year.

Element	Description
IW	Same day of the week as the first day of the ISO year.
W	Same day of the week as the first day of the month.
DDD DD J	Day.
DAY DY D	Starting day of the week.
HH HH12 HH24	Hour.
MI	Minute.

Examples

Example: Rounding the Date to the First Day of the Week

The following query:

```
SELECT ROUND(CAST('2003/09/17' AS DATE), 'D') (FORMAT 'yyyy-mm-dd');
```

returns the result 2003-09-14. The date was rounded to the first day of that week.

Example: Rounding the Date to the Beginning of the Next Month

The following query:

```
SELECT ROUND(CAST('2003/09/15' AS DATE), 'MONTH') (FORMAT 'yyyy-mm-dd');
```

returns the result 2003-09-01. Since the day is less than 16, the date returns to the beginning of the month.

The following query:

```
SELECT ROUND(CAST('2003/09/17' AS DATE), 'RM') (FORMAT 'yyyy-mm-dd');
```

returns the result 2003-10-01. Since the day is greater than or equal to 16, the date is rounded to the beginning of the next month.

Example: Returning a TIMESTAMP Value for TIMESTAMP Input

In this example, the TruncRoundReturnTimestamp DBS Control field is set to TRUE; therefore, ROUND returns a TIMESTAMP value when the input data type is TIMESTAMP.

```
SELECT ROUND( cast('2003-08-17 12:15:23+05:00' as timestamp with
time zone), 'HH24');
```

Result:

```
ROUND('2003-08-17 12:15:23+05:00', 'HH24')
-----
2003-08-17 12:00:00.000000+05:00
```

Related Information

- For details, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For date formats for ROUND(Date), see [TRUNC\(Date\)](#).

EXTRACT

Extracts a single specified full ANSI SQL:2011 field from any DateTime or Interval value, converting it to an exact numeric value.

ANSI Compliance

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

EXTRACT Syntax

```
EXTRACT (
  { YEAR | MONTH | DAY | HOUR | MINUTE |
    SECOND | TIMEZONE_HOUR | TIMEZONE_MINUTE
  } FROM value
)
```

Syntax Elements

YEAR

The integer value for YEAR to be extracted from the date represented by value.

MONTH

The integer value for MONTH to be extracted from the date represented by value.

DAY

The integer value for DAY to be extracted from the date represented by value.

HOURL

The integer value for HOUR to be extracted from the date represented by value.

MINUTE

The integer value for MINUTE to be extracted from the date represented by value.

TIMEZONE_HOUR

The integer value for TIMEZONE_HOUR to be extracted from the date represented by value.

TIMEZONE_MINUTE

The integer value for TIMEZONE_MINUTE to be extracted from the date represented by value.

SECOND

The integer and decimal values for SECOND are to be extracted from the date represented by value. The returned value has a data type of DECIMAL(8,2).

value

An expression that results in a DateTime, Interval, or UDT value:

value	Description
Character string expression that represents a date	String must match the 'YYYY-MM-DD' format.
Character string expression that represents a time	String must match the 'HH:MI:SS.SSSSSS' format.

value	Description
Floating point type	<p>Value must be a time value encoded with the algorithm $\text{HOUR} * 10000 + \text{MINUTE} * 100 + \text{SECOND}$.</p> <p>Only HOUR, MINUTE, and SECOND can be extracted from a floating point value.</p> <p>Externally created time values can be appropriately encoded and stored in a REAL column to any desired precision if the encoding creates a value representable by REAL without precision loss.</p> <p>Do not store time values as REAL in any new applications. Instead, use the more rigorously defined ANSI SQL:2011 DateTime data types.</p>
UDT	<p>UDT must have an implicit cast that casts between the UDT and any of the following predefined types:</p> <ul style="list-style-type: none"> • Numeric • Character • DateTime <p>To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information, see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</p> <p>Implicit type conversion of UDTs for system operators and functions, including EXTRACT, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. See <i>Teradata Vantage™ - Database Utilities</i>, B035-1102.</p> <p>For more information on implicit type conversion of UDTs, see <i>Teradata Vantage™ - Data Types and Literals</i>, B035-1143.</p>
Not a character string expression or floating point type or UDT	Expression must evaluate to a DateTime or Interval type.

Result Types

EXTRACT returns an exact numeric value for ANSI SQL:2011 DateTime values.

EXTRACT returns values adjusted for the appropriate time zone if the data type of the argument is TIME or TIMESTAMP.

If no time zone is specified for the argument, then the time zone displacement based on the current session time zone is used; otherwise, the explicit time zone of the argument is used. You can use the AT clause to explicitly specify a time zone for the argument. For more information, see [ANSI DateTime Expressions](#).

If you extract SECOND, then if the value has a seconds fraction precision of zero, the result is integer; if the value has a seconds fractional prevision of greater than zero, the result is DECIMAL with the scaling as specified for the SECOND field in its data description.

If you extract anything else, the result is INTEGER with 32 bits of precision.

If you extract ...	THEN ...
SECOND	If <i>value</i> has a seconds fractional of precision of: <ul style="list-style-type: none"> • zero, the result is INTEGER. • greater than zero, the result is DECIMAL with the scaling as specified for the SECOND field in its data description.
anything else	the result is INTEGER, with 32 bits of precision.

If *value* is NULL, the result is NULL.

Examples

Example: Returning the Year From the Current Date

The following example returns the year, as an integer, from the current date.

```
SELECT EXTRACT (YEAR FROM CURRENT_DATE);
```

Example: Adding 90 days to PurchaseDate

Assuming PurchaseDate is a DATE field, this example returns the month of the *date value* formed by adding 90 days to PurchaseDate as an integer.

```
SELECT EXTRACT (MONTH FROM PurchaseDate+90) FROM SalesTable;
```

Example: Returning an Integer

The following returns 12 as an integer.

```
SELECT EXTRACT (DAY FROM '1996-12-12');
```

Example: Character Literal Does Not Evaluate to a Valid Date

This example returns an error because the character literal does not evaluate to a valid date.

```
SELECT EXTRACT (DAY FROM '1996-02-30');
```

Example: Character String Literal Does Not Match the ANSI SQL:2011 Date Format

The following returns an error because the character string literal does not match the ANSI SQL:2011 date format.

```
SELECT EXTRACT (DAY FROM '96-02-15');
```

If the argument to `EXTRACT` is a value of type `DATE`, the value contained is warranted to be a valid date, for which `EXTRACT` cannot return an error.

Example: Non-ANSI DateTime Definitions

The following example relates to non-ANSI DateTime definitions. If the argument is a character literal formatted as a time value, it is converted to `REAL` and processed. In this example, 59 is returned.

```
SELECT EXTRACT (MINUTE FROM '23:59:17.3');
```

Example: Returning the Hour From the Current Time

This example returns the hour, as an integer, from the current time.

```
SELECT EXTRACT (HOUR FROM CURRENT_TIME);
```

Current time is retrieved as the system value `TIME`, to the indicated precision.

Example: Returning the Seconds as DECIMAL

The following example returns the seconds as `DECIMAL(8,2)`. This is based on the fractional seconds precision of 2 for `CURRENT_TIME`.

```
SELECT EXTRACT (SECOND FROM CURRENT_TIME (2));
```

GetTimeZoneDisplacement

Returns the rules and time zone displacement information for a specified time zone string.

`GetTimeZoneDisplacement` is a system user-defined function (UDF) that Vantage invokes internally to resolve a time zone string specified in an SQL statement or Specification for Data Formatting (SDF) file. You do not invoke this function directly; however, you can modify this UDF to add new time zone strings or add or modify the rules of an existing time zone string.

ANSI Compliance

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

GetTimeZoneDisplacement Syntax

```
GetTimeZoneDisplacement ( time_zone_string )
```

Syntax Elements

time_zone_string

A valid time zone string specified using a VARBYTE data type. For a list of time zone strings supported by Teradata, see [AT LOCAL and AT TIME ZONE Time Zone Specifiers](#).

If *time_zone_string* is not valid or unsupported, GetTimeZoneDisplacement returns a value of 1 in the first byte to indicate that the time zone string does not exist.

Result Type

GetTimeZoneDisplacement returns a string of bytes containing the rules and time zone displacement information for the specified time zone string. The result data type is BYTE. The information returned is:

Byte	Value
First byte	<ul style="list-style-type: none"> • 1, if the time zone string is not found. That is, the time zone string specified in the input argument is not valid or unsupported. • 0, if the time zone string is found.
Second byte	<ul style="list-style-type: none"> • 1, if the time zone string has separate daylight saving time and standard time zone displacements from Coordinated Universal Time (UTC) time. In this case, the next 480 or so bytes store the set of rules describing a valid standard time zone displacement, daylight saving time zone displacement, and the start and end time for daylight saving time. A maximum of 6 rules are stored for each time zone string. • 0, if the time zone string does not have separate daylight saving time and standard time zone displacements from UTC time. In this case, the next 4 bytes store the time zone displacement hour and minute values.

Usage Notes

Limitation on the Use of TimeZone Strings

When using standard Teradata system time zone strings, no time zone rules are enforced for the years 1986 and before, for example, daylight saving time (DST) shifts. Valid years for Teradata standard time zone strings are 1987 through 9999.

If Teradata standard time zone strings do not meet your requirements, you can add a new custom time zone string or modify an existing string by modifying or adding new rules to GetTimeZoneDisplacement.

Adding or Modifying Time Zone Strings

Teradata provides a set of time zone strings that represent commonly used time zones. For a list of supported time zone strings, see [AT LOCAL and AT TIME ZONE Time Zone Specifiers](#). The GetTimeZoneDisplacement UDF stores and maintains these time zone strings and the related rules for converting between UTC and the time in the local time zone.

If the supplied time zone strings do not meet your requirements, you may add or modify the time zone strings by modifying the GetTimeZoneDisplacement UDF, which is located in the SYSLIB database. The source code for the UDF is available as part of the DBS package and is located at /tdbms/etc/dem/src.

To define new time zone strings or add or modify the rules of an existing time zone string:

1. Make a backup copy of the existing GetTimeZoneDisplacement UDF.

```
# cd /home
# mkdir workdir
# cd workdir
# cp /tdbms/etc/dem/src/udfgettimezonedisplacement.c .
```

2. To modify an existing time zone string:

Note:

Please engage Teradata Services to consult about any rule change in existing time zone strings provided in tdlocaledef_tzrules.txt.

```
# cd /opt/teradata/tdat/tdbms/xx.xx.xx.xx/etc
# vi tdlocaledef_tzrules.txt

TimeZoneString {"America Eastern"; "-5"; "0"; "2";
    "4"; "4"; "1"; "0"; "0"; "02:00:00";
    "3"; "10"; "0"; "0"; "-1"; "02:00:00";
    "1987"; "2006"; "-5"; "0"; "-4"; "0";
    "4"; "3"; "8"; "0"; "0"; "02:00:00";
    "4"; "11"; "1"; "0"; "0"; "02:00:00";
    "2007"; "9999"; "-5"; "0"; "-4"; "0"}
```

- a. Find the time zone string entry in the TZ_DST structure of the GetTimeZoneDisplacement UDF. For example:

```
{"America Eastern", 2,
  {{{4, 4, 1, 0, 0, "02:00:00"},
    {3, 10, 0, 0, -1, "02:00:00"},
    {1987, 2006, -5, 0, -4, 0}},
  {{4, 3, 8, 0, 0, "02:00:00"},
```



```

    {4, 11, 1, 0, 0, "02:00:00"},
    {2007, 9999, -5, 0, -4, 0}},
    {{0, 0, 0, 0, 0, "00:00:00"},
    {0, 0, 0, 0, 0, "00:00:00"},
    {0, 0, 0, 0, 0, 0}},
    {{0, 0, 0, 0, 0, "00:00:00"},
    {0, 0, 0, 0, 0, "00:00:00"},
    {0, 0, 0, 0, 0, 0}},
    {{0, 0, 0, 0, 0, "00:00:00"},
    {0, 0, 0, 0, 0, "00:00:00"},
    {0, 0, 0, 0, 0, 0}},
    {{0, 0, 0, 0, 0, "00:00:00"},
    {0, 0, 0, 0, 0, "00:00:00"},
    {0, 0, 0, 0, 0, 0}}
  },
  -5, 0
},

```

Note:

Rules based on a time zone string in `tdlocaledef_tzrules.txt` and `udfgettimezonedisplacement.c` are listed in different formats. To run `tdlocaledef` to set the time zone string for a system, the format for `tdlocaledef_tzrules.txt` is used.

- b. Modify the rules and information associated with the time zone string entry or add new rules to the entry.
3. To add a new time zone string:
 - a. Create a new entry in the `TZ_DST` structure for the new time zone string and its related rules.

Note:

Rules based on a time zone string in `tdlocaledef_tzrules.txt` and `udfgettimezonedisplacement.c` are listed in different formats. To run `tdlocaledef` to set the time zone string for a system, the format for `tdlocaledef_tzrules.txt` is used.

- b. Place the new time zone string entry in the correct alphabetical position within the `TZ_DST` structure.
- c. `MAX_DST_ENTRIES` in `udfgettimezonedisplacement.c` must match the existing time zone strings. Increase one when adding one new time zone string.
4. Recompile the UDF using the `REPLACE FUNCTION` statement. For more details, see the information about the external form of `REPLACE FUNCTION` in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Note:

Make sure to log in as DBC and that no other user is logged on.

For example:

```
Database SYSLIB;
DROP FUNCTION GetTimeZoneDisplacement;
REPLACE FUNCTION GetTimeZoneDisplacement
    (tzstringinfo VARBYTE(130))
RETURNS BYTE(340)
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'CS!GetTimeZoneDisplacement!/home/
workdir/udfgettimezonedisplacement.c' ;
```

5. Test the new time zone string:

```
.os date
.os date -u

SELECT CURRENT_TIMESTAMP AS SYSTEMTIME,
       CURRENT_TIMESTAMP AT 'new_time_zone_string';
```

TZ_DST Structure

The TZ_DST structure is an array of TZwithDST elements where each element describes a time zone string and its related rules. The definition of the TZwithDST structure is:

```
typedef struct TZwithDST
{
    CHARACTER_LATIN  tzstring[TZSTRINGSIZE];
    INTEGER          number_of_rules;
    DSTRules         TZRules[TZRulesEntries];
    SMALLINT         Standardtzdispl_hour;
    SMALLINT         Standardtzdispl_minute;
} TZwithDST;
```

tzstring

The name of the time zone string. For example, "America Pacific."

The maximum length of a time zone string is 130 bytes.

number_of_rules

The number of rules related to this time zone string.

A maximum of 6 rules is allowed for each time zone string.

TZRules

An array where each DSTRules element describes a rule. These rules are used to calculate the time zone displacement for the time zone string.

The definition of the DSTRules structure is:

```
typedef struct DSTRules
{
    startendDSTInfo  startDST;
    startendDSTInfo  endDST;
    yearDisplInfo    validyrs;
} DSTRules;
```

Standardtzdispl_hour

The standard time zone displacement hour.

Standardtzdispl_minute

The standard time zone displacement minute.

startDST

Specifies the date and time when daylight saving time (DST) starts. You can specify the fields in the following table. If a field is not applicable, enter 0.

endDST

Specifies the date and time when daylight saving time ends. You can specify the fields in the following table. If a field is not applicable, enter 0.

validyrs

Specifies the years in which the DST start and end dates apply. The following information related to this year range is included:

- start_year - the year when these DST rules start.
- end_year - the year when these DST rules end.
- Standardtzdispl_hour - the standard time zone displacement hour.
- Standardtzdispl_minute - the standard time zone displacement minute.
- DSTtzdispl_hour - the time zone displacement hour for daylight saving time.

- DSTtzdispl_minute -the time zone displacement minute for daylight saving time.

You can specify the following for startDST and endDST. Enter zero if a field is not applicable.

startDST and endDST Fields

Field	Description												
rule_type	<p>Indicates how the start and end date for DST is specified. The valid values are:</p> <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0</td><td>Unspecified. Defaults to standard time zone displacement.</td></tr> <tr> <td>1</td><td>DST starts or ends on specified fixed date, which is specified by month and day_of_month fields.</td></tr> <tr> <td>2</td><td>DST starts or ends on the 1st, 2nd, or 3rd weekday of month as indicated by month, day_of_week, and week_of_month fields.</td></tr> <tr> <td>3</td><td>DST starts or ends on the 2nd to the last, 3rd to the last, or the last weekday of the month as indicated by month, day_of_week, and week_of_month fields.</td></tr> <tr> <td>4</td><td>DST starts or ends on next weekday or immediately after date specified in day_of_month field. Month and weekday are specified in month and day_of_week fields.</td></tr> </table> <p>For example, for time zone string 'America Pacific', start date rule is first Sunday after March 8 (March 14 in year 2010).</p>	Value	Meaning	0	Unspecified. Defaults to standard time zone displacement.	1	DST starts or ends on specified fixed date, which is specified by month and day_of_month fields.	2	DST starts or ends on the 1st, 2nd, or 3rd weekday of month as indicated by month, day_of_week, and week_of_month fields.	3	DST starts or ends on the 2nd to the last, 3rd to the last, or the last weekday of the month as indicated by month, day_of_week, and week_of_month fields.	4	DST starts or ends on next weekday or immediately after date specified in day_of_month field. Month and weekday are specified in month and day_of_week fields.
Value	Meaning												
0	Unspecified. Defaults to standard time zone displacement.												
1	DST starts or ends on specified fixed date, which is specified by month and day_of_month fields.												
2	DST starts or ends on the 1st, 2nd, or 3rd weekday of month as indicated by month, day_of_week, and week_of_month fields.												
3	DST starts or ends on the 2nd to the last, 3rd to the last, or the last weekday of the month as indicated by month, day_of_week, and week_of_month fields.												
4	DST starts or ends on next weekday or immediately after date specified in day_of_month field. Month and weekday are specified in month and day_of_week fields.												
month	<p>The month when DST starts or ends. Valid values are 0- 12. This field is used for rule_type 1, 2, 3, and 4.</p> <p>For example, for time zone string 'America Pacific', the start date rule is the first Sunday after March 8; therefore, this field has a value of 3 in the startDST structure to represent March.</p>												
day_of_month	<p>If rule_type is 1, this field specifies the day of the month when DST starts or ends. For example, if DST ends at 12:00 am local time on August 21, this field contains the value 21 in the endDST structure.</p> <p>If rule_type is 4, DST starts or ends on the next weekday on or immediately after the date specified by this field. For example, for time zone string 'America Pacific', the start date rule is the first Sunday after March 8; therefore, this field has a value of 8 in the startDST structure.</p> <p>When rule_type is 0, 2 or 3, this field is not used and the value is 0.</p>												
day_of_week	<p>The valid values are 0-7 representing the weekdays Sunday-Saturday. This field is used for rule_type 2, 3 and 4.</p> <p>For example, for time zone string 'America Pacific', the start date rule is the first Sunday after March 8; therefore, this field has a value of 0 in the startDST structure to represent Sunday.</p>												
week_of_month	<p>The valid values are 1, 2, 3, 4, 5, -1, and -2 representing the 1st, 2nd, 3rd, 4th, 5th, last, and second to the last weekday of the month. This field is used for rule_type 2 and 3.</p> <p>For example, for time zone string 'Europe Azores', the start date rule is the last Sunday in March; therefore, this field has a value of -1 in the startDST structure to represent the last week of the month.</p>												
loctime	<p>The local time when DST starts or ends.</p> <p>For example, "02:00:00" indicates that DST starts or ends at 2:00 am local time.</p>												

Example: Adding a New Time Zone String

Assume that you want to add a new time zone string 'Europe Azores', which has one rule with the following time zone displacement information:

- DST starts on the last Sunday in March at 12:00 am local time.
- DST ends on the last Sunday in October at 1:00 am local time.
- The standard time zone offset from UTC is -1.
- The daylight saving time offset from UTC is 0.
- The start year for the rule is 2009.
- The end year for the rule is 2010.

Based on this information, the new time zone string entry for 'Europe Azores' is:

<code>{"Europe Azores", 1,</code>	<code><= 1 rule defined for 'Europe Azores'</code>
<code> {{{3, 3, 0, 0, -1, "00:00:00"},</code>	<code><= Start of rule 1,</code>
<code>startDST information</code>	
<code> {3, 10, 0, 0, -1, "01:00:00"},</code>	<code><= endDST information</code>
<code> {2009, 2010, -1, 0, 0, 0}},</code>	<code><= validyrs information</code>
<code> {{0, 0, 0, 0, 0, "00:00:00"},</code>	<code><= Start of rule 2</code>
<code> {0, 0, 0, 0, 0, "00:00:00"},</code>	
<code> {0, 0, 0, 0, 0, 0}},</code>	
<code> {{0, 0, 0, 0, 0, "00:00:00"},</code>	<code><= Start of rule 3</code>
<code> {0, 0, 0, 0, 0, "00:00:00"},</code>	
<code> {0, 0, 0, 0, 0, 0}},</code>	
<code> {{0, 0, 0, 0, 0, "00:00:00"},</code>	<code><= Start of rule 4</code>
<code> {0, 0, 0, 0, 0, "00:00:00"},</code>	
<code> {0, 0, 0, 0, 0, 0}},</code>	
<code> {{0, 0, 0, 0, 0, "00:00:00"},</code>	<code><= Start of rule 5</code>
<code> {0, 0, 0, 0, 0, "00:00:00"},</code>	
<code> {0, 0, 0, 0, 0, 0}},</code>	
<code> {{0, 0, 0, 0, 0, "00:00:00"},</code>	<code><= Start of rule 6</code>
<code> {0, 0, 0, 0, 0, "00:00:00"},</code>	
<code> {0, 0, 0, 0, 0, 0}}</code>	
<code>},</code>	
<code>-1, 0</code>	<code><= Standard time zone displacement</code>
<code>},</code>	

Note that the time zone string entry has space for 6 rules but only one rule is used for the start year 2009 and end year 2010.

You must place the new 'Europe Azores' time zone string in between the 'Australia Western' and 'Europe Central' time zone strings in the TZ_DST structure to maintain the alphabetical order of the structure.

Related Information

- For more information on setting session time zones, see SET TIME ZONE, CREATE USER, MODIFY USER in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For more information on system time zone settings, see *Teradata Vantage™ - Database Utilities*, B035-1102.
- For more information on automatic adjustment of the system time to account for daylight saving time, see the information about the SDF file and Teradata Locale Definition Utility (tdlocaledef) in *Teradata Vantage™ - Database Utilities*, B035-1102.

Period Functions and Operators

The following sections describe the Period functions and operators.

Several of these functions and operators support both true Period data types and derived period columns. The Period data types and derived period columns comprise two individual DateTime columns that store the beginning and ending bound values of the duration represented by the derived period.

Period Value Constructor

Initializes an instance of a Period data type.

Period Value Constructor Syntax

```
PERIOD (
    datetime_expression
    [, { datetime_expression | UNTIL_CHANGED | UNTIL_CLOSED } ]
)
```

Syntax Elements

datetime_expression

An expression that evaluates to a DATE, TIME, or TIMESTAMP value.

UNTIL_CHANGED

A DATE or TIMESTAMP value that is considered to be forever or until it is changed.

For PERIOD(DATE) types, UNTIL_CHANGED has a value of DATE '9999-12-31'.

For PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) types, UNTIL_CHANGED has a value of TIMESTAMP '9999-12-31 23:59:59.999999 00:00'(with the precision truncated to the precision of the beginning bound and the time zone omitted if the beginning bound does not have a time zone).

UNTIL_CHANGED supports derived periods. See [IS UNTIL_CHANGED/IS NOT UNTIL_CHANGED](#).

UNTIL_CLOSED

An ending bound for the Period value of a temporal table transaction-time column that indicates that the row is an open row.

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

For more information about temporal tables, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

UNTIL_CLOSED supports derived periods. See [IS UNTIL_CLOSED/IS NOT UNTIL_CLOSED](#).

Usage Notes

Result Value Rules

The following rules apply to the result value:

- If the beginning or ending bound is NULL, or both the bounds are NULL, the result is NULL.
- If the beginning and ending bounds are NULL or if the beginning bound is NULL and the ending bound is UNTIL_CHANGED, then the type of the period defaults to `PERIOD(TIMESTAMP(0))`.
- If only the beginning bound is specified, the result ending bound is the beginning bound plus one granule of the result element type. If the result ending bound exceeds or becomes equal to the maximum allowed DATE or TIMESTAMP value for result data type of `PERIOD(DATE)` or `PERIOD(TIMESTAMP(n) [WITH TIME ZONE])`, respectively, an error is reported.
- If an ending bound is specified as a value expression and the beginning bound and ending bound have different precisions, the result precision is the higher of the two precisions. Otherwise, the result precision is the precision of the beginning bound.
- UNTIL_CHANGED sets the result ending element to a maximum DATE or TIMESTAMP value depending on the data type of the beginning bound. If the data type of the beginning bound is `TIMESTAMP(n) WITH TIME ZONE`, the result ending element is set to the maximum `TIMESTAMP(n) WITH TIME ZONE` value at UTC (that is, the time zone displacement for the ending bound is `INTERVAL '00:00' HOUR TO MINUTE`).
- If the beginning bound or the ending bound or the beginning and ending bounds include a time zone value, and the ending bound is not UNTIL_CHANGED, the result data type is `WITH TIME ZONE`. If only one of the bounds includes a time zone value, the time zone field of the other is set to the current session time zone displacement. If both bounds include time zone values, the result bounds include the corresponding time zone value.
- The result Period data type has an element type that is the same as the DateTime data type of the beginning bound except with the precision and time zone as defined previously.
- The handling of leap seconds for Period data types with TIME and TIMESTAMP element types is as follows. If the value for the beginning or ending bound contains leap seconds, the seconds portion gets adjusted to 59.999999 with the precision truncated to the result precision. During this process, if the beginning and ending bounds are the same, an error is reported.

Period Value Constructor Rules

The following rules apply to the Period value constructor:

- The beginning bound must have a DateTime data type and, if an ending bound is specified, the data types of the beginning and ending bounds must be comparable.
- The ending bound where the data type of the beginning bound is DATE or TIMESTAMP can be set to UNTIL_CHANGED.
- If the ending bound is set to UNTIL_CLOSED, the following must be true:
 - The data type of the beginning bound value must be comparable with TIMESTAMP(6) WITH TIME ZONE.
 - The constructor is only valid in an assignment operation where the target column to which the result is assigned is a transaction-time column.
 - Because the only way to set the value of a transaction-time column is by using nontemporal DML, the constructor is only valid in a nontemporal DML statement.
- The system reports an error if any of the following are true:
 - UNTIL_CHANGED is specified for the beginning bound.
 - The result beginning bound is greater than or equal to the result ending bound.
 - The data types of the beginning and ending bounds are not comparable.
 - UNTIL_CHANGED is specified for the ending bound and the data type of the beginning bound is TIME(n) [WITH TIME ZONE].
 - UNTIL_CLOSED is specified for the beginning bound.

Example

In the following example, assume t1 is a table with an INTEGER column c1 and a PERIOD(DATE) column c2 and t2 is a table with an INTEGER column a and two DATE columns b and c.

This example shows the Period value constructor used in two INSERT statements.

```
INSERT INTO t1
VALUES (1, PERIOD(DATE '2005-02-03', DATE '2006-02-04'));
INSERT INTO t1 SELECT a, PERIOD(b, c) FROM t2;
```

Arithmetic Operators

Adds or subtracts an Interval value to or from a Period value, or adds a Period value to an Interval value.

Arithmetic Operator Syntax

period_expression {+|-} *interval_expression*

Note:

The order of *period_expression* and *interval_expression* is not important.

Syntax Elements

period_expression

The Period expression to be converted.

Any expression that evaluates to a Period data type.

interval_expression

An expression that evaluates to an INTERVAL data type.

Assuming that *p* is a Period expression of element type DATE or TIMESTAMP and *v* is an Interval expression:

- *p* + *v* and *v* + *p* are both equivalent to:

```
PERIOD(BEGIN(p) + v, CASE WHEN END(p) IS UNTIL_CHANGED THEN
END(p) ELSE (END(p) + v) END)
```

- *p* - *v* is equivalent to:

```
PERIOD(BEGIN (p) - v, CASE WHEN END(p) IS UNTIL_CHANGED THEN
END(p) ELSE (END(p) - v) END)
```

Assuming that *p* is a Period expression of element type TIME and *v* is an interval expression:

- *p* + *v* and *v* + *p* are both equivalent to:

```
PERIOD(BEGIN(p) + v, END(p) + v)
```

- *p* - *v* is equivalent to:

```
PERIOD(BEGIN (p) - v, END(p) - v)
```

Usage Notes

The following rules apply to arithmetic operators and Period data types:

- The interval expression must be a valid interval expression and must follow the rules of an Interval expression (see “ANSI Interval Expressions”). Otherwise, an error is reported. For example, the interval expression (DATE '2006-02-03' - DATE '2005-02-03') DAY, results in a value of 365 days which cannot fit into the default precision 2 of the interval qualifier DAY; therefore, an error is reported.
- The period arithmetic operations of adding or subtracting an Interval to or from a period or adding a period to an Interval follow the rules of DateTime expressions. Otherwise, errors are reported. See [ANSI Interval Expressions](#) for details on DateTime expression rules.
- An interval expression can be subtracted from a Period expression but not vice versa. If a Period expression is subtracted from an interval expression, an error is reported.
- For a Period expression with an element type of TIME, if the Period arithmetic operation results in a beginning bound less than the ending bound, an error is reported.
- For a period of element type DATE or TIMESTAMP, if the ending bound is UNTIL_CHANGED, the ending bound in the result ending bound is UNTIL_CHANGED. If the ending bound is not UNTIL_CHANGED and the ending bound in the result evaluates to an UNTIL_CHANGED value, an error is reported.
- For a period arithmetic operation, one of the operands must be an INTERVAL data type. Otherwise, an error is reported.

Comparison of Period Types

Data Type	Primary fields
DATE	YEAR, MONTH, and DAY
TIME	HOUR, MINUTE, and SECOND
TIMESTAMP	YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND

Two Period values are comparable if their element types are of same DateTime data type. The DateTime data types are DATE, TIME and TIMESTAMP.

- The PERIOD(DATE) date type is comparable with the PERIOD(DATE) data type.
- The PERIOD(TIME(n)[WITH TIME ZONE]) data type is comparable with a PERIOD(TIME(m)[WITH TIME ZONE]) data type,
- The PERIOD(TIMESTAMP(n)[WITH TIME ZONE]) data type is comparable with a PERIOD(TIMESTAMP(m)[WITH TIME ZONE]) data type.

Teradata extends this to allow a CHARACTER and VARCHAR value to be implicitly cast as a Period data type for some operators and, therefore, have a Period data type. Since the Period data type is the data type of the other Period expression, these Period expressions will be comparable.

DateTime and Period data are saved internally with the maximum precision of 6, although the specified precision may be less than this and is padded with zeros. Thus, the comparison operations with differing precisions work without any additional logic.

In addition, the internal value is saved in UTC for a Time or Timestamp value, or for a Period value with an element type of TIME or TIMESTAMP. All comparable Period expressions can be compared directly due to

this internal representation irrespective of whether they contain a time zone value, or whether they have the same precision.

Note:

The time zone values are ignored when comparing values.

All comparison operations involving UNTIL_CLOSED in a system-versioned system-time or temporal table transaction-time column use the internal value of UNTIL_CLOSED (TIMESTAMP '9999-12-31 23:59:59:999999+00:00') to evaluate the result.

Comparison Operators

The following table describes the comparison operators.

Operator	Purpose
EQ or =	<p>Assume p1 and p2 are Period expressions and have comparable Period data types. If $BEGIN(p1) = BEGIN(p2)$ AND $END(p1) = END(p2)$, the result of the comparison is TRUE; otherwise, the result is FALSE.</p> <p>If either Period expression is NULL, the result is UNKNOWN.</p> <p>If the Period expressions have different element types, one of them must be explicitly CAST as the other.</p> <p>If one Period expression has a Period data type and the other Period expression has CHARACTER or VARCHAR data type, the CHARACTER or VARCHAR expression is implicitly converted, before comparison, to the data type of the Period expression based on the format of the Period expression.</p> <p>EQ supports comparisons between derived periods.</p>
LT or <	<p>Assume p1 and p2 are Period expressions and have comparable Period data types. If $BEGIN(p1) < BEGIN(p2)$ OR $(BEGIN(p1) = BEGIN(p2) \text{ AND } END(p1) < END(p2))$, the result of the comparison is TRUE; otherwise, the result is FALSE.</p> <p>If either Period expression is NULL, the result is UNKNOWN.</p> <p>If the Period expressions have different element types, one of them must be explicitly CAST as the other.</p> <p>If one Period expression has a Period data type and the other Period expression has CHARACTER or VARCHAR data type, the CHARACTER or VARCHAR operand is implicitly converted, before comparison, to the data type of the Period expression based on the format of the Period expression.</p> <p>If the ending bound value of a system-versioned system-time or temporal table transaction-time column is UNTIL_CLOSED, the ending bound value is only less than a TIMESTAMP column value or TIMESTAMP literal if the column value or literal is the maximum TIMESTAMP value with leap seconds. This can be possible only if the ending bound of the system-versioned system-time or transaction-time column is used in a comparison with the timestamp value. For more information about temporal tables, see <i>Teradata Vantage™ - ANSI Temporal Table Support</i>, B035-1186 or <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p> <p>LT supports comparisons between derived periods.</p>
GT or >	<p>Assume p1 and p2 are Period expressions and have comparable Period data types.</p>

Operator	Purpose
	<p>If $BEGIN(p1) > BEGIN(p2)$ OR $(BEGIN(p1) = BEGIN(p2) \text{ AND } END(p1) > END(p2))$, the result of the comparison is TRUE; otherwise, it is FALSE.</p> <p>If either Period expression is NULL, the result is UNKNOWN.</p> <p>If one Period expression has a Period data type and the other Period expression has CHARACTER or VARCHAR data type, the CHARACTER or VARCHAR Period expression is implicitly converted, before comparison, to the data type of the Period expression based on the format of the Period expression.</p> <p>GT supports comparisons between derived periods.</p>
NE or <> or NOT= or ^= or LE or <= or GE or >=	<p>These comparison operators are supported for comparable Period expressions.</p> <p>If one Period expression has a Period data type and the other Period expression has CHARACTER or VARCHAR data type, the CHARACTER or VARCHAR Period expression is implicitly converted, before comparison, to the data type of the Period expression based on the format of the Period expression.</p> <p>NE, LE, and GE support comparisons between derived periods.</p>

BEGIN

Returns the beginning bound of a Period expression or a derived period.

Result Type

The result data type of the BEGIN function is same as the element type of the Period expression or the data type of the begin column if the argument is a derived period column. If the argument is NULL, the result is NULL.

BEGIN Syntax

```
BEGIN ( { period_expression | derived_period } )
```

Syntax Elements

period_expression

The Period expression to be converted.

derived_period

Any expression that evaluates to a Period data type.

Usage Notes

Format and Title

The format is the default format for the element type of the Period expression or the format of the begin column if the argument is a derived period column.

Error Conditions

If the argument does not have a Period data type, an error is reported.

Examples

Example

BEGIN returns the beginning bound of a Period data type.

```
SELECT * FROM employee WHERE BEGIN(period1) = DATE '2004-06-19';
```

Assume the query is executed on the following *employee* table where period1 is a PERIOD(DATE) column:

ename	dept	period1
----	-----	-----
Jones	Sales	('2004-01-02', '2004-01-05')
Adams	Marketing	('2004-06-19', '2005-02-09')
Mary	Development	('2004-06-19', '2005-01-05')
Simon	Sales	('2004-06-22', '2005-01-07')

The result is:

ename	dept	period1
----	-----	-----
Adams	Marketing	('2004-06-19', '2005-02-09')
Mary	Development	('2004-06-19', '2005-01-05')

Example

BEGIN returns the beginning bound (*jdbegin*) of a derived period called *jobduration* created using the following SQL statement:

```
CREATE TABLE employee(id INTEGER,
                        name VARCHAR(50),
                        jdbegin DATE NOT NULL FORMAT 'YYYY-MM-DD',
                        jdend   DATE NOT NULL FORMAT 'YYYY-MM-DD',
                        PERIOD FOR jobduration(jdbegin,jdend)
)PRIMARY INDEX(id);
```

The following SQL statements:

```
INSERT INTO employee(1025,'John',DATE'2011-01-02',DATE'2012-05-02');
SELECT BEGIN (jobduration) FROM employee;
```

returns:

```
jdbegin
-----
2011-01-02
```

CONTAINS

Evaluates two Period expressions, or derived periods, or DateTime expressions to TRUE, FALSE, or UNKNOWN.

CONTAINS Syntax

```
period_spec [NOT] CONTAINS period_spec
```

Syntax Elements

period_spec

```
{ period_expression | datetime_expression | derived_period }
```

period_expression

Any expression that evaluates to a Period data type.

The Period expression must be comparable with the other expression. Implicit casting to a Period data type is not supported.

datetime_expression

An expression that evaluates to a DATE, TIME, or TIMESTAMP value.

derived_period

Any expression that evaluates to a Period data type.

Usage Notes

Error Conditions

If either expression evaluates to a data type that is other than a Period or DateTime, an error is reported.

If the expressions do not have comparable data types, an error is reported.

Examples

Example

Assume the following query is executed on the *employee* table where *period1* and *period2* are PERIOD(DATE) columns:

```
SELECT * FROM employee WHERE period2 CONTAINS period1;
```

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is:

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')

ename	period1	period2
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')

Example

Assume that in *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
  eid INTEGER NOT NULL,
  name VARCHAR(100) NOT NULL,
  deptno INTEGER NOT NULL,
  jobst1 DATE NOT NULL,
  jobend1 DATE NOT NULL,
  PERIOD FOR jobdur1(jobst1, jobend1),
  jobst2 DATE NOT NULL,
  jobend2 DATE NOT NULL,
  PERIOD FOR jobdur2(jobst2, jobend2)
) PRIMARY INDEX(eid);
```

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, CONTAINS is used with derived period columns of the employee table:

```
SELECT eid, name, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 CONTAINS jobdur2;
```

The result is:

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

END

Returns the ending bound of the period argument.

The format is the default format for the element type of the Period expression or the format of the end column if the argument is a derived period column.

Result Type

The result data type of the END function is same as the element type of the Period expression or the data type of the end column if the argument is a derived period column. If the argument is NULL, the result is NULL.

END Syntax

```
END ( { period_expression | derived_period } )
```

Syntax Elements

period_expression

The Period expression to be converted.

derived_period

Any expression that evaluates to a Period data type.

Usage Notes

Format and Title

The format is the default format for the element type of the Period expression or the format of the end column if the argument is a derived period column.

Error Conditions

If an argument of any data type other than a Period data type is passed to the function, an error is reported.

Examples

Example

Assume the following query:

```
SELECT * FROM employee WHERE END(period1) = DATE '2005-01-07';
```

executes on the *employee* table with PERIOD(DATE) column period1:

ename	dept	period1
-----	-----	-----
Jones	Sales	('2004-01-02', '2004-01-05')
Adams	Marketing	('2004-06-19', '2005-02-09')
Mary	Development	('2004-06-19', '2005-01-05')
Simon	Sales	('2004-06-22', '2005-01-07')

The result is:

ename	dept	period1
-----	-----	-----
Simon	Sales	('2004-06-22', '2005-01-07')

Example

END returns the ending bound (*jdend*) of a derived period called *jobduration* created using the following SQL statement:

```
CREATE TABLE employee(id INTEGER,
                        name VARCHAR(50),
                        jdbegin DATE NOT NULL FORMAT 'YYYY-MM-DD',
                        jdend   DATE NOT NULL FORMAT 'YYYY-MM-DD',
                        PERIOD FOR jobduration(jdbegin,jdend)
)PRIMARY INDEX(id);
```

The following SQL statements:

```
INSERT INTO employee(1025, 'John', DATE '2011-01-02', DATE '2012-05-02');
SELECT END (jobduration) FROM employee;
```

return is:

jded
2011-05-02

EQUALS

Evaluates two period expressions or derived periods to TRUE, FALSE, or UNKNOWN.

Result Type

- If both expressions have a Period data type or a derived period, the function returns TRUE if the beginning and ending bound of the first expression is equal to the beginning and ending bound of the second expression; otherwise, the function returns FALSE.
- If either operand is NULL, the operator returns UNKNOWN.

EQUALS Syntax

```
period_spec [NOT] EQUALS period_spec
```

Syntax Elements

period_spec

```
{ period_expression | derived_period }
```

period_expression

Any expression that evaluates to a Period data type.

The Period expression must be comparable with the other expression. Implicit casting to a Period data type is not supported.

derived_period

Any expression that evaluates to a Period data type.

Examples

Example

Assume that in the *employee1* table, created by the following CREATE TABLE statement, period1 and period2 are PERIOD (DATE) columns.

```
CREATE TABLE employee1 (
    eid INTEGER NOT NULL,
    name VARCHAR(100) NOT NULL,
    deptno INTEGER NOT NULL,
    period1(date),
    period2(date)
) PRIMARY INDEX(eid);
```

EID	Name	DeptNo	Period 1	Period 2
1	Adams	101	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
2	Mary	201	('2005-04-02', '2006-01-03')	('2006-01-03', '2007-02-03')
3	Jones	301	('2004-01-02', '2005-03-05')	('2003-03-05', '2004-01-02')

In the following SQL statement, EQUALS is used with period columns of the employee1 table:

```
SELECT eid, name, depno, period1, period2
FROM employee1
WHERE period1 EQUALS period2;
```

The result is:

EID	Name	DeptNo	Period 1	Period 2
1	Adams	101	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')

Example

Assume that in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
    eid INTEGER NOT NULL,
```

```

name VARCHAR(100) NOT NULL,
deptno INTEGER NOT NULL,
jobst1 DATE NOT NULL,
jobend1 DATE NOT NULL,
PERIOD FOR jobdur1(jobst1, jobend1),
jobst2 DATE NOT NULL,
jobend2 DATE NOT NULL,
PERIOD FOR jobdur2(jobst2, jobend2)
) PRIMARY INDEX(eid);

```

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, EQUALS is used with derived period columns of the employee table:

```

SELECT eid, name, deptno, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 EQUALS jobdur2;

```

The result is:

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

IS UNTIL_CHANGED/IS NOT UNTIL_CHANGED

Tests whether the ending bound value of a Period expression or a derived period IS UNTIL_CHANGED or IS NOT UNTIL_CHANGED.

IS UNTIL_CHANGED/IS NOT UNTIL_CHANGED Syntax

```

END ( { period_expression | derived_period } )
IS [NOT] UNTIL_CHANGED

```

Syntax Elements

period_expression

Any expression that evaluates to a Period data type, including:

- PERIOD(TIMESTAMP WITH TIME ZONE)
- PERIOD(TIMESTAMP)
- PERIOD(DATE) type

derived_period

Any expression that evaluates to a Period data type.

Usage Notes

You can only compare UNTIL_CHANGED to the ending bound of a Period value with an element type of DATE or TIMESTAMP WITH TIME ZONE. Therefore, the result type of the END function must be DATE or TIMESTAMP WITH TIME ZONE. For information about the END function, see [END](#).

In comparisons, the precision of the UNTIL_CHANGED value is truncated to the precision of the ending bound value being compared. That is, the number of digits after the decimal point for UNTIL_CHANGED depends upon the precision of the ending bound to which it is compared. The time zone is omitted if the ending bound value has no time zone.

If the ending bound value is NULL, IS [NOT] UNTIL_CHANGED returns UNKNOWN.

You cannot use IS [NOT] UNTIL_CHANGED on the ending bound of a system-time or temporal table columns.

Examples

Example

Consider the following employee table, where the column *eduration* is defined as a PERIOD(DATE) data type:

ename	eid	eduration
-----	-----	-----
Adams	210677	('05/03/01', '06/05/21')
Gunther	199347	('04/06/06', '99/12/31')
Montoya	199340	('04/06/02', '99/12/31')
Chan	210427	('04/09/24', '99/12/31')
Fuller	197899	('03/05/27', '03/11/30')

The following SQL query uses `IS UNTIL_CHANGED` to compare the ending bound value of the *eduration* column to `UNTIL_CHANGED`:

```
SELECT ename, eid
FROM employee
WHERE END(eduration) IS UNTIL_CHANGED;
```

The result is:

ename	eid
Gunther	199347
Montoya	199340
Chan	210427

Example

If we assume that the *employee* table, created by the following SQL statement:

```
CREATE MULTISET TABLE employee(empno    INTEGER,
                                ename     VARCHAR(50),
                                deptno    INTEGER,
                                jobstart  DATE NOT NULL,
                                jobend    DATE NOT NULL,
                                PERIOD FOR jobduration(jobstart, jobend))
PRIMARY INDEX(empno);
```

contains the following row:

```
INSERT INTO employee(1025, 'John', '999', DATE'2005-02-03', UNTIL_CHANGED);
```

The following `SELECT` statement:

```
SELECT empno,ename(CHAR(30)) FROM employee WHERE END(jobduration)
IS UNTIL_CHANGED;
```

returns:

empno	ename
1025	John

IS UNTIL_CLOSED/IS NOT UNTIL_CLOSED

Tests the ending bound value of a temporal table system-time or transaction-time column to see whether the row is open (the ending bound value IS UNTIL_CLOSED) or closed (the ending bound value IS NOT UNTIL_CLOSED).

IS UNTIL_CLOSED/IS NOT UNTIL_CLOSED Syntax

```
END ( { period_expression | derived_period } )
    IS [NOT] UNTIL_CLOSED
```

Syntax Elements

period_expression

The Period expression to be converted.

derived_period

Any expression that evaluates to a Period data type.

Usage Notes

When a row is created in a temporal table that has a system-time or transaction-time dimension (column), Vantage sets the ending bound of the column to UNTIL_CLOSED and the row is considered open. When the row is closed, Vantage sets the ending bound value to the closing timestamp.

IS UNTIL_CLOSED evaluates to true if the ending bound of the specified transaction-time column is the maximum timestamp value, 9999-12-31 23:59:59.999999+00:00.

Examples

Example

```
CREATE MULTISET TABLE employee(
  empno INTEGER,
  ename VARCHAR(50),
  deptno INTEGER,
  jobstart TIMESTAMP WITH TIME ZONE NOT NULL
    GENERATED ALWAYS AS ROW START,
  jobend TIMESTAMP WITH TIME ZONE NOT NULL AS
    GENERATED ALWAYS AS ROW END,
  PERIOD FOR SYSTEM_TIME(jobstart,jobend)
```

```
)
PRIMARY INDEX(empno) WITH SYSTEM VERSIONING;
```

Assume the table contains the following row:

empno	ename	deptno	jobstart	jobend
1025	John	999	2005-02-03 12:12:12.123456+00:00	9999-12-31 23:59:59.999999+00:00

The following SELECT statements would give these results:

```
SELECT empno,ename(CHAR(6)) FROM employee
WHERE END(SYSTEM_TIME) IS UNTIL_CLOSED;
```

```
empno ename
-----
1025 John
```

```
SELECT empno,ename(CHAR(6)) FROM employee
WHERE END(SYSTEM_TIME) IS NOT UNTIL_CLOSED;
```

```
*** Query completed. No rows found.
```

Example

If we assume that the *employee* table, created by the following SQL statement:

```
CREATE MULTISET TABLE employee(
  id INTEGER,
  name VARCHAR(50),
  dept INTEGER,
  tt PERIOD(TIMESTAMP(6)WITH TIME ZONE) NOT NULL AS TRANSACTIONTIME
)
PRIMARY INDEX(id);
```

contains the following row, a real period:

```
INSERT INTO employee(102, 'John', 222);
```

The following SELECT statement:

```
SELECT * FROM employee WHERE END(tt) IS UNTIL_CLOSED;
```

returns:

id	name	dept
102	John	222

IMMEDIATELY PRECEDES

Evaluates two Period expressions or derived periods.

Result Type

This predicate applies when both *period_expression_1* and *period_expression_2* are a period column, period constructor, or period value expression. In this case, the predicate returns *True* if the end bound value of *period_expression_1* is equal to the begin bound value of *period_expression_2*. If either or both of the operands are NULL, the result is UNKNOWN.

IMMEDIATELY PRECEDES Syntax

```
{ period_expression_1 IMMEDIATELY PRECEDES { period_expression_2 | derived_period } |
  derived_period_1 IMMEDIATELY PRECEDES { derived_period_2 | period_expression }
}
```

Syntax Elements

period_expression_1/period_expression_2

Any expression that evaluates to a Period data type.

The Period expression must be comparable with the other expression. Implicit casting to a Period data type is not supported.

derived_period_1/derived_period_2

Any expression that evaluates to a Period data type.

Examples

Example

Assume that in *employee1* table, created by the following CREATE TABLE statement, *period1* and *period2* are PERIOD (DATE) columns.

```
CREATE TABLE employee1 (
    eid INTEGER NOT NULL,
    name VARCHAR(100) NOT NULL,
    deptno INTEGER NOT NULL,
    period1(date),
    period2(date)
) PRIMARY INDEX(eid);
```

EID	Name	DeptNo	Period 1	Period 2
1	Adams	101	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
2	Mary	201	('2005-04-02', '2006-01-03')	('2006-01-03', '2007-02-03')
3	Jones	301	('2004-01-02', '2005-03-05')	('2003-03-05', '2004-01-02')

In the following SQL statement, IMMEDIATELY PRECEDES is used with period columns of the employee1 table:

```
SELECT eid, name, depno, period1, period2
FROM employee1
WHERE period1 IMMEDIATELY PRECEDES period2;
```

The result is:

EID	Name	DeptNo	Period 1	Period 2
2	Mary	201	('2005-04-02', '2006-01-03')	('2006-01-03', '2007-02-03')

Example

Assume that in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
    eid INTEGER NOT NULL,
    name VARCHAR(100) NOT NULL,
    deptno INTEGER NOT NULL,
    jobst1 DATE NOT NULL,
    jobend1 DATE NOT NULL,
    PERIOD FOR jobdur1(jobst1, jobend1),
    jobst2 DATE NOT NULL,
    jobend2 DATE NOT NULL,
```

```
PERIOD FOR jobdur2(jobst2, jobend2)
) PRIMARY INDEX(eid);
```

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

IMMEDIATELY SUCCEEDS

Evaluates two Period expressions or derived periods to TRUE, FALSE, or UNKNOWN.

Result Type

This predicate applies when both *period_expression_1* and *period_expression_2* are a period column, period constructor, or period value expression. In this case, the predicate returns *True* if the begin bound value of *period_expression_1* is equal to the end bound value of *period_expression_2*. If either or both of the operands are NULL, the result is UNKNOWN.

If either expression is NULL, the operator returns UNKNOWN.

IMMEDIATELY SUCCEEDS Syntax

```
{ period_expression_1 IMMEDIATELY SUCCEEDS { period_expression_2 | derived_period } |
  derived_period_1 IMMEDIATELY SUCCEEDS { derived_period_2 | period_expression }
}
```

Syntax Elements

period_expression1/period_expression2

Any expression that evaluates to a Period data type.

The Period expression must be comparable with the other expression. Implicit casting to a Period data type is not supported.

derived_period1/derived_period2

Any expression that evaluates to a Period data type.

Examples

Example

Assume that in *employee1* table, created by the following CREATE TABLE statement, period1 and period2 are PERIOD (DATE) columns

```
CREATE TABLE employee1 (
    eid INTEGER NOT NULL,
    name VARCHAR(100) NOT NULL,
    deptno INTEGER NOT NULL,
    period1(date),
    period2(date)
) PRIMARY INDEX(eid);
```

EID	Name	DeptNo	Period 1	Period 2
1	Adams	101	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
2	Mary	201	('2005-04-02', '2006-01-03')	('2006-01-03', '2007-02-03')
3	Jones	301	('2004-01-02', '2005-03-05')	('2003-03-05', '2004-01-02')

In the following SQL statement, IMMEDIATELY SUCCEEDS is used with period columns of the employee1 table:

```
SELECT eid, name, deptno, period1, period2
FROM employee1
WHERE period1 IMMEDIATELY SUCCEEDS period2;
```

The result is:

EID	Name	DeptNo	Period 1	Period 2
3	Jones	301	('2004-01-02', '2005-03-05')	('2003-03-05', '2004-01-02')

Example

Assume that in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
  eid INTEGER NOT NULL,
  name VARCHAR(100) NOT NULL,
  deptno INTEGER NOT NULL,
  jobst1 DATE NOT NULL,
  jobend1 DATE NOT NULL,
  PERIOD FOR jobdur1(jobst1, jobend1),
  jobst2 DATE NOT NULL,
  jobend2 DATE NOT NULL,
  PERIOD FOR jobdur2(jobst2, jobend2)
) PRIMARY INDEX(eid);
```

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, IMMEDIATELY SUCCEEDS is used with derived period columns of the employee table:

```
SELECT eid, name, depno, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 IMMEDIATELY SUCCEEDS jobdur2;
```

The result is:

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'

INTERVAL

Finds the difference between the ending and beginning bounds of a Period expression or a derived period and returns the difference as the duration of the argument in a specified interval qualifier.

Result Type

The result of the INTERVAL (p) IQ function is the value of (END(p) - BEGIN(p)) IQ, where argument p is a Period expression and IQ is an interval qualifier.

If the argument is NULL, the result is NULL.

INTERVAL Syntax

```
INTERVAL ( { period_expression | derived_period } ) interval_qualifier
```

Syntax Elements

period_expression

The Period expression to be converted.

derived_period

Any expression that evaluates to a Period data type.

interval_qualifier

Any interval qualifier appropriate for the argument's element type. The interval qualifiers are as follows:

- Year-Month intervals:
 - YEAR
 - YEAR TO MONTH
 - MONTH
- Day-Time intervals:
 - DAY
 - DAY TO HOUR, MINUTE or SECOND
 - HOUR
 - HOUR TO MINUTE or SECOND
 - MINUTE
 - MINUTE to SECOND
 - SECOND

Usage Notes

Format and Title

The format is the default format for the interval data type corresponding to the specified interval qualifier.

The title is `INTERVAL(period_expression) interval_qualifier`.

Error Conditions

An error may be reported:

- If the argument of the `INTERVAL` function does not have a Period data type.
- If the argument has a `PERIOD(DATE)` data type and the interval qualifier is not `YEAR`, `YEAR TO MONTH`, `MONTH`, or `DAY`.
- If the argument has a `PERIOD(TIME(n) [WITH TIME ZONE])` data type and the interval qualifier is not `HOUR`, `HOUR TO MINUTE`, `HOUR TO SECOND`, `MINUTE`, `MINUTE TO SECOND` or `SECOND`.
- If the result of an `INTERVAL` expression violates the rules specified for the precision of an interval qualifier, an error is reported. For example, assume `p1` is a `PERIOD(TIMESTAMP(0))` expression that has a value of `PERIOD ' (2006-01-01 12:12:12, 2007-01-01 12:12:12) '`. If `INTERVAL(p1) DAY` is specified, the default precision for the `DAY` interval qualifier is 2, and, since the result is 365 days which is a three-digit value that cannot fit into a `DAY(2)` interval qualifier, an error is reported.
- If the argument of the `INTERVAL` function is a period of a `DATE` or `TIMESTAMP(n) [WITH TIME ZONE]` and the ending bound value is `UNTIL_CHANGED`.

Examples

Example: Using the INTERVAL Function

The following example uses the `INTERVAL` function:

```
SELECT INTERVAL(PERIOD(
DATE '2016-02-29',
DATE '2016-03-01')) MONTH ;

INTERVAL(PERIOD(2016-02-29, 2016-03-01)) MONTH
-----
1
```

The result is an `INTERVAL` value.

Example: Executing a Query on the Employee Table

Assume the following query is executed on the *employee* table with PERIOD(DATE) column period1:

```
SELECT INTERVAL (period1) MONTH FROM employee;
ename    dept      period1
-----  -
Jones    Sales      ('2004-01-02', '2004-03-05')
```

The result is:

```
INTERVAL(period) MONTH
-----
2
```

Example: Returning the Interval Value of a Derived Period

INTERVAL returns the interval value of a derived period called *jobduration* created using the following SQL statement:

```
CREATE TABLE employee(id INTEGER,
                        name VARCHAR(50),
                        jdbegin DATE NOT NULL FORMAT 'YYYY-MM-DD',
                        jdend   DATE NOT NULL FORMAT 'YYYY-MM-DD',
                        PERIOD FOR jobduration(jdbegin,jdend)
)PRIMARY INDEX(id);
```

When the following values are inserted into the *employee* table:

```
INSERT INTO employee(1025,'John',DATE'2011-01-02',DATE'2012-05-02');
```

the following SQL statement:

```
SELECT INTERVAL(jobduration)MONTH(4) FROM employee;
```

returns:

```
INTERVAL(jobduration) MONTH
-----
16
```

LAST

Returns the last value of the Period argument (that is, the ending bound minus one granule of the element type of the argument).

Result Type

The result type of the LAST function is same as the element type of the Period expression or the data type of the begin or end column if the argument is a derived period. If the argument is NULL, the result is NULL.

LAST Syntax

```
LAST ( { period_expression | derived_period } )
```

Syntax Elements

period_expression

The Period expression to be converted.

derived_period

Any expression that evaluates to a Period data type.

Usage Notes

Format and Title

The format is the default format for the element type of the Period expression or the format of the begin or end column if the argument is a derived period.

Error Conditions

If an argument has a data type other than a Period data type, an error is reported.

Examples

Example

Assume the following query is executed on the employee table PERIOD(DATE) column period1:

```
SELECT * FROM employee WHERE LAST(period1) = DATE '2004-01-04';
ename   dept      period1
-----  -
Jones   Sales      ('2004-01-02', '2004-01-05')
Adams   Marketing  ('2004-06-19', '2005-02-09')
Mary    Development ('2004-06-19', '2005-01-05')
Simon   Sales      ('2004-06-22', '2005-01-07')
```

The result is:

```
ename   dept      period1
-----  -
Jones   Sales      ('2004-01-02', '2004-01-05')
```

Example

LAST returns the last value of a derived period called jobduration created by the following SQL statement:

```
CREATE TABLE employee(id INTEGER,
                        name VARCHAR(50),
                        jdbegin DATE NOT NULL FORMAT 'YYYY-MM-DD',
                        jdend   DATE NOT NULL FORMAT 'YYYY-MM-DD',
                        PERIOD FOR jobduration(jdbegin,jdend)
)PRIMARY INDEX(id);
```

When the following values are inserted into the employee table:

```
INSERT INTO employee(1025, 'John', DATE '2011-01-02', DATE '2012-05-02');
```

the following SQL statement:

```
SELECT LAST (jobduration) FROM employee;
```

returns:

```
LAST(jobduration)
-----
2012-05-01
```

MEETS

Evaluates two period expressions, or derived periods, or DateTime expressions and evaluates to TRUE, FALSE, or UNKNOWN.

Result Type

- If both expressions have a Period data type or a derived period, the function returns TRUE if the ending bound of the first expression is equal to the beginning bound of the second expression or the ending bound of the second expression is equal to the beginning bound of the first expression; otherwise, the function returns FALSE.
- If one expression is a Period data type and the other expression is a DateTime expression, the function returns TRUE if the ending bound of the Period expression is equal to the DateTime expression or if the DateTime expression plus one granule is equal to the beginning bound of the Period expression; otherwise, the function returns FALSE.
- If either expression is NULL, the function returns UNKNOWN.

MEETS Syntax

```
{ period_expression [NOT] MEETS
  { period_expression | datetime_expression | derived_period } |

  datetime_expression [NOT] MEETS period_expression |

  derived_period [NOT] MEETS { derived_period | period_expression }
}
```

Syntax Elements

period_expression

Any expression that evaluates to a Period data type.

The Period expression must be comparable with the other expression. Implicit casting to a Period data type is not supported.

datetime_expression

An expression that evaluates to a DATE, TIME, or TIMESTAMP value.

derived_period

Any expression that evaluates to a Period data type.

Usage Notes

Error Conditions

If either expression evaluates to a data type other than a Period or DateTime, an error is reported.
 If the expressions are not comparable, an error is reported.

Examples

Example

Assume the following query is executed on the *employee* table where period1 and period2 are PERIOD(DATE) columns:

```
SELECT * FROM employee WHERE period2 MEETS period1;
```

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is:

ename	period1	period2
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')

Example: Using MEETS

Assume that in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
    eid INTEGER NOT NULL,
    name VARCHAR(100) NOT NULL,
```

```

deptno INTEGER NOT NULL,
jobst1 DATE NOT NULL,
jobend1  DATE NOT NULL,
PERIOD FOR jobdur1(jobst1, jobend1),
jobst2 DATE NOT NULL,
jobend2  DATE NOT NULL,
PERIOD FOR jobdur2(jobst2, jobend2)
) PRIMARY INDEX(eid);

```

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, MEETS is used with derived period columns of the employee table:

```

SELECT eid, name, deptno, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 MEETS jobdur2;

```

The result is:

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'

NEXT

Returns the succeeding value of the argument so that there is one granule of the argument type between the argument and the returned value.

Result Type

The return data type is the same as that of the argument. If the value of the argument is NULL, the result is NULL.

NEXT Syntax

```
NEXT ( datetime_expression )
```

Syntax Elements

datetime_expression

An expression that evaluates to a DATE, TIME, or TIMESTAMP value.

Usage Notes

Format and Title

The format is the default format for the data type of the proximity argument.

Error Conditions

If the argument does not have a DateTime data type, an error is reported.

If the result is outside the permissible range of a value for the argument's data type, an error is reported. For example, if NEXT(DATE '9999-12-31') is specified, an error is reported.

Example

Assume the following query is executed on the *employee* table where period1 is a PERIOD(DATE) column:

```
SELECT *
FROM employee
WHERE NEXT(END(period1)) = DATE '2004-03-06';
ename    dept      period1
-----  -
Jones    Sales      ('2004-01-02', '2004-03-05')
Simon    Sales      ?
```

The result is:

```
ename    dept      period1
-----  -
Jones    Sales      ('2004-01-02', '2004-03-05')
```


OVERLAPS

Tests whether two time periods overlap each another.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

OVERLAPS Syntax

```
time_period_expression OVERLAPS time_period_expression
```

Syntax Elements

time_period_expression

```
{ ( { datetime_expression, { datetime_expression | interval_expression } |  
    row_subquery  
  } ) |  
  period_expression |  
  derived_expression  
}
```

datetime_expression

A start and end DateTime.

interval_expression

An end DateTime.

row_subquery

A subquery that selects the same number of expressions as are specified in the expression or list of expressions.

The subquery cannot specify a SELECT AND CONSUME statement.

period_expression

The Period expression to be converted.

derived_period

Any expression that evaluates to a Period data type.

Result Type

Consider the general case of an OVERLAPS comparison, stated as follows.

```
(S1, E1) OVERLAPS (S2, E2)
```

The result of OVERLAPS is as follows.

```
(S1 > S2 AND NOT (S1 >= E2 AND E1 >= E2))  
OR  
(S2 > S1 AND NOT (S2 >= E1 AND E2 >= E1))  
OR  
(S1 = S2 AND (E1 = E2 OR E1 <> E2))
```

For Period data types or derived periods, where p1 is the first Period expression or derived period and p2 is the second Period expression or derived period, the values of S1, E1, S2, and E2 are as follows:

```
S1 = BEGIN(p1)  
E1 = END(p1)  
S2 = BEGIN(p2)  
E2 = END(p2)
```

Usage Notes

Time Periods

If the start and end DateTime values in a time period are not ordered chronologically, they are manipulated to make them so prior to making the comparison, using the rule that *end_DateTime* >= *start_DateTime* for all cases.

If a time period contains a null *start_DateTime* and a non-null *end_DateTime*, then the values are switched to indicate a non-null *start_DateTime* and a null *end_DateTime*.

Note:

Implicit casting to a Period data type is not supported.

Rules

- When you specify two DateTime types, they must be comparable.
- When you specify two Period types, including derived periods, they must be comparable.

- If you specify a Period type for either one or both time periods, the Period expression must not include an explicit NULL.
- If the first columns of each left and right time periods are DateTime types, they must have the same data type: both DATE, both TIME, or both TIMESTAMP.
- If only one time period is a Period type, the first column of the other time period must have the same data type as the element type of the Period.
- If neither time period is a Period type, then the second column of each left and right time period must either be the same DateTime type as its corresponding first column or it must be an Interval type that involves only DateTime fields where the precision is such that its value can be added to that of the corresponding DateTime type.

Examples

Example

The following example compares two time periods that share a single common point, CURRENT_TIME. The result returned is FALSE because when two time periods share a single point, they do not overlap by definition.

```
SELECT 'OVERLAPS'
WHERE (CURRENT_TIME(0), INTERVAL '1' HOUR)
OVERLAPS (CURRENT_TIME(0), INTERVAL -'1' HOUR);
```

The following example is nearly identical to the previous one, except that the arguments have been adjusted to overlap by one second. The result is TRUE and the value 'OVERLAPS' is returned.

```
SELECT 'OVERLAPS'
WHERE (CURRENT_TIME(0), INTERVAL '1' HOUR)
OVERLAPS (CURRENT_TIME(0) + INTERVAL '1' SECOND, INTERVAL -'1' HOUR);
```

The following example uses the *datetime_expression, datetime_expression* form of OVERLAPS. The two DATE periods overlap each other; thus the result is TRUE.

```
SELECT 'OVERLAPS'
WHERE (DATE '2000-01-15', DATE '2002-12-15')
OVERLAPS (DATE '2001-06-15', DATE '2005-06-15');
```

The following example is the same as the previous one, but in *row_subquery* form:

```
SELECT 'OVERLAPS'
WHERE (SELECT DATE '2000-01-15', DATE '2002-12-15')
OVERLAPS (SELECT DATE '2001-06-15', DATE '2005-06-15');
```

The NULL in the following example means the second *datetime_expression* has a start time of 2001-06-13 15:00:00 and a null end time.

```
SELECT 'OVERLAPS'
WHERE (TIMESTAMP '2001-06-12 10:00:00', TIMESTAMP '2001-06-15 08:00:00')
OVERLAPS (TIMESTAMP '2001-06-13 15:00:00', NULL);
```

Because the start time for the second expression falls within the TIMESTAMP interval defined by the first expression, the result is TRUE.

Assume the following query is executed on the *employee* table where period1 and period2 are PERIOD(DATE) columns:

```
SELECT * FROM employee WHERE period2 OVERLAPS period1;
```

Ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is:

Ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')

Consider the following table and query:

```
CREATE TABLE project
(id INTEGER,
 analysis_phase PERIOD(DATE))
UNIQUE PRIMARY INDEX (id);
INSERT project (1, PERIOD(DATE '2010-06-21', DATE '2010-06-25'));
SELECT 'OVERLAPS'
```

```
FROM project
WHERE analysis_phase OVERLAPS
      PERIOD(DATE '2010-06-24', NULL);
```

The SELECT statement returns an error because one of the operands of OVERLAP is a Period type with a Period expression specifying an explicit NULL.

Assume that in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
  eid INTEGER NOT NULL,
  name VARCHAR(100) NOT NULL,
  deptno INTEGER NOT NULL,
  jobst1 DATE NOT NULL,
  jobend1 DATE NOT NULL,
  PERIOD FOR jobdur1(jobst1, jobend1),
  jobst2 DATE NOT NULL,
  jobend2 DATE NOT NULL,
  PERIOD FOR jobdur2(jobst2, jobend2)
) PRIMARY INDEX(eid);
```

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, OVERLAPS is used with derived period columns of the employee table:

```
SELECT eid, name, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 OVERLAPS jobdur2;
```

The result is:

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

Example

The following example is nearly identical to the previous one, except that the arguments have been adjusted to overlap by one second. The result is TRUE and the value 'OVERLAPS' is returned.

```
SELECT 'OVERLAPS'
WHERE (CURRENT_TIME(0), INTERVAL '1' HOUR)
OVERLAPS (CURRENT_TIME(0) + INTERVAL '1' SECOND, INTERVAL -'1' HOUR);
```

Example

The following example uses the *datetime_expression, datetime_expression* form of OVERLAPS. The two DATE periods overlap each other; thus the result is TRUE.

```
SELECT 'OVERLAPS'
WHERE (DATE '2000-01-15', DATE '2002-12-15')
OVERLAPS (DATE '2001-06-15', DATE '2005-06-15');
```

Example

The following example is the same as the previous one, but in *row_subquery* form:

```
SELECT 'OVERLAPS'
WHERE (SELECT DATE '2000-01-15', DATE '2002-12-15')
OVERLAPS (SELECT DATE '2001-06-15', DATE '2005-06-15');
```

Example

The NULL in the following example means the second *datetime_expression* has a start time of 2001-06-13 15:00:00 and a null end time.

```
SELECT 'OVERLAPS'
WHERE (TIMESTAMP '2001-06-12 10:00:00', TIMESTAMP '2001-06-15 08:00:00')
OVERLAPS (TIMESTAMP '2001-06-13 15:00:00', NULL);
```

Because the start time for the second expression falls within the `TIMESTAMP` interval defined by the first expression, the result is `TRUE`.

Example

Assume the following query is executed on the `employee` table where `period1` and `period2` are `PERIOD(DATE)` columns:

```
SELECT * FROM employee WHERE period2 OVERLAPS period1;
```

Ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is:

Ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')

Example

Consider the following table and query:

```
CREATE TABLE project
  (id INTEGER,
   analysis_phase PERIOD(DATE))
UNIQUE PRIMARY INDEX (id);
INSERT project (1, PERIOD(DATE '2010-06-21', DATE '2010-06-25'));
SELECT 'OVERLAPS'
FROM project
WHERE analysis_phase OVERLAPS
      PERIOD(DATE '2010-06-24', NULL);
```

The SELECT statement returns an error because one of the operands of OVERLAP is a Period type with a Period expression specifying an explicit NULL.

Example

Assume that in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
  eid INTEGER NOT NULL,
  name VARCHAR(100) NOT NULL,
  deptno INTEGER NOT NULL,
  jobst1 DATE NOT NULL,
  jobend1 DATE NOT NULL,
  PERIOD FOR jobdur1(jobst1, jobend1),
  jobst2 DATE NOT NULL,
  jobend2 DATE NOT NULL,
  PERIOD FOR jobdur2(jobst2, jobend2)
) PRIMARY INDEX(eid);
```

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, OVERLAPS is used with derived period columns of the employee table:

```
SELECT eid, name, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 OVERLAPS jobdur2;
```

The result is:

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

P_INTERSECT

Returns the portion of the Period expression that is common to the Period expressions if they overlap.

P_INTERSECT Syntax

```
period_expression P_INTERSECT period_expression
```

Syntax Elements

period_expression

Any expression that evaluates to a Period data type.

The Period expression must be comparable with the other expression. Implicit casting to a Period data type is not supported.

Result Type

- If the Period expressions do not overlap, the result is NULL. If either Period expression is NULL, the result is NULL. Otherwise, the result has a Period data type that is comparable to the Period expressions.
- If the Period expressions have PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) or PERIOD(TIME(n) [WITH TIME ZONE]) data types but different precisions, the result is a Period value of the higher precision data type. If neither Period expression has a time zone, the resulting period does not have a time zone; otherwise, the resulting period has a time zone and the value of the time zone in the result is determined using the following rules:
 - If both Period expressions have a time zone, the time zone displacement of a result bound is obtained from the corresponding bound of the Period expression as defined by the Period value constructor that follows.
 - If only one of the Period expressions has a time zone, the other Period expression is considered to be at the current session time zone and the result is computed as follows.

Assuming *p1* and *p2* are Period expressions and the result element type as determined above is *rt*, the result of *p1* P_INTERSECT *p2* is as follows if *p1* OVERLAPS *p2* is TRUE:

```
PERIOD(
  CASE WHEN CAST(BEGIN(p1) AS rt) >= CAST(BEGIN(p2) AS rt)
    THEN CAST(BEGIN(p1) AS rt)
```

```
ELSE CAST(BEGIN(p2) AS rt) END,
CASE WHEN CAST(END(p1) AS rt) <= CAST(END(p2) AS rt)
THEN CAST(END(p1) AS rt)
ELSE CAST(END(p2) AS rt) END)
```

Internally, Period values are saved in UTC and the OVERLAPS operator is evaluated using these UTC represented formats and the P_INTERSECT operation is performed if they overlap.

Usage Notes

Format and Title

The format is the default format for the resulting Period data type.

Error Conditions

If either expression is not a Period expression, an error is reported.

If the Period expressions are not comparable, an error is reported.

Examples

Example

ename	period1	period2
Adams	('2005-02-03 10:10:10.1', '2007-02-03 10:10:10.1')	('2004-02-03 10:10:10', '2006-02-03 10:10:10')
Paul	('2004-02-03 10:10:10.1', '2006-02-03 10:10:10.1')	('2005-02-03 10:10:10', '2007-02-03 10:10:10')
James	('2004-03-03 10:10:10.1', '2006-01-03 10:10:10.1')	('2004-02-03 10:10:10', '2006-02-03 10:10:10')
Mary	('2007-04-02 10:10:10.1', '2008-01-03 10:10:10.1')	('2005-02-03 10:10:10', '2006-02-03 10:10:10')

(period2 P_INTERSECT period1)
('2005-02-03 10:10:10.1', '2006-02-03 10:10:10.0')

Example

In the following example, the P_INTERSECT operator is used in the selection list.

```
SELECT period2 P_INTERSECT period1
FROM product_tests
WHERE pid = 11804;
```

Assume the query is executed on the following table *product_tests* where period1 is a PERIOD(TIME(1)) column and period2 is a PERIOD(TIME(0)) column:

pid	period1	period2
-----	-----	-----
11804	('10:10:10.1', '11:10:10.1')	('10:10:10', '10:10:11')
10996	('11:10:10.1', '11:40:40.1')	('10:10:10', '10:10:11')

The result is as follows:

```
(period2 P_INTERSECT period1)
-----
('10:10:10.1', '10:10:11.0')
```

PRECEDES

Evaluates two Period expressions or derived periods, or DateTime expressions to TRUE, FALSE, or UNKNOWN.

PRECEDES Syntax

```
{ period_expression [NOT] PRECEDES
  { period_expression | datetime_expression | derived_period } |

  datetime_expression [NOT] PRECEDES period_expression |

  derived_period [NOT] PRECEDES { derived_period | period_expression }
}
```

Syntax Elements

period_expression

Any expression that evaluates to a Period data type.

The Period expression must be comparable with the other expression. Implicit casting to a Period data type is not supported.

datetime_expression

An expression that evaluates to a DATE, TIME, or TIMESTAMP value.

derived_period

Any expression that evaluates to a Period data type.

Result Type

- If both expressions have a Period data type or a derived period, the function returns TRUE if the ending bound of the first expression is less than or equal to the beginning bound of the second expression; otherwise, the function returns FALSE.
- If the first expression is a Period expression and the second expression is a DateTime expression, the function returns TRUE if the ending bound of the first expression is less than or equal to the second expression; otherwise, the function returns FALSE.
- If the first expression is a DateTime expression and the second expression has a Period data type, the function returns TRUE if the first expression is less than the beginning bound of the second expression; otherwise, the function returns FALSE.
- If either expression is NULL, the operator returns UNKNOWN.

Usage Notes

Error Conditions

If either expression is other than a Period data type or a DateTime value expression, an error is reported.

If the Period expressions are not comparable, an error is reported.

Examples

Example

Assume the following query is executed on the *employee* table where period1 and period2 are PERIOD(DATE) columns:

```
SELECT * FROM employee WHERE period1 PRECEDES period2;
```

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is:

ename	period1	period2
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')

Example

Assume in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
  eid INTEGER NOT NULL,
  name VARCHAR(100) NOT NULL,
  deptno INTEGER NOT NULL,
  jobst1 DATE NOT NULL,
  jobend1 DATE NOT NULL,
  PERIOD FOR jobdur1(jobst1, jobend1),
  jobst2 DATE NOT NULL,
  jobend2 DATE NOT NULL,
  PERIOD FOR jobdur2(jobst2, jobend2)
) PRIMARY INDEX(eid);
```

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, PRECEDES is used with derived period columns of the employee table:

```
SELECT eid, name, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 PRECEDES jobdur2;
```

The result is:

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'

PRIOR

Returns the preceding value of the argument so that there is one granule of the argument type between the returned value and the argument.

Result Type

The return data type is the same as that of the argument. If the value of the argument is NULL, the result is NULL.

PRIOR Syntax

```
PRIOR ( datetime_expression )
```

Syntax Elements

datetime_expression

An expression that evaluates to a DATE, TIME, or TIMESTAMP value.

Usage Notes

Format and Title

The format is the default format for the argument's data type.

Restrictions

The built-in function `DATE` is not supported as an argument to the `PRIOR` function. For example, `PRIOR(DATE)` is not valid. You must use `PRIOR(CURRENT_DATE)` instead.

Error Conditions

If the argument does not have a `DateTime` data type, an error is reported.

If the result is outside the permissible range of the argument's data type, an error is reported. For example, if `PRIOR(DATE '0001-01-01')` is specified, an error is reported.

Example

Assume the following query is executed on the *employee* table where *period1* is a `PERIOD(DATE)` column:

```
SELECT *
FROM employee
WHERE PRIOR(END(period1)) = DATE '2004-03-04';
ename    dept      period1
-----  -
Jones    Sales    ('2004-01-02', '2004-03-05')
Simon    Sales    ?
```

The result is:

```
ename    dept      period1
-----  -
Jones    Sales    ('2004-01-02', '2004-03-05')
```

LDIFF

Returns the portion of the first Period expression that exists before the beginning of the second Period expression when the Period expressions overlap.

When the Period expressions overlap but there is no portion of the first Period expression before the beginning of the second Period expression or the Period expressions do not overlap, the function returns NULL. If either Period expression is NULL, LDIFF returns NULL.

LDIFF Syntax

```
period_expression LDIFF period_expression
```

Syntax Elements

period_expression

Any expression that evaluates to a Period data type.

The Period expression must be comparable with the other expression. Implicit casting to a Period data type is not supported.

Result Type

- Assuming p1 and p2 are comparable Period expressions, p1 LDIFF p2 returns PERIOD(BEGIN(p1), BEGIN(p2)) if p1 OVERLAPS p2 is TRUE and BEGIN(p1) is less than BEGIN(p2). If either Period expression is NULL, p1 OVERLAPS p2 is FALSE, or BEGIN(p1) is not less than BEGIN(p2), the result is NULL.
- If the Period expressions have PERIOD(TIME(n) [WITH TIME ZONE]) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) data types but have different precisions, the result has the higher of the two precisions. If one of the Period expressions contains time zones and the other does not, the result contains a time zone for each element. The result time zones are evaluated using the following rules:
 - If both Period expressions have a time zone, the time zone displacement of a result bound is obtained from the corresponding bound of the expressions as defined by the Period value constructor that follows.
 - If only one of the Period expressions has a time zone, the other Period expression is considered to be at the current session time zone and the result is computed as follows.

Assuming p1 and p2 are Period expressions and the result element type as determined above is rt, the result of p1 LDIFF p2 is as follows if p1 OVERLAPS p2 is TRUE:

```
PERIOD(
  CASE WHEN CAST(BEGIN(p1) AS rt) < CAST(BEGIN(p2) AS rt)
    THEN CAST(BEGIN(p1) AS rt)
    ELSE NULL END,
  CASE WHEN CAST(BEGIN(p1) AS rt) < CAST(BEGIN(p2) AS rt)
    THEN CAST(BEGIN(p2) AS rt)
    ELSE NULL END)
```


Internally, Period values are saved in UTC and the OVERLAPS operator is evaluated using these UTC represented formats and the LDIFF operation is performed if they overlap.

Usage Notes

Format and Title

The format is the default format for the resulting Period data type.

Error Conditions

If either expression is not a Period expression, an error is reported.

If the Period expressions are not comparable, an error is reported.

Example

LDIFF is used to find the left difference of the first Period expression with the second Period expression.

```
SELECT ename, period2 LDIFF period1 FROM employee;
```

Assume the query is executed on the following employee table where period1 and period2 are PERIOD(DATE) columns:

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2002-03-05', '2004-10-07')
Randy	('2006-01-02', '2007-03-05')	('2004-03-07', '2005-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is:

ename	(period2 LDIFF period1)
Adams	?
Mary	('2005-02-03', '2005-04-02')
Jones	('2002-03-05', '2004-01-02')
Randy	?
Simon	?

RDIFF

Returns the portion of the first Period expression that exists from the end of the second Period expression when the Period expressions overlap. When the Period expressions overlap but there is no portion of the first Period expression from the end of the second Period expression or if the Period expressions do not overlap, RDIFF returns NULL. If either Period expression is NULL, RDIFF returns NULL.

RDIFF Syntax

```
period_expression RDIFF period_expression
```

Syntax Elements

period_expression

Any expression that evaluates to a Period data type.

The Period expression must be comparable with the other expression. Implicit casting to a Period data type is not supported.

Result Type

- Assuming p1 and p2 are comparable Period expressions, p1 RDIFF p2 returns PERIOD(END(p2), END(p1)) if p1 OVERLAPS p2 is TRUE and END(p1) is greater than END(p2). If either Period expression is NULL, p1 OVERLAPS p2 is FALSE, or END(p1) is not greater than END(p2), the result is NULL.
- If the Period expressions have PERIOD(TIME[(n)] [WITH TIME ZONE]) or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) data types but have different precisions, the result has the higher of the two precisions. If one of the Period expressions contains time zones and the other does not, the result contains a time zone for each element. The result time zones are evaluated using the following rules:
 - If both Period expressions have a time zone, the time zone displacement of a result bound is obtained from the corresponding bound of the Period expressions as defined by the Period value constructor that follows.
 - If only one of the Period expressions has a time zone, the other Period expression is considered to be at the current session time zone and the result is computed as follows.

Assuming p1 and p2 are Period expressions and the result element type as determined above is rt, the result of p1 RDIFF p2 is as follows if p1 OVERLAPS p2 is TRUE:

```
PERIOD(
  CASE WHEN CAST(END(p1) AS rt) > CAST(END(p2) AS rt)
    THEN CAST(END(p2) AS rt)
    ELSE NULL END,
  CASE WHEN CAST(END(p1) AS rt) > CAST(END(p2) AS rt)
```

```
THEN CAST(END(p1) AS rt)
ELSE NULL END)
```

Internally, Period values are saved in UTC and the OVERLAPS operator is evaluated using these UTC represented formats and the RDIFF operation is performed if they overlap.

Usage Notes

Format and Title

The format is the default format for the resulting Period data type.

Error Conditions

If either expression is not a Period expression, an error is reported.

If the Period expressions are not comparable, an error is reported.

Example

RDIFF is used to find the right difference of the first Period expression with the second Period expression.

Assume the query is executed on the following *employee* table where period1 and period2 are PERIOD(DATE) columns:

```
SELECT ename, period2 RDIFF period1 FROM employee;
ename    period1                                period2
-----
Adams    ('2005-02-03', '2006-02-03') ('2005-02-03', '2006-02-03')
Mary     ('2005-04-02', '2006-01-03') ('2005-02-03', '2006-02-03')
Jones    ('2001-01-02', '2003-03-05') ('2002-03-05', '2004-10-07')
Randy    ('2006-01-02', '2007-03-05') ('2004-03-07', '2005-10-07')
Simon    ?                               ('2005-02-03', '2005-07-27')
```

The result is:

```
ename    (period2 RDIFF period1)
-----
Adams    ?
Mary     ('2006-01-03', '2006-02-03')
Jones    ('2003-03-05', '2004-10-07')
Randy    ?
Simon    ?
```

SUCCEEDS

Evaluates two period expressions, or derived periods, or DateTime expressions to TRUE, FALSE, or UNKNOWN.

SUCCEEDS Syntax

```
{ period_expression [NOT] SUCCEEDS
  { period_expression | datetime_expression | derived_period } |

datetime_expression [NOT] SUCCEEDS period_expression |

derived_period [NOT] SUCCEEDS { derived_period | period_expression }
}
```

Syntax Elements

period_expression

Any expression that evaluates to a Period data type.

The Period expression must be comparable with the other expression. Implicit casting to a Period data type is not supported.

datetime_expression

An expression that evaluates to a DATE, TIME, or TIMESTAMP value.

derived_period

Any expression that evaluates to a Period data type.

Result Type

- If both expressions have a Period data type or a derived period, the function returns TRUE if the beginning bound of the first expression is greater than or equal to the ending bound of the second expression; otherwise, the function returns FALSE.
- If the first expression is a Period expression and the second expression is a DateTime expression, the function returns TRUE if the beginning bound of the first expression is greater than the second expression; otherwise, the function returns FALSE.
- If the first expression is a DateTime expression and the second expression is a Period expression, the function returns TRUE if the DateTime expression is greater than or equal to the ending bound of the second expression; otherwise, the function returns FALSE.
- If either expression is NULL, the operator returns UNKNOWN.

Usage Notes

Error Conditions

If either expression is other than a Period data type or a DateTime value expression, an error is reported.
 If the expressions are not comparable types, an error is reported.

Examples

Example

Assume the query is executed on the following *employee* table where period1 and period2 are PERIOD(DATE) columns:

```
SELECT * FROM employee WHERE period2 SUCCEEDS period1;
```

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is:

ename	period1	period2
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')

Example

Assume in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
  eid INTEGER NOT NULL,
  name VARCHAR(100) NOT NULL,
  deptno INTEGER NOT NULL,
  jobst1 DATE NOT NULL,
  jobend1 DATE NOT NULL,
  PERIOD FOR jobdur1(jobst1, jobend1),
  jobst2 DATE NOT NULL,
  jobend2 DATE NOT NULL,
  PERIOD FOR jobdur2(jobst2, jobend2)
) PRIMARY INDEX(eid);
```

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, SUCCEEDS is used with derived period columns of the employee table:

```
SELECT eid, name, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 SUCCEEDS jobdur2;
```

The result is:

EID	Name	Dept No	JobSt1	JobEnd1	JobSt2	JobEnd2
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'

TD_NORMALIZE_OVERLAP

Combines the rows whose Period values overlap so that the resulting normalized row contains the earliest beginning bound and the latest ending bound from the Period values of all the rows involved.

Result Type

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

TD_NORMALIZE_OVERLAP Syntax

```
[TD_SYSFNLIB.] TD_NORMALIZE_OVERLAP ( grouping_column_list, period_column )
```

Syntax Elements

grouping_column_list

One or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.

period_column

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

TD_SYSFNLIB.

Name of the database where the function is located.

Usage Notes

Input and Output

TD_NORMALIZE_OVERLAP is a table function that takes two arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For details, see the information about NEW VARIANT_TYPE in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- The second argument is the Period column where you want to find the Period values that overlap.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- You must specify the output columns to be the same as the columns specified in the input arguments, including the Period column.
- You must specify the output columns with the same data types and in the same order as the corresponding input columns.
- You can specify an optional INTEGER output column at the end of the RETURNS clause to contain a count of the rows that were normalized.

Format and Title

The format is the default format for the element type of the Period value expression.

The title is (*period_value_expression*).

Error Conditions

The table function returns an error in the following cases:

- The function invocation does not include a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column.
- The function invocation does not include a HASH BY clause with at least one of the grouping columns.
- The HASH BY clause includes the Period column or other columns that are not part of the grouping columns.
- The function was invoked without a RETURNS clause or the columns specified in the RETURNS clause do not match the input columns or are not in the same order as the input columns.

Example: Using TD_NORMALIZE_OVERLAP

```
WITH subtbl(flight_id, duration) AS
  (SELECT flight_id, duration FROM FlightExp)
SELECT *
FROM TABLE (TD_SYSNLIB.TD_NORMALIZE_OVERLAP(NEW VARIANT_TYPE(subtbl.flight_id),
                                              subtbl.duration)
  RETURNS (flight_id INT, duration PERIOD(TIMESTAMP(6) WITH TIME ZONE),
    NrmCount INT)
  HASH BY flight_id      /* input data is redistributed on column, flight_id */
```



```
LOCAL ORDER BY flight_id, duration)      /* input data is sorted on these
columns */
AS DT(flight_id, duration, NrmCount) ORDER BY 1,2;
```

TD_NORMALIZE_MEET

Combines the rows whose Period values meet so that the resulting normalized row contains the earliest beginning bound and the latest ending bound from the Period values of all the rows involved.

Result Type

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

TD_NORMALIZE_MEET Syntax

```
[TD_SYSFNLIB.] TD_NORMALIZE_MEET ( grouping_column_list, period_column )
```

Syntax Elements

grouping_column_list

One or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.

period_column

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

TD_SYSFNLIB.

Name of the database where the function is located.

Usage Notes

Input and Output

TD_NORMALIZE_MEET is a table function that takes two arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For details, see the information about NEW VARIANT_TYPE in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- The second argument is the Period column where you want to find the Period values that meet.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- You must specify the output columns to be the same as the columns specified in the input arguments, including the Period column.
- You must specify the output columns with the same data types and in the same order as the corresponding input columns.
- You can specify an optional INTEGER output column at the end of the RETURNS clause to contain a count of the rows that were normalized.

Format and Title

The format is the default format for the element type of the Period value expression.

The title is BEGIN(*period_value_expression*).

Error Conditions

The table function returns an error in the following cases:

- The function invocation does not include a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column.
- The function invocation does not include a HASH BY clause with at least one of the grouping columns.
- The HASH BY clause includes the Period column or other columns that are not part of the grouping columns.
- The function was invoked without a RETURNS clause or the columns specified in the RETURNS clause do not match the input columns or are not in the same order as the input columns.

Example: Using TD_NORMALIZE_MEET

```
WITH subtbl(flight_id, duration) AS
  (SELECT flight_id, duration FROM FlightExp)
SELECT *
FROM TABLE (TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(subtbl.flight_id),
                                             subtbl.duration)
RETURNS (flight_id INT, duration PERIOD(TIMESTAMP(6) WITH TIME ZONE),
NrmCount INT)
HASH BY flight_id      /* input data is redistributed on column, flight_id */
LOCAL ORDER BY flight_id, duration) /* input data is sorted on these
columns */
AS DT(flight_id, duration, NrmCount) ORDER BY 1,2;
```

TD_NORMALIZE_OVERLAP_MEET

Combines the rows whose Period values either meet or overlap so that the resulting normalized row contains the earliest beginning bound and the latest ending bound from the Period values of all the rows involved.

Result Type

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

TD_NORMALIZE_OVERLAP_MEET Syntax

```
[TD_SYSFNLIB.] TD_NORMALIZE_OVERLAP_MEET ( grouping_column_list, period_column )
```

Syntax Elements

grouping_column_list

One or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.

period_column

Any expression that evaluates to a Period data type.

The Period expression must be comparable with the other expression. Implicit casting to a Period data type is not supported.

TD_SYSFNLIB.

Name of the database where the function is located.

Usage Notes

Input and Output

TD_NORMALIZE_OVERLAP_MEET is a table function that takes two arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For details, see the information about NEW VARIANT_TYPE in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- The second argument is the Period column where you want to find the Period values that overlap or meet.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- You must specify the output columns to be the same as the columns specified in the input arguments, including the Period column.
- You must specify the output columns with the same data types and in the same order as the corresponding input columns.
- You can specify an optional INTEGER output column at the end of the RETURNS clause to contain a count of the rows that were normalized.

Format and Title

The format is the default format for the element type of the Period value expression.

The title is BEGIN(*period_value_expression*).

Error Conditions

The table function returns an error in the following cases:

- The function invocation does not include a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column.
- The function invocation does not include a HASH BY clause with at least one of the grouping columns.
- The HASH BY clause includes the Period column or other columns that are not part of the grouping columns.
- The function was invoked without a RETURNS clause or the columns specified in the RETURNS clause do not match the input columns or are not in the same order as the input columns.

Example: Using TD_NORMALIZE_OVERLAP_MEET

```
WITH subtbl(flight_id, duration) AS
  (SELECT flight_id, duration FROM FlightExp)
SELECT *
FROM TABLE
(TD_SYSFNLIB.TD_NORMALIZE_OVERLAP_MEET(NEW VARIANT_TYPE(subtbl.flight_id),
  RETURNS (flight_id INT, duration PERIOD(TIMESTAMP(6) WITH TIME ZONE), NrmCount
  INT)
  subtbl.duration)
HASH BY flight_id      /* input data is redistributed on column, flight_id */
LOCAL ORDER BY flight_id, duration) /* input data is sorted on these
columns */
AS DT(flight_id, duration, NrmCount) ORDER BY 1,2;
```

TD_SUM_NORMALIZE_OVERLAP

Finds the sum of a column for all the rows that were normalized because their Period values overlap.

Result Type

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values overlap.

TD_SUM_NORMALIZE_OVERLAP Syntax

```
[TD_SYSFNLIB.] TD_SUM_NORMALIZE_OVERLAP (
    grouping_column_list,
    numeric_column,
    period_column
)
```

Syntax Elements

grouping_column_list

One or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.

numeric_column

A numeric column on which SUM() is requested. You must specify the input as a dynamic UDT.

period_column

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

A column with a data type of PERIOD(DATE), PERIOD(TIMESTAMP), or PERIOD(TIMESTAMP WITH TIME ZONE).

TD_SYSFNLIB.

Name of the database where the function is located.

Usage Notes

Input and Output

TD_SUM_NORMALIZE_OVERLAP is a table function that takes three arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For details, see the information about NEW VARIANT_TYPE in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

- The second argument is a numeric column on which SUM() is requested. All numeric data types are supported. You must specify this argument as a dynamic UDT where the column is an attribute of the UDT.
- The third argument is the Period column where you want to find the Period values that overlap.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- You must specify the output columns to be the same as the columns specified in the input arguments, including the Period column.
- You must specify the output columns with the same data types and in the same order as the corresponding input columns.
- You must include a numeric output column to contain the sum result value. The data type of this column should be the same data type as the corresponding input column. To prevent a possible overflow error, you can use the CAST function to convert the data type of the input column to a larger numeric data type.

Format and Title

The format is the default format for the element type of the Period value expression.

The title is BEGIN(*period_value_expression*).

Error Conditions

The table function returns an error in the following cases:

- The function invocation does not include a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column.
- The function invocation does not include a HASH BY clause with at least one of the grouping columns.
- The HASH BY clause includes the Period column or other columns that are not part of the grouping columns.
- The function was invoked without a RETURNS clause or the columns specified in the RETURNS clause do not match the input columns or are not in the same order as the input columns.

Example: Using TD_SUM_NORMALIZE_OVERLAP

```
WITH subtbl(flight_id, charges, duration) AS
  (SELECT flight_id, charges, duration FROM FlightExp)
SELECT *
  FROM TABLE
    (TD_SYSFNLIB.TD_SUM_NORMALIZE_OVERLAP(NEW VARIANT_TYPE(subtbl.flight_id),
      NEW VARIANT_TYPE(subtbl.charges),
      subtbl.duration)
  RETURNS (flight_id INT, cnt INT, charges FLOAT,
    duration PERIOD(TIMESTAMP(6) WITH TIME ZONE))
  HASH BY flight_id /* input data is redistributed on column, flight_id */
  LOCAL ORDER BY flight_id, duration) /* input data is sorted on these columns */
AS DT(flight_id, charges, duration) ORDER BY 1,3;
```

TD_SUM_NORMALIZE_MEET

Finds the sum of a column for all the rows that were normalized because their Period values meet.

Result Type

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

TD_SUM_NORMALIZE_MEET Syntax

```
[TD_SYSFNLIB.] TD_SUM_NORMALIZE_MEET (
  grouping_column_list,
  numeric_column,
  period_column
)
```

Syntax Elements

grouping_column_list

One or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.

numeric_column

A numeric column on which SUM() is requested. You must specify the input as a dynamic UDT.

period_column

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

TD_SYSFNLIB.

Name of the database where the function is located.

Usage Notes

Input and Output

TD_SUM_NORMALIZE_MEET is a table function that takes three arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For details, see the information about NEW VARIANT_TYPE in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- The second argument is a numeric column on which SUM() is requested. All numeric data types are supported. You must specify this argument as a dynamic UDT where the column is an attribute of the UDT.
- The third argument is the Period column where you want to find the Period values that meet.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- You must specify the output columns to be the same as the columns specified in the input arguments, including the Period column.
- You must specify the output columns with the same data types and in the same order as the corresponding input columns.

- You must include a numeric output column to contain the sum result value. The data type of this column should be the same data type as the corresponding input column. To prevent a possible overflow error, you can use the CAST function to convert the data type of the input column to a larger numeric data type.

Format and Title

The format is the default format for the element type of the Period value expression.

The title is BEGIN(*period_value_expression*).

Error Conditions

The table function returns an error in the following cases:

- The function invocation does not include a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column.
- The function invocation does not include a HASH BY clause with at least one of the grouping columns.
- The HASH BY clause includes the Period column or other columns that are not part of the grouping columns.
- The function was invoked without a RETURNS clause or the columns specified in the RETURNS clause do not match the input columns or are not in the same order as the input columns.

Example: Using TD_SUM_NORMALIZE_MEET

```
WITH subtbl(flight_id, charges, duration) AS
  (SELECT flight_id, charges, duration FROM FlightExp)
SELECT *
FROM TABLE
  (TD_SYSFNLIB.TD_SUM_NORMALIZE_MEET(NEW VARIANT_TYPE(subtbl.flight_id),
                                     NEW VARIANT_TYPE(subtbl.charges),
                                     subtbl.duration)
  RETURNS (flight_id INT, charges FLOAT,
           duration PERIOD(TIMESTAMP(6) WITH TIME ZONE))
  HASH BY flight_id      /* input data is redistributed on column, flight_id */
  LOCAL ORDER BY flight_id, duration) /* input data is sorted on these
columns */
AS DT(flight_id, charges, duration) ORDER BY 1,3;
```

TD_SUM_NORMALIZE_OVERLAP_MEET

Finds the sum of a column for all the rows that were normalized because their Period values either overlap or meet.

Result Type

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values overlap or meet.

TD_SUM_NORMALIZE_OVERLAP_MEET Syntax

```
[TD_SYSFNLIB.] TD_SUM_NORMALIZE_OVERLAP_MEET (
    grouping_column_list,
    numeric_column,
    period_column
)
```

Syntax Elements

grouping_column_list

One or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.

numeric_column

A numeric column on which SUM() is requested. You must specify the input as a dynamic UDT.

period_column

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

TD_SYSFNLIB.

Name of the database where the function is located.

Usage Notes

TD_SUM_NORMALIZE_OVERLAP_MEET is a table function that takes three arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For details, see the about NEW VARIANT_TYPE in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- The second argument is a numeric column on which SUM() is requested. All numeric data types are supported. You must specify this argument as a dynamic UDT where the column is an attribute of the UDT.
- The third argument is the Period column where you want to find the Period values that overlap or meet.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- You must specify the output columns to be the same as the columns specified in the input arguments, including the Period column.
- You must specify the output columns with the same data types and in the same order as the corresponding input columns.
- You must include a numeric output column to contain the sum result value. The data type of this column should be the same data type as the corresponding input column. To prevent a possible overflow error, you can use the CAST function to convert the data type of the input column to a larger numeric data type.

Example: Using TD_SUM_NORMALIZE_OVERLAP_MEET

```
WITH subtbl(flight_id, charges, duration) AS
  (SELECT flight_id, charges, duration FROM FlightExp)
SELECT * FROM TABLE (

  TD_SYSFNLIB.TD_SUM_NORMALIZE_OVERLAP_MEET(NEW VARIANT_TYPE(subtbl.flight_id),
                                             NEW VARIANT_TYPE(subtbl.charges),
                                             subtbl.duration)

  RETURNS (flight_id INT, charges FLOAT,
           duration PERIOD(TIMESTAMP(6) WITH TIME ZONE))
  HASH BY flight_id      /* input data is redistributed on column, flight_id */)
```

```
LOCAL ORDER BY flight_id, duration)      /* input data is sorted on these
columns */
AS DT(flight_id, charges, duration) ORDER BY 1,3;
```

TD_SEQUENCED_SUM

Finds the sum of a column for all adjacent periods in normalized rows whose Period values either meet or overlap.

Result Type

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

TD_SEQUENCED_SUM Syntax

```
[TD_SYSFNLIB.] TD_SEQUENCED_SUM (
  grouping_column_list,
  numeric_column,
  period_column
)
```

Syntax Elements

grouping_column_list

One or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.

numeric_column

A numeric column on which SUM() is requested. You must specify the input as a dynamic UDT.

period_column

Any expression that evaluates to a Period data type.

The Period expression must be comparable with the other expression. Implicit casting to a Period data type is not supported.

TD_SYSFNLIB.

Name of the database where the function is located.

Usage Notes

Input and Output

TD_SEQUENCED_SUM is a table function that takes three arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For details, see the information about NEW VARIANT_TYPE in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- The second argument is a numeric column on which SUM() is requested. All numeric data types are supported. You must specify this argument as a dynamic UDT where the column is an attribute of the UDT.
- The third argument is the Period column where you want to find the Period values that overlap or meet.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- The output columns must include all of the grouping columns with the same data type and in the same order as the input columns.
- You must include a numeric output column to contain the sum result value. The data type of this column should be the same data type as the corresponding input column. To prevent a possible overflow error, you can use the CAST function to convert the data type of the input column to a larger numeric data type.
- A Period column with the same Period data type as the input Period column.

Format and Title

The format is the default format for the element type of the Period value expression.

The title is BEGIN(*period_value_expression*).

Error Conditions

The table function returns an error in the following cases:

- The function invocation does not include a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column.
- The function invocation does not include a HASH BY clause with at least one of the grouping columns.
- The HASH BY clause includes the Period column or other columns that are not part of the grouping columns.
- The function was invoked without a RETURNS clause or the columns specified in the RETURNS clause do not match the input columns or are not in the same order as the input columns.

Example: Using TD_SEQUENCED_SUM

```
WITH subtbl(flight_id, charges, duration) AS
  (SELECT flight_id, charges, duration FROM FlightExp)
SELECT * FROM TABLE (
  TD_SYSFNLIB.TD_SEQUENCED_SUM(NEW VARIANT_TYPE(subtbl.flight_id),
                                NEW VARIANT_TYPE(subtbl.charges),
                                subtbl.duration)

  RETURNS (flight_id INT, charges FLOAT,
            duration PERIOD(TIMESTAMP(6) WITH TIME ZONE))
  HASH BY flight_id      /* input data is redistributed on column, flight_id */
  LOCAL ORDER BY flight_id, duration) /* input data is sorted on these
  columns */
AS DT(flight_id, charges, duration) ORDER BY 1,3;
```

TD_SEQUENCED_AVG

Finds the average of a column for all adjacent periods in normalized rows whose Period values either meet or overlap.

Result Type

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

TD_SEQUENCED_AVG Syntax

```
[TD_SYSFNLIB.] TD_SEQUENCED_AVG (
    grouping_column_list,
    numeric_column,
    period_column
)
```

Syntax Elements

grouping_column_list

One or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.

numeric_column

A numeric column on which the average is requested. You must specify the input as a dynamic UDT.

period_column

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

TD_SYSFNLIB.

Name of the database where the function is located.

Usage Notes

Input and Output

TD_SEQUENCED_AVG is a table function that takes three arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For details, see the information about NEW VARIANT_TYPE in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- The second argument is a numeric column on which AVG() is requested. All numeric data types are supported. You must specify this argument as a dynamic UDT where the column is an attribute of the UDT.

- The third argument is the Period column where you want to find the Period values that overlap or meet.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- The output columns must include all of the grouping columns with the same data type and in the same order as the input columns.
- You must include a numeric output column to contain the average result value. The data type of this column can be FLOAT or the same data type as the corresponding input column; however, to avoid possible rounding of the result value, it is recommended that you use FLOAT. To prevent a possible overflow error, you can use the CAST function to convert the data type of the input column to a larger numeric data type.
- A Period column with the same Period data type as the input Period column.

Format and Title

The format is the default format for the element type of the Period value expression.

The title is BEGIN(*period_value_expression*).

Error Conditions

The table function returns an error in the following cases:

- The function invocation does not include a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column.
- The function invocation does not include a HASH BY clause with at least one of the grouping columns.
- The HASH BY clause includes the Period column or other columns that are not part of the grouping columns.
- The function was invoked without a RETURNS clause or the columns specified in the RETURNS clause do not match the input columns or are not in the same order as the input columns.

Example: Using TD_SEQUENCED_AVG

```
WITH subtbl(flight_id, charges, duration) AS
  (SELECT flight_id, charges, duration FROM FlightExp)
SELECT * FROM TABLE (
  TD_SYSFNLIB.TD_SEQUENCED_AVG(NEW VARIANT_TYPE(subtbl.flight_id),
    NEW VARIANT_TYPE(subtbl.charges),
    subtbl.duration)
RETURNS (flight_id INT, charges FLOAT,
  duration PERIOD(TIMESTAMP(6) WITH TIME ZONE))
HASH BY flight_id /* input data is redistributed on column, flight_id */
LOCAL ORDER BY flight_id, duration) /* input data is sorted on these columns */
AS DT(flight_id, charges, duration) ORDER BY 1,3;
```

TD_SEQUENCED_COUNT

Finds the count of a column for all adjacent periods in normalized rows whose Period values either meet or overlap.

Result Type

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

TD_SEQUENCED_COUNT Syntax

```
[TD_SYSFNLIB.] TD_SEQUENCED_COUNT (
  grouping_column_list,
  numeric_column,
  period_column
)
```

Syntax Elements

grouping_column_list

One or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.

period_column

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

TD_SYSFNLIB.

Name of the database where the function is located.

Usage Notes

Input and Output

TD_SEQUENCED_COUNT is a table function that takes two arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For details, see the information about NEW VARIANT_TYPE in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- The second argument is the Period column where you want to find the Period values that overlap or meet.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- The output columns must include all of the grouping columns with the same data type and in the same order as the input columns.
- You must include an INTEGER output column to contain the count result.
- A Period column with the same Period data type as the input Period column.

Format and Title

The format is the default format for the element type of the Period value expression.

The title is (*period_value_expression*).

Error Conditions

The table function returns an error in the following cases:

- The function invocation does not include a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column.
- The function invocation does not include a HASH BY clause with at least one of the grouping columns.
- The HASH BY clause includes the Period column or other columns that are not part of the grouping columns.
- The function was invoked without a RETURNS clause or the columns specified in the RETURNS clause do not match the input columns or are not in the same order as the input columns.

Example: Using TD_SEQUENCED_COUNT

```
WITH subtbl(flight_id, duration) AS
  (SELECT flight_id, duration FROM FlightExp)
SELECT * FROM TABLE (
  TD_SYSFNLIB.TD_SEQUENCED_COUNT(NEW VARIANT_TYPE(subtbl.flight_id),
                                   subtbl.duration)
  RETURNS (flight_id INT, cnt INT,
           duration PERIOD(TIMESTAMP(6) WITH TIME ZONE))
  HASH BY flight_id /* input data is redistributed on column, flight_id */
  LOCAL ORDER BY flight_id, duration) /* input
```

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 ss3.
!	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by ss2, forming an individual multibyte character.
△	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