

# Teradata Vantage™ - Data Types and Literals

---

Release 17.10

July 2021

# Copyright and Trademarks

Copyright © 2000 - 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](mailto: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

<b>Chapter 1: Introduction to SQL Data Types and Literals</b>	<b>9</b>
Changes and Additions	9
<b>Chapter 2: SQL Data Definition</b>	<b>10</b>
Data Definition in SQL	10
Data Definition Phrases	11
Data Conversion	18
COMPRESS and DECOMPRESS Phrases	20
Constraint Attributes	29
Uniqueness Constraints	29
CHECK Constraints	30
Referential Constraints	31
<b>Chapter 3: Data Literals</b>	<b>32</b>
Hexadecimal Byte Literals	32
Numeric Literals	33
Date and Time Literals	42
Interval Literals	49
Period Literals	64
Character String Literals	69
Unicode Delimited Character Literals	75
Hexadecimal Character Literals	79
Graphic Literals	80
<b>Chapter 4: Numeric Data Types</b>	<b>83</b>
BYTEINT Data Type	83
SMALLINT Data Type	84
INTEGER Data Type	85
BIGINT Data Type	86
DECIMAL/NUMERIC Data Types	87
FLOAT/REAL/DOUBLE PRECISION Data Types	91
NUMBER Data Type	93
Operations on Floating Point Values	98
Rounding	100
<b>Chapter 5: Character and CLOB Data Types</b>	<b>103</b>
Character Data	103
CHARACTER Data Type	104
VARCHAR Data Type	108

CLOB Data Type . . . . .	112
Default Case Specificity of Character Columns . . . . .	116
CASESPECIFIC Phrase . . . . .	117
UPPERCASE Phrase . . . . .	120
Teradata SQL Character Strings and Client Physical Bytes . . . . .	122
CHARACTER SET Phrase . . . . .	124
LATIN Server Character Set . . . . .	126
UNICODE Server Character Set . . . . .	126
GRAPHIC Server Character Set . . . . .	127
KANJISJIS Server Character Set . . . . .	128
KANJI1 Server Character Set [Deprecated] . . . . .	129
<b>Chapter 6: Byte and BLOB Data Types . . . . .</b>	<b>133</b>
Data Storage of Byte and BLOB Types . . . . .	133
BYTE Data Type . . . . .	133
VARBYTE Data Type . . . . .	134
BLOB Data Type . . . . .	135
<b>Chapter 7: LOB Functions . . . . .</b>	<b>139</b>
EMPTY_BLOB . . . . .	139
EMPTY_CLOB . . . . .	139
<b>Chapter 8: DateTime and Interval Data Types . . . . .</b>	<b>141</b>
DateTime Fields . . . . .	141
Time Zones . . . . .	141
Daylight Saving Time . . . . .	144
DATE Data Type . . . . .	145
TIME Data Type . . . . .	153
TIMESTAMP Data Type . . . . .	155
TIME WITH TIME ZONE Data Type . . . . .	158
TIMESTAMP WITH TIME ZONE Data Type . . . . .	160
INTERVAL YEAR Data Type . . . . .	163
INTERVAL YEAR TO MONTH Data Type . . . . .	164
INTERVAL MONTH Data Type . . . . .	166
INTERVAL DAY Data Type . . . . .	167
INTERVAL DAY TO HOUR Data Type . . . . .	169
INTERVAL DAY TO MINUTE Data Type . . . . .	171
INTERVAL DAY TO SECOND Data Type . . . . .	173
INTERVAL HOUR Data Type . . . . .	176
INTERVAL HOUR TO MINUTE Data Type . . . . .	177
INTERVAL HOUR TO SECOND Data Type . . . . .	179
INTERVAL MINUTE Data Type . . . . .	182
INTERVAL MINUTE TO SECOND Data Type . . . . .	183
INTERVAL SECOND Data Type . . . . .	186

<b>Chapter 9: Period Data Types</b>	<b>190</b>
Period Data Types: Basic Definitions	190
Period Type Input and Output Parameters	191
Related Information	192
PERIOD(DATE) Data Type	192
PERIOD(TIME) Data Type	195
PERIOD(TIME WITH TIME ZONE) Data Type	198
PERIOD(TIMESTAMP) Data Type	201
PERIOD(TIMESTAMP WITH TIME ZONE) Data Type	204
<b>Chapter 10: ARRAY/VARRAY Data Type</b>	<b>208</b>
ANSI Compliance	208
Syntax	209
One-Dimensional (1-D) ARRAY Data Type	209
Multidimensional (n-D) ARRAY Data Type	210
Privileges Required for Creating an ARRAY Data Type	211
Creating an ARRAY Data Type	211
Supported Data Types for ARRAY Elements	211
Autogenerated Functionality for an ARRAY Data Type	212
Rules and Guidelines	213
ARRAY Functions, Operators, and Expressions	213
ARRAY Type Input and Output Parameters	213
Restrictions	214
Example: Creating an ARRAY Data Type	214
Example: Creating a Table with an ARRAY Column	215
Example: ARRAY Parameter in a Java UDF	215
Example: ARRAY Parameter in a Java External Stored Procedure	216
Related Information	216
<b>Chapter 11: ARRAY/VARRAY Functions and Operators</b>	<b>217</b>
ARRAY/VARRAY Functions and Operators	217
ARRAY Element Reference	217
ARRAY_Constructor_Expression	220
ARRAY Scope Reference	223
ARRAY_AGG	224
UNNEST	228
CARDINALITY	231
ARRAY_CONCATENATION_OPERATOR	233
ARRAY_CONCATENATION_FUNCTION	235
ARRAY_COMPARISON_FUNCTION	237
ARRAY_ARITHMETIC_FUNCTION	242
ARRAY_SUM	246
ARRAY_AVG	249
ARRAY_MAX	251

ARRAY_MIN .....	253
ARRAY_COUNT_DISTINCT .....	256
ARRAY_GET .....	258
ARRAY_COMPARE .....	260
ARRAY_UPDATE .....	263
ARRAY_UPDATE_STRIDE .....	266
OEXISTS .....	269
OCOUNT .....	273
OLIMIT .....	276
OFIRST .....	279
OLAST .....	281
OPRIOR .....	284
ONEXT .....	287
OEXTEND .....	291
OTRIM .....	296
ODELETE .....	300
<b>Chapter 12: UDT Data Type .....</b>	<b>304</b>
UDT Data Type Syntax .....	304
Usage Notes .....	305
UDT Data Type Examples .....	308
Related Information .....	310
<b>Chapter 13: Parameter Data Types .....</b>	<b>312</b>
TD_ANYTYPE Data Type .....	312
VARIANT_TYPE Data Type .....	315
<b>Chapter 14: Data Type Formats and Format Phrases .....</b>	<b>318</b>
Data Type Default Formats .....	318
DATE Formats .....	321
TIME and TIMESTAMP Formats .....	324
FORMAT .....	328
FORMAT Phrase and Character Formats .....	332
FORMAT Phrase and NUMERIC Formats .....	335
FORMAT Phrase and DateTime Formats .....	351
FORMAT Phrase, DateTime Formats, and Japanese Character Sets .....	358
Naming Columns and Expressions .....	361
AS .....	364
NAMED .....	365
TITLE .....	366
<b>Chapter 15: Data Type Conversions .....</b>	<b>372</b>
Data Type Conversions .....	372
Forms of Data Type Conversions .....	372
Implicit Type Conversions .....	372

CAST in Explicit Data Type Conversions . . . . .	379
Data Conversions in Field Mode . . . . .	382
Byte-to-Byte Conversion . . . . .	383
Character-to-Character Conversion . . . . .	387
Implicit Character-to-Character Translation . . . . .	391
Character-to-DATE Conversion . . . . .	392
Character-to-INTERVAL Conversion . . . . .	398
Character-to-Numeric Conversion . . . . .	400
Character-to-Period Conversion . . . . .	406
Character-to-TIME Conversion . . . . .	409
Character-to-TIMESTAMP Conversion . . . . .	416
Character-to-UDT Conversion . . . . .	421
Character Data Type Assignment Rules . . . . .	423
DATE-to-Character Conversion . . . . .	424
DATE-to-DATE Conversion . . . . .	428
DATE-to-Numeric Conversion . . . . .	430
DATE-to-Period Conversion . . . . .	433
DATE-to-TIMESTAMP Conversion . . . . .	435
DATE-to-UDT Conversion . . . . .	441
INTERVAL-to-Character Conversion . . . . .	443
INTERVAL-to-INTERVAL Conversion . . . . .	445
INTERVAL-to-Numeric Conversion . . . . .	449
INTERVAL-to-UDT Conversion . . . . .	451
Numeric-to-Character Conversion . . . . .	453
Numeric-to-DATE Conversion . . . . .	458
Numeric-to-INTERVAL Conversion . . . . .	460
Numeric-to-Numeric Conversion . . . . .	462
Numeric-to-UDT Conversion . . . . .	467
Period-to-Character Conversion . . . . .	469
Period-to-DATE Conversion . . . . .	472
Period-to-Period Conversion . . . . .	473
Period-to-TIME Conversion . . . . .	477
Period-to-TIMESTAMP Conversion . . . . .	479
Signed Zone DECIMAL Conversion . . . . .	481
TIME-to-Character Conversion . . . . .	485
TIME-to-Period Conversion . . . . .	488
TIME-to-TIME Conversion . . . . .	490
TIME-to-TIMESTAMP Conversion . . . . .	497
TIME-to-UDT Conversion . . . . .	511
TIMESTAMP-to-Character Conversion . . . . .	512
TIMESTAMP-to-DATE Conversion . . . . .	516
TIMESTAMP-to-Period Conversion . . . . .	527
TIMESTAMP-to-TIME Conversion . . . . .	529
TIMESTAMP-to-TIMESTAMP Conversion . . . . .	537
TIMESTAMP-to-UDT Conversion . . . . .	546

TRYCAST .....	547
UDT-to-Byte Conversion .....	549
UDT-to-Character Conversion .....	551
UDT-to-DATE Conversion .....	554
UDT-to-INTERVAL Conversion .....	557
UDT-to-Numeric Conversion .....	559
UDT-to-TIME Conversion .....	562
UDT-to-TIMESTAMP Conversion .....	565
UDT-to-UDT Conversion .....	568
<b>Chapter 16: Data Type Conversion Functions .....</b>	<b>570</b>
TO_BYTES .....	570
FROM_BYTES .....	572
TO_NUMBER .....	574
TO_CHAR(Numeric) .....	578
TO_CHAR(DateTime) .....	581
TO_DATE .....	582
TO_TIMESTAMP .....	587
TO_TIMESTAMP_TZ .....	589
TO_YMINTERVAL .....	591
TO_DSINTERVAL .....	593
NUMTODSINTERVAL .....	596
NUMTOYMINTERVAL .....	598
<b>Chapter 17: Default Value Control Phrases .....</b>	<b>601</b>
Using Default Value Control Phrases .....	601
NOT NULL Phrase .....	601
DEFAULT Phrase .....	602
WITH DEFAULT Phrase .....	608
<b>Appendix A: Notation Conventions .....</b>	<b>612</b>
<b>Appendix B: External Representations for UDTs .....</b>	<b>615</b>
<b>Appendix C: Additional Information .....</b>	<b>630</b>



# Introduction to SQL Data Types and Literals

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.

Advanced SQL Engine is a core capability of Teradata Vantage, based on our best-in-class Teradata Database. Advanced SQL refers to the ability to run advanced analytic functions beyond that of standard SQL.

For information on data type mapping between Advanced SQL Engine and ML Engine, see *Teradata Vantage™ User Guide*, B700-4002.

*Teradata Vantage™ - Data Types and Literals* describes how to use data types and literals with SQL Engine.

Use this document in conjunction with the other documents in the SQL document set:

- *Teradata Vantage™ - Geospatial Data Types*, B035-1181
- *Teradata Vantage™ - XML Data Type*, B035-1140
- *Teradata Vantage™ - JSON Data Type*, B035-1150
- *Teradata Vantage™ - DATASET Data Type*, B035-1198

## Changes and Additions

Date	Description
July 2021	<ul style="list-style-type: none"><li>• Digits in fractional seconds are now truncated in the timestamp value depending on the specified result format string. See <a href="#">TO_CHAR(DateTime)</a>.</li><li>• Kanji date markers now include the Reiwa Imperial Era. See <a href="#">Kanji Date Markers</a>.</li></ul>

# SQL Data Definition

This section describes some of the general principles of SQL data definition.

## Data Definition in SQL

Data definition phrases, also referred to as data description phrases, are used in SQL statements to define how to store data in the columns of a table, how to present the data in the results of queries, and whether to apply column-level integrity constraints.

You can also use data definition phrases in expressions to convert data to another type or modify data attributes.

## Syntax

```
column_name data_type_declaration  
[ data_type_attribute |  
  column_storage_attribute |  
  column_constraint_attribute  
] [...]
```

### Syntax Elements

#### ***data\_type\_declaration***

The data type of a column, such as BYTE or FLOAT.

For more information, see [Data Type Phrases](#).

#### ***data\_type\_attribute***

Attributes for a column, such as a default value to insert when an INSERT statement omits the value.

For more information, see [Core Data Type Attributes](#).

#### ***column\_storage\_attribute***

Compress certain values and nulls for one or more columns of a table.

For more information, see:

- [Storage and Constraint Attributes](#)
- [COMPRESS and DECOMPRESS Phrases](#)

**column\_constraint\_attribute**

Column-level integrity constraints, such as PRIMARY KEY.

For more information, see:

- [Storage and Constraint Attributes](#)
- [Column and Table Constraints](#)
- ALTER TABLE and CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144

## Using Data Definition

The following CREATE TABLE statement uses Teradata SQL data definition phrases to define the columns of the Employee table:

```
CREATE TABLE Employee
(EmpNo PRIMARY KEY SMALLINT FORMAT '9(5)'
  CHECK(EmpNo BETWEEN 1000 AND 32001),
Name VARCHAR(12) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
DeptNo SMALLINT FORMAT '999'
  CHECK (DeptNo BETWEEN 100 AND 900),
JobTitle VARCHAR(12) CHARACTER SET LATIN NOT CASESPECIFIC,
Salary DECIMAL(8,2) FORMAT 'ZZZ,ZZ9.99'
  CHECK (Salary BETWEEN 1.00 AND 999000.00),
YrsExp BYTEINT FORMAT 'Z9'
  CHECK (YrsExp BETWEEN -99 AND 99),
DOB DATE FORMAT 'MMbDDbYYYY' NOT NULL,
Sex CHAR CHARACTER SET LATIN UPPERCASE NOT NULL,
Race CHAR CHARACTER SET LATIN UPPERCASE,
MStat CHAR CHARACTER SET LATIN UPPERCASE,
EdLev BYTEINT FORMAT 'Z9'
  CHECK (EdLev BETWEEN 0 AND 2) NOT NULL,
HCap BYTEINT FORMAT 'Z9'
  CHECK (HCap BETWEEN -99 AND 99)
INDEX (Name) ;
```

The following SELECT statement uses Teradata SQL data definition phrases to modify the heading of the EmpNo column and the format of the results of the Salary column:

```
SELECT EmpNo (TITLE 'Employee Number'),
  Salary (FORMAT 'GLLZ(I)D9(F)')
FROM Employee;
```

## Data Definition Phrases

### Data Type Phrases

A data type phrase (*data\_type\_declaration*) determines the type of data to store in a column of a table on SQL Engine. When you create a table, you must specify a data type phrase for each column. A column does not have a default data type.

You can also use data type phrases to specify data conversions, casts in expressions, parameter types, and so forth.

The following table shows typical data type phrases you can use to define various data types.

Data Type	ANSI SQL	Teradata Extension to ANSI SQL
<b>Array</b>		
ARRAY/VARRAY		X
<b>Byte</b>		
BLOB[(n)]	X	
BYTE[(n)]		X
VARBYTE[(n)]		X
<b>Numeric</b>		
BIGINT	X	
BYTEINT		X
DATE <sup>a</sup>		X
DECIMAL [(n[,m])]	X	
DOUBLE PRECISION	X	
FLOAT	X	
INTEGER	X	
NUMBER(n[,m])		X
NUMBER[( *[,m])]		X
NUMERIC [(n[,m])]	X	
REAL	X	
SMALLINT	X	
<b>DateTime</b>		
DATE	X	

Data Type	ANSI SQL	Teradata Extension to ANSI SQL
TIME [(n)]	X	
TIMESTAMP [(n)]	X	
<b>Interval</b>		
INTERVAL	X	
INTERVAL DAY [(n)]	X	
INTERVAL DAY [(n)] TO HOUR	X	
INTERVAL DAY [(n)] TO MINUTE	X	
INTERVAL DAY [(n)] TO SECOND	X	
INTERVAL HOUR [(n)]	X	
INTERVAL HOUR [(n)] TO MINUTE	X	
INTERVAL HOUR [(n)] TO SECOND	X	
INTERVAL MINUTE [(n)]	X	
INTERVAL MINUTE [(n)] TO SECOND [(m)]	X	
INTERVAL MONTH	X	
INTERVAL SECOND [(n],[m])	X	
INTERVAL YEAR [(n)]	X	
INTERVAL YEAR [(n)] TO MONTH	X	
<b>Character</b>		
CHAR[(n)]	X	
CHARACTER(n) CHARACTER SET GRAPHIC		X
CLOB	X	
CHAR VARYING(n)	X	
LONG VARCHAR		X
LONG VARCHAR CHARACTER SET GRAPHIC		X
VARCHAR(n)	X	
VARCHAR(n) CHARACTER SET GRAPHIC		X
<b>Period</b>		
PERIOD(DATE)		X
PERIOD(TIME [(n)])		X

Data Type	ANSI SQL	Teradata Extension to ANSI SQL
PERIOD(TIMESTAMP [(n)])		X
<b>UDT</b>		
<i>udt_name</i>	X	
<b>Parameter Types</b>		
TD_ANYTYPE		X
VARIANT_TYPE		X

**Note:**

- DATE is supported both in its Teradata form and in the preferred ANSI DateTime form. For new development, define DATE using ANSI DATE type.
- The CREATE TYPE statement determines the name of a UDT.

For details on the different level of ANSI compliance, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141

## Data Type Classification

Data types in Teradata can be classified as follows:

### Predefined Data Types

System or predefined data types. A collection of traditional data types such as INTEGER, DECIMAL, FLOAT, VARCHAR, CHAR, BYTE, VARBYTE, and so forth. These are scalar data types.

### ANSI User-defined Types (UDTs)

User-defined distinct and structured data types. The composition of these types and all associated functionality is defined by the user. The types can be developed in C or C++. Users can determine the following:

- The data type or types that comprise the UDT using the CREATE TYPE statement.
- The ordering and casting functionality by authoring the appropriate routines and using the CREATE CAST and CREATE ORDERING statements to form associations for the type.
- The import and export behavior (transform behavior) by authoring the appropriate routines and using the CREATE TRANSFORM statement to form associations for the type.

Users can also create methods to augment the functionality associated with the UDT.

## Complex Data Types (CDTs)

Data types provided by Teradata with some similarity in functionality to UDTs:

- They are nonscalar. For most types, relational-style operations are not applicable.
- They have their own individual literal forms.
- They have formats for import and export that are defined by an ANSI standard, a formal specification, or is Teradata specified.
- Teradata may include methods or functions that provide additional functionality for the data types.

The following are some CDTs provided by Teradata:

- Period data types
- Geospatial data types such as ST\_Geometry, MBR, and MBB
- ARRAY/VARRAY
- XML
- JSON
- DATASET

## Core Data Type Attributes

The following table lists the main Teradata SQL data type attributes.

Data Type Attribute	ANSI	Teradata Extension to ANSI
NOT NULL	X	
UPPERCASE		X
[NOT] CASESPECIFIC		X
FORMAT <i>format_string</i>		X
TITLE <i>char_string</i>		X
AS <i>name</i>	X	
NAMED <i>name</i>		X
DEFAULT <i>value</i>	X	
DEFAULT USER		X
DEFAULT DATE		X
DEFAULT TIME		X
DEFAULT NULL	X	
WITH DEFAULT		X

Data Type Attribute	ANSI	Teradata Extension to ANSI
WITH TIME ZONE	X	
CHARACTER SET	X	

The expression “data type attributes” is non-ANSI, as are attributes such as TITLE and FORMAT.

Restrictions apply to some attributes. For example, you cannot use the WITH DEFAULT attribute with UDT data types. For more information, see:

- [Default Value Control Phrases](#)
- [Data Type Formats and Format Phrases](#)
- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144

## Storage and Constraint Attributes

The following table lists Teradata SQL storage and constraint attributes.

Attribute Family	Data Type Attribute	ANSI	Teradata Extension to ANSI
Column Storage	COMPRESS		X
	COMPRESS NULL		X
	COMPRESS <i>constant_value</i>		X
	COMPRESS ( <i>multivalue_list</i> )		X
	COMPRESS USING <i>compress_udf</i> DECOMPRESS USING <i>decompress_udf</i>		X
Column Constraints	CONSTRAINT	X	
	CONSTRAINT UNIQUE	X	
	CONSTRAINT PRIMARY KEY	X	
	CONSTRAINT CHECK ( <i>boolean_condition</i> )	X	
	CONSTRAINT REFERENCES <i>table_name</i> <i>column_name</i>	X	
Table Constraints	FOREIGN KEY ( <i>column_name_list</i> )	X	
	PRIMARY KEY	X	
	UNIQUE	X	
	CHECK ( <i>boolean_expression</i> )	X	
	REFERENCES <i>table_name</i>	X	



Attribute Family	Data Type Attribute	ANSI	Teradata Extension to ANSI
	[(column_name_list)]		

Restrictions apply to some attributes such as COMPRESS. For more information, see:

- [COMPRESS and DECOMPRESS Phrases](#).
- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144

## Data Type Attributes and Views

When defining a view, any expression in the SELECT expression list can include a data definition. The data type attributes determine the form of the view display. The data type attributes for a view column can differ from those defined for the associated column of the underlying base tables or views. However, not all data definitions are relevant to view expressions.

If you define data type attributes for a view column, these attributes will override any data type attributes defined for the associated column of the underlying base tables or views. For example, a TITLE phrase in the CREATE VIEW statement will override a TITLE phrase in the CREATE TABLE statement of the underlying table.

Any data type attributes defined for a column in the underlying base tables or views are carried over to the view only if the associated view column is not modified in any way.

## Example: Data Type Attribute Carried Over to a View

In this example, the title “This is the title” defined in table *tb* is automatically carried over to the view since there were no changes to column *b* in the view.

```
CREATE TABLE tb
(
  a INTEGER,
  b INTEGER TITLE 'This is the title');

CREATE VIEW vtb AS
(SELECT a, b
 FROM tb);

SELECT TITLE (vtb.b);
```

The SELECT statement produces the following output:

```
Title(This is the title)
-----
This is the title
```

## Example: Overriding the Data Type Attribute Defined in a Base Table

In this example, the *vtb2* view is created from table *tb* in example 1. The view definition changes the name of column *b* to *c*. Therefore, the title “This is the title” defined in table *tb* is not carried over to the view.

```
CREATE VIEW vtb2 AS
(SELECT a, b AS c
 FROM tb);

SELECT TITLE(vtb2.c);
```

The SELECT statement produces the following output:

```
Title(c)
-----
c
```

## Example: Data Type Attribute Not Carried Over to a View

In this example, the *vtb3* view is created from table *tb* in example 1. The title “This is the title” defined in table *tb* is not carried over to the view because column *b* is an expression in the SELECT list of the view definition.

```
CREATE VIEW vtb3 AS
(SELECT a, b (SMALLINT)
 FROM tb);

SELECT TITLE(vtb3.b);
```

The SELECT statement produces the following output:

```
Title(b)
-----
b
```

## Data Conversion

You can use data definition phrases in expressions to convert data to another type or modify data attributes.

When used to modify the attributes of returned values, the data description phrase immediately follows the column being modified.

## Syntax

```
expression ( data_type_list )
```

### Syntax Elements

#### *expression*

The data expression to be converted to the new definition defined by *data\_type\_list*

#### *data\_type\_list*

A data type declaration or data attributes or both.

List elements must be separated by commas.

---

#### Note:

This syntax is called Teradata conversion syntax, and is non-ANSI. Using Teradata conversion syntax is strongly discouraged. It is a Teradata extension to the ANSI SQL:2011 standard and is retained only for backward compatibility with existing applications. You should always use the CAST function to perform conversions in new applications to ensure ANSI compatibility.

---

For example, in the following SELECT statement, a TITLE phrase overrides the default heading (the column name) for EmpNo, and a FORMAT phrase modifies the display format defined for Salary data in the CREATE TABLE statement.

```
SELECT EmpNo (TITLE 'Emp#'), Salary (FORMAT '$$$,$$9.99')
FROM Employee;
```

Restrictions apply to the data types that Teradata conversion syntax supports. For example, you cannot use Teradata conversion syntax to convert data to a UDT. For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## ANSI SQL-Compliant Data Conversion

For applications that need to conform to ANSI as well as to modify attributes of a value, use the CAST function instead of Teradata conversion syntax.

To select Name as a fixed length field and Salary as an integer value:

```
SELECT CAST (Name AS CHAR(12)), CAST (Salary AS INTEGER)
FROM Employee ;
```

## Related Information

For information on Teradata conversion syntax and the CAST function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## COMPRESS and DECOMPRESS Phrases

Compresses data values and nulls in one or more columns of a table, and decompresses the previously compressed values.

### ANSI Compliance

COMPRESS and DECOMPRESS are Teradata extensions to the ANSI SQL:2011 standard.

## Syntax

```
{ COMPRESS [ constant | ( { constant | NULL } [,...] ) ] |  
  COMPRESS_USING_phrase DECOMPRESS_USING_phrase  
} [...]
```

### Note:

The order of *COMPRESS\_USING\_phrase* and *DECOMPRESS\_USING\_phrase* is not important.

### Syntax Elements

#### *COMPRESS\_USING\_phrase*

```
COMPRESS USING [ dbname. ] compress_udf
```

#### *DECOMPRESS\_USING\_phrase*

```
DECOMPRESS USING [ dbname. ] decompress_udf
```

### COMPRESS With No Arguments

Compress nulls only.

Note that if you do not specify COMPRESS, nulls are not automatically compressed.

## Multivalue Compression (MVC) Specification

### COMPRESS

Compress column data using multivalue compression.

#### *constant*

A set of values to be compressed.

You can specify a single *constant* value, or a multivalued, comma-separated list of up to 255 distinct *constant* values enclosed in parentheses.

### NULL

Nulls will be compressed.

NULL can be specified alone, but it must be enclosed in parentheses. This is the same as specifying COMPRESS without an argument.

NULL can be specified with up to 255 *constant* values in a multivalued, comma-separated list enclosed in parentheses.

## Algorithmic Compression (ALC) Specification

### COMPRESS USING

Compress column data using algorithmic compression.

### DECOMPRESS USING

Decompress column data that was previously compressed using algorithmic compression.

#### *dbname*

The name of the database in which the compress or decompress user-defined function (UDF) is stored. If a database name is not specified, the default database used is SYSUDTLIB.

You should create all compress and decompress UDFs in the SYSUDTLIB database. Note that compression functions supplied by Teradata are located in TD\_SYSFNLIB.

#### *compress\_udf*

The name of the UDF used to compress values in the column.

#### *decompress\_udf*

The name of the UDF used to decompress values in the column.

## Number of Columns that Can Be Compressed

You can compress as many columns as practical for any table. However, compressing a substantial number of values in a table using MVC can contribute to table header overflow.

IF you specify ...	THEN ...
COMPRESS with: <ul style="list-style-type: none"> <li>• no argument</li> <li>• NULL enclosed in parentheses</li> </ul>	nulls are compressed for nullable columns. NULL cannot be specified in a COMPRESS phrase for a column that is declared NOT NULL.
COMPRESS with a <i>constant</i>	the indicated value is compressed. For nullable columns, nulls are also compressed.
COMPRESS with: <ul style="list-style-type: none"> <li>• a multi-valued, comma-separated list of <i>constant</i> values enclosed in parentheses</li> <li>• a multi-valued, comma-separated list of <i>constant</i> values and NULL enclosed in parentheses</li> </ul>	nulls and the specified distinct values are compressed for nullable columns. Nulls are compressed for nullable columns regardless of whether NULL appears in the list. NULL cannot appear in a COMPRESS phrase for a column that is declared NOT NULL.
COMPRESS USING <i>compress_udf</i> DECOMPRESS USING <i>decompress_udf</i>	nulls are compressed for nullable columns. Non-null column values that are not specified in the value compression list are compressed using the specified compress UDF, and decompressed using the specified decompress UDF.

## Multi-value Compression (MVC)

You can compress data at the column level using multi-value compression, a lossless, dictionary-based compression scheme. With MVC, you specify a list of values to be compressed when defining a column in the CREATE TABLE/ALTER TABLE statement. When you insert a value into the column which matches a value in the compression list, a corresponding compress bit is set instead of storing the actual value, thus saving the disk storage space.

The best candidates for compression are the most frequently occurring values in each column. MVC is a good compression scheme when there are many repeating values for a column.

If MVC is not an efficient compression scheme for data in a particular column, you can compress the column using algorithmic compression (ALC). You can specify MVC alone, or both MVC and ALC on the same column. If you define both on the same column, ALC is applied only to those non-null values that are not specified in the value compression list of the MVC specification. You can also use MVC together with block-level compression (BLC).

You can use MVC to compress columns with these data types:

- Any numeric data type
- BYTE
- VARBYTE

- CHARACTER
- VARCHAR
- DATE

To compress a DATE value, you must specify the value as a Date literal using the ANSI DATE format (DATE 'YYYY-MM-DD'). For example:

```
COMPRESS (DATE '2000-06-15')
```

- TIME and TIME WITH TIME ZONE
- TIMESTAMP and TIMESTAMP WITH TIME ZONE

To compress a TIME or TIMESTAMP value, you must specify the value as a TIME or TIMESTAMP literal. For example:

```
COMPRESS (TIME '15:30:00')
COMPRESS (TIMESTAMP '2006-11-23 15:30:23')
```

In addition, you can use COMPRESS (NULL) for columns with these data types:

- ARRAY
- Period
- Non-LOB distinct or structured UDT

## Using MVC with CHARACTER or VARCHAR Data Type

When you specify a *constant* for a CHARACTER or VARCHAR data type without the UPPERCASE option, a data value must match the case of the *constant* to be compressed.

For example, if a column named Doc\_Type is defined as follows:

```
Doc_Type CHARACTER(6) COMPRESS 'Manual'
```

then each value must be entered as 'Manual' in order to be compressed. A value in a different case, such as 'MANUAL' or 'manual,' is not compressed.

## Rules for Specifying *constant* Values

The following rules apply to a MVC specification that specifies a comma-separated list of *constant* values:

- The maximum number of distinct *constant* values in the list is 255 plus NULL.
- Values can appear in any order.
- NULL can appear at any point in the multi-valued constant list.
- The multi-value list cannot contain duplicate values. Note that case variants are not considered duplicate values if UPPERCASE is not specified.
- NULL cannot appear in the list if the column is declared NOT NULL.

## Compress Values of a CHAR or VARCHAR Column in SHOW TABLE

When the value compression list of a CHARACTER or VARCHAR column contains characters that are not displayable in the current session character set, a SHOW TABLE query displays a Unicode delimited identifier.

## Achieving Maximum Benefit with COMPRESS

Maximum benefit is achieved when the COMPRESS phrase is applied to a column under the following conditions:

- Enough rows contain a compressible value (for example, null, zero, blank, or *constant value*) in the compressed field to exceed the break-even point. For details, see *Teradata Vantage™ - Database Design*, B035-1094.
- The shortened row length results in the elimination of one or more data blocks.

## Algorithmic Compression (ALC)

In some cases, such as when column values are mostly unique, algorithmic compression can provide better compression results than multi-value compression. Algorithmic compression allows you to define your own compression and decompression algorithms and apply them to data at the column level.

You implement the algorithms as external C/C++ scalar UDFs, and then specify them in the column definition of a CREATE TABLE/ALTER TABLE statement. Vantage invokes these algorithms to compress and uncompress the column data when the data is moved into the tables or when data is retrieved from the tables.

Use ALC to implement the compression scheme that is most suitable for data in a particular column. The cost of compression and uncompression depends on the algorithm chosen.

You can specify ALC alone, or both MVC and ALC on the same column. If you define both on the same column, ALC is applied only to those non-null values that are not specified in the value compression list of the MVC specification.

---

### Note:

Using ALC together with block-level compression (BLC) may degrade performance, so this practice is not recommended.

---

You can use algorithmic compression to compress table columns with the following data types:

- ARRAY
- BYTE
- VARBYTE
- BLOB



- CHARACTER
- VARCHAR
- CLOB
- JSON, with some restrictions listed below
- TIME and TIME WITH TIME ZONE
- TIMESTAMP and TIMESTAMP WITH TIME ZONE
- Period types
- Distinct UDTs, with some restrictions listed below
- System-defined UDTs, with some restrictions listed below

## Teradata Compression and Decompression Functions

Teradata provides the following functions for compressing and uncompressing data. These functions are stored in the TD\_SYSFNLIB system database.

Function Type	UDF Name	Description
Compress	TransUnicodeToUTF8	Takes UNICODE character input and stores it in UTF-8 format. This is useful when the characters are in the ASCII script (U+0000 to U+007F) because UTF-8 uses one byte to represent these characters and UNICODE (UTF-16) uses two bytes.
Decompress	TransUTF8ToUnicode	Takes the data previously compressed using the TransUnicodeToUTF8 function and converts it back to UNICODE.
Compress	LZCOMP	Compresses UNICODE character data using the Lempel-Ziv algorithm.
Decompress	LZDECOMP	Uncompresses UNICODE data that was compressed using LZCOMP.
Compress	LZCOMP_L	Compresses LATIN character data using the Lempel-Ziv algorithm.
Decompress	LZDECOMP_L	Uncompresses LATIN data that was compressed using LZCOMP_L.
Compress	CAMSET	Compresses UNICODE character data into partial byte (for example, 4-bit digits or 5-bit alphabetic letters), one byte, or two byte values using a proprietary Teradata algorithm.
Decompress	DECAMSET	Uncompresses the UNICODE character data that was compressed using CAMSET.
Compress	CAMSET_L	Compresses LATIN character data into partial byte (for example, 4-bit digits or 5-bit alphabetic letters), or one byte values using a proprietary Teradata algorithm.

Function Type	UDF Name	Description
Decompress	DECAMSET_L	Uncompresses the LATIN character data that was compressed using CAMSET_L.
Compress	TD_LZ_COMPRESS	Compresses any supported ALC data type or predefined type data using Lempel-Ziv coding.
Decompress	TD_LZ_DECOMPRESS	Uncompresses any supported ALC data type or predefined type data that was compressed using TZ_LZ_COMPRESS.
Compress	TS_COMPRESS	Compresses TIME and TIMESTAMP data.
Decompress	TS_DECOMPRESS	Uncompresses TIME and TIMESTAMP data that was compressed using TS_COMPRESS.
Compress	JSON_COMPRESS	Compresses JSON data.
Decompress	JSON_DECOMPRESS	Uncompresses the JSON data that was compressed using JSON_COMPRESS.

For more information about these functions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

If these functions do not provide optimal compression for your data, you may implement your own UDFs for use in compressing and uncompressing table columns. For more information, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

---

**Note:**

If you create custom ALC UDFs, make sure to test them thoroughly. If the compression or decompression algorithm fails, compressed data may not be recoverable, or may be corrupted.

---

## Restrictions

- You cannot use ALC to compress columns that have a data type of structured UDT.
- The TD\_LZ\_COMPRESS and TD\_LZ\_DECOMPRESS system functions compress all large UDTs including UDT-based system types such as Geospatial, XML, and JSON. However, if you write your own compression functions, the following restrictions apply:
  - Custom compression functions cannot be used to compress UDT-based system types (except for ARRAY and Period types).
  - Custom compression functions cannot be used to compress distinct UDTs that are based on UDT-based system types (except for ARRAY and Period types).
- You cannot write your own compression functions to perform algorithmic compression on JSON type columns. However, Teradata provides the JSON\_COMPRESS and JSON\_DECOMPRESS functions that you can use to perform ALC on JSON type columns.
- You cannot use ALC to compress temporal columns:
  - A column defined as SYSTEM\_TIME, VALIDTIME, or TRANSACTIONTIME.

- The DateTime columns that define the beginning and ending bounds of a temporal derived period column (SYSTEM\_TIME, VALIDTIME, or TRANSACTIONTIME).

You can use ALC to compress period data types in columns that are nontemporal; however, you cannot use ALC to compress derived period columns.

For details about temporal tables, see *Teradata Vantage™ - Temporal Table Support*, B035-1182 and *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186.

- You cannot specify multi-value or algorithmic compression for a row-level security constraint column.

## Example: Compression Using MVC with a *constant* Value

The following example uses MVC to compress a specified date in a DATE column.

```
CREATE TABLE Duration
  (DurationID INTEGER,
   StartDate DATE COMPRESS (DATE '2000-01-01'));
```

## Example: Compression Using MVC with a Multi-valued List

The following example uses MVC to compress a list of date values in a DATE column.

```
CREATE TABLE Duration
  (DurationID INTEGER,
   StartDate DATE COMPRESS (DATE '2000-01-01',
                             DATE '2000-01-02',
                             DATE '2000-01-03'));
```

## Example: Compression Using MVC on an INTEGER Column

The following example uses MVC to compress a list of numeric values in an INTEGER column.

```
CREATE TABLE ID
  (IDNum INTEGER,
   Post INTEGER COMPRESS (44, 45, 63));
```

## Example: Compression Using ALC Only

In this example, assume that the default server character set is UNICODE. The NULLs in the Description column are compressed. The non-null values of the Description column are compressed using the Teradata-supplied function, TransUnicodeToUTF8. The TransUTF8ToUnicode function uncompresses the values compressed by the TransUnicodeToUTF8 function.

```
CREATE TABLE Pendants
  (ItemNo INTEGER,
   Description VARCHAR(1000)
     COMPRESS USING TD_SYSFNLIB.TransUnicodeToUTF8
     DECOMPRESS USING TD_SYSFNLIB.TransUTF8ToUnicode);
```

## Example: Compression Using MVC and ALC

In this example, assume that the default server character set is UNICODE. The NULLs in the Gem column are compressed. MVC is used to compress the values 'amethyst' and 'amber' in the Gem column. Because UPPERCASE is specified, all values of 'amethyst' and 'amber' are compressed regardless of case. The Teradata-supplied function, TransUnicodeToUTF8, compresses all non-null values in the Gem column that are not 'amethyst' or 'amber'. The TransUTF8ToUnicode function uncompresses the values compressed by TransUnicodeToUTF8.

```
CREATE TABLE Pendants
  (ItemNo INTEGER,
   Gem CHAR(10) UPPERCASE COMPRESS ('amethyst', 'amber')
     COMPRESS USING TD_SYSFNLIB.TransUnicodeToUTF8
     DECOMPRESS USING TD_SYSFNLIB.TransUTF8ToUnicode);
```

## Example: ALC Using the LZCOMP Function

In this example, the NULLs in the Description column are compressed. The non-null UNICODE values in the Description column are compressed using the Teradata-supplied LZCOMP function. The LZDECOMP function uncompresses the values compressed by LZCOMP.

```
CREATE MULTISET TABLE Pendants
  (ItemNo INTEGER,
   Gem CHAR(10) UPPERCASE CHARACTER SET UNICODE,
   Description VARCHAR(1000) CHARACTER SET UNICODE
     COMPRESS USING TD_SYSFNLIB.LZCOMP
     DECOMPRESS USING TD_SYSFNLIB.LZDECOMP);
```

## Related Information

For information on ...	See ...
the performance and storage capacity savings benefits of value compression	<i>Teradata Vantage™ - Database Design</i> , B035-1094.
using COMPRESS when defining or modifying columns in a table	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.

For information on ...	See ...
Teradata compression and decompression functions	<i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145.
rules and restrictions for compressing column data	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
rules for creating compress and decompress UDFs	<ul style="list-style-type: none"> <li>• <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li> <li>• <i>Teradata Vantage™ - SQL External Routine Programming</i>, B035-1147.</li> </ul>
compression methods supported by Vantage and a comparison of the various methods	<i>Teradata Vantage™ - Database Administration</i> , B035-1093.

## Constraint Attributes

Constraint attributes specify integrity rules. Constraints can be any of the following types:

- Uniqueness (see [Uniqueness Constraints](#)).
- CHECK (see [CHECK Constraints](#)).
- Referential integrity (see [Referential Constraints](#)).

## Column and Table Constraints

You can specify constraints during table creation and modification.

Column constraints apply to single columns as a part of the column definition. Column constraints include:

- CHECK constraint definition clause on a single column
- PRIMARY KEY constraint definition clause on a single column
- REFERENCES constraint definition clause on a single column
- UNIQUE constraint definition clause

Table constraints apply to multiple columns. Table-level constraints include:

- CHECK constraint definition clause on multiple columns
- REFERENCES constraint definition clause on multiple columns
- PRIMARY KEY constraint definition clause on multiple columns
- UNIQUE constraint definition clause on multiple columns
- FOREIGN KEY constraint definition clause

FOREIGN KEY constraint definitions must also specify a REFERENCES clause.

The full syntax for constraints is in ALTER TABLE and CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144

# Uniqueness Constraints

## Definition

A uniqueness constraint means the table cannot include two or more rows in which the values for the column or set of columns are identical.

Vantage supports the following uniqueness constraints:

- PRIMARY KEY
- UNIQUE

## PRIMARY KEY Constraint

A column or set of columns defined as PRIMARY KEY must also be NOT NULL.

Vantage instantiates a PRIMARY KEY as a unique primary or secondary index.

Only one PRIMARY KEY can be defined for a table.

For more details, see:

- CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ - SQL Fundamentals*, B035-1141
- *Teradata Vantage™ - Database Design*, B035-1094.

## UNIQUE Constraint

A column or set of columns defined as UNIQUE must also be NOT NULL.

Vantage instantiates UNIQUE as a unique primary or secondary index. One or more columns or sets of columns can be defined as UNIQUE for a table.

For more details, see:

- CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ - Database Design*, B035-1094.

## CHECK Constraints

CHECK constraints compare values of a field or fields in the same row with constants or other fields.

## Usage Notes

CHECK constraints are applied to rows generated as candidates for INSERT and UPDATE operations.

You cannot specify CHECK constraints for Identity columns.

If the condition is met or the proposed INSERT or UPDATE contains NULLs, the operation is permitted.

If the condition is not met, an error is reported as a constraint violation. For details on the syntax, see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

For character data, constraints are checked using the current session collation. Therefore, it is possible for a value to meet the constraint for one collation and violate the constraint for another collation.

Also see *Teradata Vantage™ - Database Design*, B035-1094.

## Example: CHECK Constraints

```
CREATE TABLE stats_tbl
  (Id INTEGER
   ,Sex CHAR(1)
   ,EdLev INTEGER
   ,CHECK (Sex = 'F' OR Sex = 'M')
   ,CHECK ((EdLev >= 0) AND (EdLev <= 22)));
```

## Referential Constraints

Use referential constraints to indicate relationships between columns of different tables. There are three specific types of referential constraints.

Referential Constraint Type	DDL Definition	Does It Enforce Referential Integrity?	Level of Referential Integrity Enforcement
Referential constraint	REFERENCES WITH NO CHECK OPTION	No	None
Batch referential integrity constraint	REFERENCES WITH CHECK OPTION	Yes	Request
Referential integrity constraint	REFERENCES	Yes	Row

## Usage Notes

You cannot specify REFERENCES constraints for Identity columns.

For additional information about REFERENCES constraints, see:

- ALTER TABLE and CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ - SQL Fundamentals*, B035-1141
- *Teradata Vantage™ - Database Design*, B035-1094.

# Data Literals

Literals are values coded directly in the text of an SQL statement, view or macro definition text, or CHECK constraint definition text. In general, the system is able to determine the type of a literal by its form.

Literals are also referred to as constants.

## Hexadecimal Byte Literals

Declares a hexadecimal byte literal value.

Hexadecimal literals consist of 0 to 62000 hexadecimal digits delimited by a matching pair of apostrophes, where a hexadecimal digit is a character from 0 to 9, a to f, or A to F.

The modifier following the XB determines the hexadecimal literal data type.

IF a hexadecimal literal uses this form ...	THEN the data type is ...
' <i>hexadecimal digits</i> 'XBV	VARBYTE
' <i>hexadecimal digits</i> 'XB ' <i>hexadecimal digits</i> 'XBF	BYTE

## ANSI Compliance

Hexadecimal byte literals are Teradata extensions to the ANSI SQL:2011 standard.

## Syntax

```
'hexadecimal digits' XB [ V | F ]
```

## Syntax Elements

### *hexadecimal digits*

A string of hexadecimal digits, where a hexadecimal digit is a character from 0 to 9, a to f, or A to F.

### V

The hexadecimal literal is in byte variable format.

### F

The hexadecimal literal is in byte fixed format. This is the default if F or V is not specified.



## Usage Notes

Hexadecimal byte literal is the only form for entering a byte string.

Hexadecimal byte literals are represented by an even number of hexadecimal digits. Hexadecimal literals are extended on the right with zeros when required.

Consider the following table:

```
CREATE TABLE bvalues (b1 BYTE(2));
```

Suppose you insert the hexadecimal byte literal 'C1C'XB into column b1:

```
INSERT bvalues ('C1C'XB);
```

The value is extended on the right with zeros:

```
SELECT * FROM bvalues;

b1
----
C1C0
```

### Example: Hexadecimal Byte Literal

Suppose the column CodeVal has been defined as BYTE(2).

```
CREATE TABLE bvalues (IDVal INTEGER, CodeVal BYTE(2));
```

To insert a BYTE string as a hexadecimal literal, use the following form:

```
INSERT bvalues (112193, '7879'XB) ;
```

To select those rows from CodeVal, specify the conditional like this:

```
SELECT IDVal FROM bvalues WHERE CodeVal = '7879'XB ;
```

## Numeric Literals

### Definition

A numeric literal is a string of 1 to 40 characters selected from the following:

- plus sign
- minus sign

- digits 0 through 9
- decimal point

Numeric literals are also referred to as numeric constants.

**Note:**

Vantage also supports a hexadecimal form of numeric literals to represent integer values. For more information, see [Hexadecimal Integer Literals](#).

## Types of Numeric Literals

There are three kinds of numeric literals:

- Integers (see [Integer Literals](#) and [Hexadecimal Integer Literals](#))
- Decimals (see [Decimal Literals](#))
- Floating point numbers (see [Floating Point Literals](#))

### Examples: Valid Numeric Literals

Type	Examples		
BYTEINT	127	-36	-128
SMALLINT	32767	-12000	-32768
INTEGER	32768	-60400	2147483647
DECIMAL	0.0	-23554367273149967931.	2147483650
FLOATING POINT	1E1	1.4E6	18E-3

### Examples: Nonvalid Numeric Literals

This literal is not valid ...	Because it contains ...
123456789012345678901234567890123456789	more than 38 digits.
\$20,000.00	a dollar sign and a comma.
-38.7E2945	four digits following the E.

For the rules on what constitutes valid numeric literals, see the following sections:

- [Integer Literals](#)
- [Decimal Literals](#)
- [Floating Point Literals](#)

## Determining the Data Type of a Numeric Literal

The data type of a numeric literal is determined by the range of the literal value. The type used is the smallest that can contain the value.

For example, the data type of the numeric literal 127 is BYTEINT because it is the smallest type that can fit the value 127.

For decimal literals, the total number of digits determine the precision and the number of digits to the right of the decimal point determine the scale.

FOR details on ...	SEE ...
the range of values of integer literals	<a href="#">Integer Literals.</a>
determining the scale and precision of a decimal literal	<a href="#">Decimal Literals.</a>

## Implicit Conversion of Numeric Literals

Depending on the kind of operation performed or the type of column in which a value is to be stored, SQL may convert numeric literal constants from one numeric data type to another.

If a literal is outside the range for its required type, an error is reported during conversion.

For more information on numeric to numeric data type conversions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Converting Character Strings to Numeric Literals

Character data can contain a string that is intended to be interpreted as a numeric value (for example, '15'). In such cases, Vantage attempts to convert the string to a numeric value whenever the context makes such a conversion necessary. If the character string does not represent a valid numeric value, an error is reported.

When data in a character column must be compared with data in a numeric column, the character data and the numeric data are converted to FLOAT before the comparison is made. Note that this can result in repeated conversions during data access.

If numeric data is defined and stored as a character string in a character column, and a SELECT operation uses a full table scan to compare the character column with a numeric literal, then the character column is converted to numeric in every row of the table.

For more information on character to numeric data type conversions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Integer Literals

Declares literal strings of integer numbers in an expression.

## Syntax

```
[ + | - ] n
```

## Syntax Elements

**+, -**

An optional sign.

The default is +.

***n***

Any valid integer.

## Components of Integer Literals

Integer literals consist of an optional sign followed by a sequence of digits. Spaces and new line characters are not allowed in a literal except after the optional sign.

## Integer Literal Data Types

The data type of an integer literal is determined by the range of the literal value. The type used is the smallest that can contain the value.

IF an integer literal fits in this range of values ...	THEN the data type of the integer literal is ...
-128 to 127	BYTEINT
-32768 to -129	SMALLINT
128 to 32767	
-2147483648 to -32769	INTEGER
32768 to 2147483647	

A numeric literal that is outside the range of the INTEGER type is assigned to the DECIMAL type, unless it is outside the range of values that a DECIMAL type can represent. For more information, see [Decimal Literals](#).

Numeric literals are not assigned to the BIGINT type. If you need a numeric literal that is outside the range of the INTEGER type to be of a type other than DECIMAL, you can explicitly cast the literal to the desired type. For example, you can use the CAST function to explicitly cast a numeric literal to BIGINT:

```
SELECT ProdID
FROM PartsTbl
WHERE CustID = CAST(9876543210 AS BIGINT);
```

## Examples: Integer Literals

The following numbers are examples of integer literals.

```
12
0
-5
```

## Related Information

FOR information on ...	SEE ...
BYTEINT data types	<a href="#">BYTEINT Data Type.</a>
SMALLINT data types	<a href="#">SMALLINT Data Type.</a>
INTEGER data types	<a href="#">INTEGER Data Type.</a>

## Hexadecimal Integer Literals

Declares a hexadecimal integer literal value.

Hexadecimal literals consist of 0 to 16 hexadecimal digits delimited by a matching pair of apostrophes. Spaces and new line characters are not allowed in a literal.

The modifiers following the X determine the hexadecimal literal data type.

IF a hexadecimal literal uses this form ...	THEN the data type is ...	AND the maximum hexadecimal digits is...
'hexadecimal digits'X 'hexadecimal digits'X1 'hexadecimal digits'X14	INTEGER	8
'hexadecimal digits'X12	SMALLINT	4
'hexadecimal digits'X11	BYTEINT	2
'hexadecimal digits'X18	BIGINT	16

## ANSI Compliance

Hexadecimal literals are Teradata extensions to the ANSI SQL:2011 standard.

## Syntax

```
' hexadecimal digits' X [ | [ 1 | 2 | 4 | 8 ] ]
```

**Note:**

You must type the bold or colored vertical bar.

**Syntax Elements*****hexadecimal digits***

A string of hexadecimal digits, where a hexadecimal digit is a character from 0 to 9, a to f, or A to F.

**1**

Integers with a BYTEINT data type.

**2**

Integers with a SMALLINT data type.

**4**

Integers with an INTEGER data type. This is the default if 1, 2, 4, or 8 is not specified.

**8**

Integers with a BIGINT data type.

**Usage Notes**

Hexadecimal integer literals are represented by an odd or even number of hexadecimal digits. The hexadecimal literal is right-justified. For example, the value 1000 can be expressed as any of the following:

```
'3e8'X
'0003e8'X
'000003e8'X
```

1000 hex would be '1000'x, which is  $16 \times 16 \times 16 = 4096$ .

'3e8'x =  $3 \times 16 \times 16 + e \times 16 + 8$  and e is 14 so we get 1000.

Note that the literal 1000 would be a SMALLINT whereas all the hex constants shown here are INTEGERS.

**Example: Hexadecimal Integer Literal**

Consider the following table:

```
CREATE TABLE id_pairs (region_id INTEGER, region CHAR(20));
```

Submit the following statement to find the value of the region column where the region\_id is a hexadecimal value of 3e8:

```
SELECT region FROM id_pairs WHERE region_id = '3e8'X;
```

## Decimal Literals

Declares literal strings of decimal numbers in an expression.

### Syntax

```
[ + | - ] { n . | .n | n . n | n }
```

### Syntax Elements

**+, -**

An optional sign.

The default is +.

***n***

Any valid integer.

### Components of Decimal Literals

Decimal literals consist of the following components, reading from left-to-right:

1. Optional sign
2. Sequence of digits (including none)
3. Decimal point
4. Optional sequence of digits.

Spaces and new line characters are not allowed in a literal except after the optional sign.

### Decimal Literal Data Types

Decimal literals include the following types:

- DECIMAL
- NUMERIC

### Scale and Precision

The total number of digits in a decimal literal determine the precision and the number of digits to the right of the decimal point determine the scale. Leading and trailing zeros are included in the counts. The precision cannot exceed 38.

A numeric literal that is outside the range of the INTEGER type is assigned to DECIMAL(*n*, 0), where *n* is the number of digits in the literal, excluding leading zeros.

### Examples: Decimal Literals

The following numbers are examples of decimal literals.

```
3.14159
253.
-88.234
.29
2147483650
```

### Related Information

For information on DECIMAL data types, see [DECIMAL/NUMERIC Data Types](#).

## Floating Point Literals

Declares literal strings of floating point numbers in an expression.

### Syntax

```
[ + | - ] { nE | n.E | .nE | n.nE } [ + | - ] m
```

### Syntax Elements

**+, -**

An optional sign.

The default is +.

***n***

Any valid integer representing the whole and, optionally, fractional component of the mantissa.

The total number of digits cannot exceed 15, excluding leading zeros in the whole component of the mantissa.

**E**

The symbol indicating that what follows is the exponent for the number.

***m***

Any valid integer number representing the exponent for the number.

The total number of digits cannot exceed three, including leading zeros.



## Components of Floating Point Literals

Floating point literals consist of the following components, reading from left-to-right:

1. Optional sign
2. Sequence of digits (including none) representing the whole number portion of the mantissa
3. Optional decimal point
4. Sequence of digits (including none) representing the fractional portion of the mantissa
5. Literal character E
6. Optional sign
7. Sequence of digits (including none) representing the exponent

Spaces and new line characters are not allowed in a literal except after the first optional sign.

The second optional sign should immediately follow the literal character E and come before the sequence of digits representing the exponent. However, Teradata does not issue a syntax error if you specify the sign in the middle of the digits representing the exponent or after the digits representing the exponent.

## Floating Point Literal Data Types

Floating point literals are treated as having a FLOAT data type, which is treated as equivalent to:

- REAL
- DOUBLE PRECISION

## Examples: Floating Point Literals

The following numbers are examples of floating point literals.

```
1E100
3.14E-10
6.023E23
```

## Related Information

FOR information on ...	SEE ...
<ul style="list-style-type: none"> <li>FLOAT data types</li> <li>REAL data types</li> <li>DOUBLE PRECISION data types</li> </ul>	<a href="#">FLOAT/REAL/DOUBLE PRECISION Data Types.</a>
potential problems associated with floating point values in comparisons and computations	<a href="#">Operations on Floating Point Values.</a>

## NUMBER Literals

Vantage does not provide NUMBER literals. You can use other numeric literals to initialize NUMBER fields; however, you cannot directly initialize a NUMBER with more than 38 digits, or when using FLOAT literals, more than 14 digits. For example, you can use the CAST function to explicitly cast a numeric literal to NUMBER:

```
SELECT ProdID
FROM PartsTbl
WHERE CustID = CAST(9876543210 AS NUMBER);
```

You can also create a NUMBER value from a character string. Normally, you cannot have more than 38 numeric characters in a string literal; however, for the NUMBER data type, you can have up to 64 numeric characters in a string literal. For example:

```
CREATE TABLE t1 (pk INTEGER, col1 NUMBER);
INSERT INTO t1 (1, '123456789012345678901234567890123456789');
```

## Date and Time Literals

Date and time literals declare date, time, or timestamp values in an SQL expression.

### ANSI DateTime Literals

ANSI SQL provides DateTime literals to represent date, time, and timestamp values. There are three types of ANSI SQL DateTime literals:

- DATE
- TIME
- TIMESTAMP

### Teradata SQL Date and Time Literals

Teradata also provides a non-ANSI extension to DateTime functionality.

Teradata SQL literals used with date or time values are simple string literals and are interpreted as character data. These are converted to a date or time value based on the context, which is usually provided by a FORMAT clause.

The existing Teradata SQL operations on character string literals used to represent date and time values are supported as a non-ANSI extension to DateTime functionality.

While Teradata SQL date and time literals come in a variety of formats for the character representations of date and time values, and so cannot be restricted to a default or standard format, ANSI DateTime literals are restricted to a strict set of formats.

Existing functionality for Teradata SQL date literals is preserved.

For example, suppose a column is defined as DATE with a format of 'YY/MM/DD'. If the value '97/12/31' is encountered as a character literal assigned to a DATE column, the process works just as it did prior to the introduction of ANSI DateTime formats because the character literal is implicitly cast to the DATE format.

Unless explicitly stated, this section only discusses ANSI DateTime literals.

## DateTime Literals

The ANSI SQL DateTime literals provide DateTime values in source text, supporting a means for declaring values for DATE, TIME, and TIMESTAMP data types.

### Keywords for ANSI DateTime Literals

ANSI SQL DateTime literals differ from other SQL literals in that they are always prefaced by a keyword or keywords.

Here is an example showing the TIMESTAMP keyword prefacing a timestamp literal:

```
TIMESTAMP '1999-07-01 15:00:00-08:00'
```

Literals expressed in this format are interpreted exclusively as ANSI DateTime values.

## Date Literals

Declares a date value in ANSI DATE format in an expression.

ANSI DATE literal is the preferred format for date constants. All date operations accept this format.

### Syntax

```
DATE 'string'
```

### Syntax Elements

#### *string*

A 10-character string enclosed in apostrophes in the following form:

YYYY-MM-DD

- YYYY represents year. The valid range is 0001 through 9999, inclusive. You must specify all four digits.
- MM represents month. The valid range is 01 through 12, inclusive. You must specify both digits.
- DD represents day. The valid range is 01 through 31, inclusive, constrained by Gregorian calendar definitions. You must specify both digits.

## Usage Notes

Date literals consist of the word DATE followed by a character string literal. This character string specifies the date value.

The year, month, and day components of the string literal must be separated by hyphens.

Spaces and new line characters are not allowed in a literal except after the keyword DATE.

## Data Type

DATE

### Example: Date Literals

The following example selects all classes from the Classes table that start on January 6, 1998.

```
SELECT *
FROM CLASSES
WHERE startdate = DATE '1998-01-06';
```

## Related Information

For information on DATE data types, see [DATE Data Type](#).

## Time Literals

Declares a time value in an expression.

## Syntax

```
TIME 'string'
```

## Syntax Elements

*string*

8- to 21-character string in one of these formats:

Format	Description
hh:mi:ss	Time with no fractional seconds digits or Time Zone: <ul style="list-style-type: none"> <li>hh represents the hour of the day. The valid range is 00–23, inclusive. You must specify both digits.</li> <li>mi represents minute of the hour. The valid range is 00–59, inclusive. You must specify both digits.</li> <li>ss represents seconds. The valid range is 00–61. You must specify both digits.</li> </ul>
hh:mi:sssignhh:mi	Time with a specified Time Zone, but no fractional seconds digits:

Format	Description
	<ul style="list-style-type: none"> <li>• hh represents the hour of the day. The valid range is 00–23, inclusive. You must specify both digits.</li> <li>• mi represents minute of the hour. The valid range is 00–59, inclusive. You must specify both digits.</li> <li>• ss represents seconds. The valid range is 00–61. You must specify both digits.</li> <li>• signhh:mi represents the hours and minutes in the Time Zone offset. The valid range is -12:59 through +13:00, inclusive.</li> </ul>
hh:mi:ss.ssssss	Time with up to six fractional seconds digits, but no Time Zone: <ul style="list-style-type: none"> <li>• hh represents the hour of the day. The valid range is 00–23, inclusive. You must specify both digits.</li> <li>• mi represents minute of the hour. The valid range is 00–59, inclusive. You must specify both digits.</li> <li>• ss.ssssss represents seconds. The valid range for the first two digits is 00–61. You must specify both digits. You can specify from one to six fractional digits.</li> </ul>
hh:mi:ss.ssssss signhh:mi	Time with up to six fractional seconds and a Time Zone offset: <ul style="list-style-type: none"> <li>• hh represents the hour of the day. The valid range is 00–23, inclusive. You must specify both digits.</li> <li>• mi represents minute of the hour. The valid range is 00–59, inclusive. You must specify both digits.</li> <li>• ss.ssssss represents seconds. The valid range for the first two digits is 00–61. You must specify both digits. You can specify from one to six fractional digits.</li> <li>• signhh:mi represents the hours and minutes in the Time Zone offset. The valid range is -12:59 through +13:00, inclusive.</li> </ul>

The colons are required between the first three elements, and the decimal point is required if fractional seconds are specified. A decimal point is not allowed if there are no fractional digits. Spaces and new line characters are not allowed in a literal except after the keyword TIME.

IF the format of the Time literal is ...	THEN the data type is ...
hh:mi:ss	TIME(0).
hh:mi:sssignhh:mi	TIME(0) WITH TIME ZONE.
hh:mi:ss.ssssss	TIME(n), where n is the number of fractional seconds digits.
hh:mi:ss.ssssss signhh:mi	TIME(n) WITH TIME ZONE, where n is the number of fractional seconds digits.

### Example 1: hh:mi:ss Format

The following example selects all classes from the Classes table that start at 3:30 PM.

```
SELECT *
FROM Classes
WHERE start_time = TIME '15:30:00';
```

### Example 2: hh:mi:sssignhh:mi Format

```
SELECT *
FROM Classes
WHERE start_time = TIME '10:37:12-08:00';
```

### Example 3: hh:mi:ss.ssssss Format

```
SELECT *
FROM Classes
WHERE start_time = TIME '10:36:02.123456';
```

### Example 4: hh:mi:ss.sssssssignhh:mi Format

```
SELECT Customer_ID
FROM Messages
WHERE start_time = TIME '21:17:35.123456+07:30'
```

## Related Information

FOR information on ...	SEE ...
TIME data types	<a href="#">TIME Data Type.</a>
TIME WITH TIME ZONE data types	<a href="#">TIME WITH TIME ZONE Data Type.</a>

## Timestamp Literals

Declares a timestamp value in an expression.

### Syntax

```
TIMESTAMP 'string'
```

### Syntax Elements

#### *string*

Character string in apostrophes in one of these formats:

Format	Description
YYYY-MM-DD hh:mi:ss	<p>Timestamp with no time zone or fractional seconds digits:</p> <ul style="list-style-type: none"> <li>• YYYY represents year. The valid range is 0001 through 9999, inclusive. You must specify all four digits.</li> <li>• MM represents month. The valid range is 01 through 12, inclusive. You must specify both digits.</li> <li>• DD represents day. The valid range is 01 through 31, inclusive, constrained by Gregorian calendar definitions. You must specify both digits, and follow them by a single pad character.</li> <li>• hh represents the hour of the day. The valid range is 00–23, inclusive. You must specify both digits.</li> <li>• mi represents minute of the hour. The valid range is 00–59, inclusive. You must specify both digits.</li> <li>• ss represents seconds. The valid range is 00–61. You must specify both digits.</li> </ul>
YYYY-MM-DD hh:mi: sssignhh:mi	<p>A timestamp with a specified time zone offset, but no fractional seconds digits:</p> <ul style="list-style-type: none"> <li>• YYYY represents year. The valid range is 0001 through 9999, inclusive. You must specify all four digits.</li> <li>• MM represents month. The valid range is 01 through 12, inclusive. You must specify both digits</li> <li>• DD represents day. The valid range is 01 through 31, inclusive, constrained by Gregorian calendar definitions. You must specify both digits, and follow them by a single pad character.</li> <li>• hh represents the hour of the day. The valid range is 00–23, inclusive. You must specify both digits.</li> <li>• mi represents minute of the hour. The valid range is 00–59, inclusive. You must specify both digits.</li> <li>• ss represents seconds. The valid range is 00–61. You must specify both digits.</li> <li>• signhh:mi represents the hours and minutes in the Time Zone offset. The valid range is -12:59 through +13:00, inclusive. sign is + or -</li> </ul>
YYYY-MM-DD hh:mi:ss. ssssss	<p>A timestamp with up to six fractional seconds digits, but no Time Zone:</p> <ul style="list-style-type: none"> <li>• YYYY represents year. The valid range is 0001 through 9999, inclusive. You must specify all four digits.</li> <li>• MM represents month. The valid range is 01 through 12, inclusive. You must specify both digits.</li> <li>• DD represents day. The valid range is 01 through 31, inclusive, constrained by Gregorian calendar definitions. You must specify both digits, and follow them by a single pad character.</li> <li>• hh represents the hour of the day. The valid range is 00–23, inclusive. You must specify both digits.</li> <li>• mi represents minute of the hour. The valid range is 00–59, inclusive. You must specify both digits.</li> <li>• ss.ssssss represents seconds. The valid range for the first two digits is 00–61. You must specify both digits. You can specify from one to six fractional digits.</li> </ul>

Format	Description
YYYY-MM-DD	<p>A timestamp with up to six fractional seconds and a time zone offset:</p> <ul style="list-style-type: none"> <li>• YYYY represents year. The valid range is 0001 through 9999, inclusive. You must specify all four digits.</li> <li>• MM represents month. The valid range is 01 through 12, inclusive. You must specify both digits.</li> <li>• DD represents day. The valid range is 01 through 31, inclusive, constrained by Gregorian calendar definitions. You must specify both digits, and follow them by a single pad character.</li> <li>• hh represents the hour of the day. The valid range is 00–23, inclusive. You must specify both digits.</li> <li>• mi represents minute of the hour. The valid range is 00–59, inclusive. You must specify both digits.</li> <li>• ss.ssssss represents seconds. The valid range for the first two digits is 00–61. You must specify both digits. You can specify from one to six fractional digits.</li> <li>• signhh:mi represents the hours and minutes in the Time Zone offset. The valid range is -12:59 through +13:00, inclusive. sign is + or -</li> </ul>

The colons are required between the hour, minute, and second elements, and the decimal point is required if fractional seconds are specified. A decimal point is not allowed if there are no fractional digits. Spaces and new line characters are not allowed in a literal except after the keyword **TIMESTAMP**.

IF the format of the Timestamp literal is ...	THEN the data type is ...
YYYY-MM-DD hh:mi:ss	TIMESTAMP(0).
YYYY-MM-DD hh:mi:sssignhh:mi	TIMESTAMP(0) WITH TIME ZONE.
YYYY-MM-DD hh:mi:ss.ssssss	TIMESTAMP( <i>n</i> ), where <i>n</i> is the number of fractional seconds digits.
YYYY-MM-DD hh:mi:ss.ssssssignhh:mi	TIMESTAMP( <i>n</i> ) WITH TIME ZONE, where <i>n</i> is the number of fractional seconds digits.

### Example 1: YYYY-MM-DD hh:mi:ss Format

The following example selects all classes from the **Classes** table that are timestamped November 23 2006 at 3:30:23 PM.

```
SELECT *
FROM Classes
WHERE Time_stamp = TIMESTAMP '2006-11-23 15:30:23';
```



**Example 2: YYYY-MM-DD hh:mi:sssignhh:mi Format**

```
SELECT *
FROM Classes
WHERE Time_stamp = TIMESTAMP '2002-01-01 10:37:12-08:00'
```

**Example 3: YYYY-MM-DD hh:mi:ss.ssssss Format**

```
SELECT *
FROM Classes
WHERE Time_stamp = TIMESTAMP '1995-07-31 10:36:02.123456'
```

**Example 4: YYYY-MM-DD hh:mi:ss.ssssssssignhh:mi Format**

```
SELECT *
FROM Classes
WHERE Time_stamp = TIMESTAMP '1492-10-27 21:17:35.456123+07:30'
```

**Related Information**

FOR information on ...	SEE ...
TIMESTAMP data types	<a href="#">TIMESTAMP Data Type.</a>
TIMESTAMP WITH TIME ZONE data types	<a href="#">TIMESTAMP WITH TIME ZONE Data Type.</a>

## Interval Literals

Interval literals provide a means for declaring interval values in expressions.

Interval literals differ from other SQL literals in that keywords introduce and follow them.

**General Syntax**

```
INTERVAL [ sign ] 'string' interval_qualifier
```

**Syntax Elements*****sign***

An optional minus sign to designate a negative interval. The default is a positive interval.

Note that the sign must be outside the apostrophes that enclose *string*.

***string***

A character string. Spaces and new line characters are not allowed between the apostrophes.

***interval\_qualifier***

A keyword or keywords indicating the Interval literal type. Possible values are:

- YEAR
- YEAR TO MONTH
- MONTH
- DAY
- DAY TO HOUR
- DAY TO MINUTE
- DAY TO SECOND
- HOUR
- HOUR TO MINUTE
- HOUR TO SECOND
- MINUTE
- MINUTE TO SECOND
- SECOND

You cannot specify precision with the *interval\_qualifier*.

**Interval Literal Categories**

Interval literals fall under one of two categories. You cannot mix literals that fall under one category with literals that fall under the other category.

- Year-Month
- Day-Time

Year-Month Literals	Day-Time Literals	
<ul style="list-style-type: none"> <li>◦ YEAR</li> <li>◦ YEAR TO MONTH</li> <li>◦ MONTH</li> </ul>	<ul style="list-style-type: none"> <li>◦ DAY</li> <li>◦ DAY TO HOUR</li> <li>◦ DAY TO MINUTE</li> <li>◦ DAY TO SECOND</li> <li>◦ HOUR</li> </ul>	<ul style="list-style-type: none"> <li>◦ HOUR TO MINUTE</li> <li>◦ HOUR TO SECOND</li> <li>◦ MINUTE</li> <li>◦ MINUTE TO SECOND</li> <li>◦ SECOND</li> </ul>

**Interval Literal Data Types**

The data type for a literal is derived directly from the *interval\_qualifier*.

**Note:**

You cannot specify precision with any of the *interval\_qualifier* types.

## INTERVAL YEAR Literals

Declares an INTERVAL YEAR value in an expression.

Result type: [INTERVAL YEAR Data Type](#)

### ANSI Compliance

INTERVAL YEAR literals are partly ANSI SQL:2011 compliant.

The ANSI definition places the optional sign for the interval within the apostrophes; the Teradata implementation places the optional sign outside the apostrophes.

### Syntax

```
INTERVAL [ sign ] 'string' YEAR
```

### Syntax Elements

#### *sign*

An optional minus sign to indicate a negative interval. The default is a positive interval.

Note that the sign must be outside the apostrophes that enclose *string*.

#### *string*

One to four digits representing years. Spaces and new line characters are not allowed between the apostrophes.

---

#### Note:

Only digits within the apostrophes are parsed and converted to numeric. For example, '1.05' is treated as '105'.

---

### Example: INTERVAL YEAR Literal

This example adds an interval of -2 years to the current system date. (For this example, assume the current system date is 11-03-1999.)

```
SELECT INTERVAL - '2' YEAR + CURRENT_DATE;

      (-2+Date)
-----
1997/11/03
```

## INTERVAL YEAR TO MONTH Literals

Declares an INTERVAL YEAR TO MONTH value in an expression.

Result type: [INTERVAL YEAR TO MONTH Data Type](#)

### ANSI Compliance

INTERVAL YEAR TO MONTH literals are partly ANSI SQL:2011 compliant.

The ANSI definition places the optional sign for the interval within the apostrophes; the Teradata implementation places the optional sign outside the apostrophes.

### Syntax

```
INTERVAL [ sign ] 'string' YEAR TO MONTH
```

### Syntax Elements

#### *sign*

An optional minus sign to indicate a negative interval. The default is a positive interval.

Note that the sign must be outside the apostrophes that enclose *string*.

#### *string*

One to four digits representing the number of years, followed by a hyphen and two digits representing the number of months. Spaces and new line characters are not allowed between the apostrophes.

---

#### Note:

For the digits representing the number of years, only digits are parsed and converted to numeric. For example, '1.05' is treated as '105'.

---

### Examples: INTERVAL YEAR TO MONTH Literal

The following example adds an interval of two years and six months to the current system date. (For this example, assume the current system date is 1999-11-03.)

```
SELECT CURRENT_DATE + INTERVAL '2-06' YEAR TO MONTH;

( 2-06+Date)
-----
2002/05/03
```

In the following query, the decimal point is ignored and the result is an interval of 10 years and 10 months.

```
SELECT INTERVAL '1.0-10' YEAR TO MONTH;
```

## INTERVAL MONTH Literals

Declares an INTERVAL MONTH value in an expression.

Result type: [INTERVAL MONTH Data Type](#)

### ANSI Compliance

INTERVAL MONTH literals are partly ANSI SQL:2011 compliant.

The ANSI definition places the optional sign for the interval within the apostrophes; the Teradata implementation places the optional sign outside the apostrophes.

### Syntax

```
INTERVAL [ sign ] 'string' MONTH
```

### Syntax Elements

#### *sign*

An optional minus sign to indicate a negative interval. The default is a positive interval.

The sign must be outside the apostrophes enclosing *string*.

#### *string*

One to four digits representing months. Spaces and new line characters are not allowed between the apostrophes.

---

#### Note:

Only digits within the apostrophes are parsed and converted to numeric. For example, '1.05' is treated as '105'.

---

### Example: INTERVAL MONTH Literals

This example adds an interval of six months to the current system date. (For this example, assume the current system date is 1999-11-03.)

```
SELECT (INTERVAL '6' MONTH) + CURRENT_DATE;

( 6+Date)
```

```
-----
2000/05/03
```

## INTERVAL DAY Literals

Declares an INTERVAL DAY value in an expression.

Result type: [INTERVAL DAY Data Type](#)

### ANSI Compliance

INTERVAL DAY literals are partly ANSI SQL:2011 compliant.

The ANSI definition places the optional sign for the interval within the apostrophes; the Teradata implementation places the optional sign outside the apostrophes.

### Syntax

```
INTERVAL [ sign ] 'string' DAY
```

### Syntax Elements

#### *sign*

An optional minus sign to indicate a negative interval. The default is a positive interval.

Note that the sign must be outside the apostrophes that enclose *string*.

#### *string*

One to four digits representing the number of days. Spaces and new line characters are not allowed between the apostrophes.

---

#### Note:

Only digits within the apostrophes are parsed and converted to numeric. For example, '1.05' is treated as '105'.

---

### Example: INTERVAL DAY Literal

This example adds an interval of -30 days to the current system date. (For this example, assume the current system date is 1999-11-03).

```
SELECT INTERVAL - '30' DAY + CURRENT_DATE;

(-30+Date)
```

```
-----
1999/10/04
```

## INTERVAL DAY TO HOUR Literals

Declares an INTERVAL DAY TO HOUR value in an expression.

Result type: [INTERVAL DAY TO HOUR Data Type](#)

### ANSI Compliance

INTERVAL DAY TO HOUR literals are partly ANSI SQL:2011 compliant.

The ANSI definition places the optional sign within the apostrophes; the Teradata implementation places the optional sign outside the apostrophes.

### Syntax

```
INTERVAL [ sign ] 'string' DAY TO HOUR
```

### Syntax Elements

#### *sign*

An optional minus sign to indicate a negative interval. The default is a positive interval.

The sign must be outside the apostrophes that enclose *string*.

#### *string*

One to four digits representing the number of days followed by a pad character and two digits representing the number of hours. Besides the pad character, no other spaces or new line characters are allowed between the apostrophes.

---

#### Note:

For the digits representing the number of days, only digits are parsed and converted to numeric. For example, '1.05' is treated as '105'.

---

### Examples: INTERVAL DAY TO HOUR Literal

The following example adds an interval of 1 day and 12 hours to the current system timestamp.

```
SELECT INTERVAL '1 12' DAY TO HOUR + CURRENT_TIMESTAMP;

( 1 12+Current TimeStamp)
```

```
-----
1999-11-05 02:32:19.770000+00:00
```

In the following query, the decimal point is ignored and the result is an interval of 10 days and 10 hours.

```
SELECT INTERVAL '1.0 10' DAY TO HOUR;
```

## INTERVAL DAY TO MINUTE Literals

Declares an INTERVAL DAY TO MINUTE value in an expression.

Result type: [INTERVAL DAY TO MINUTE Data Type](#)

### ANSI Compliance

INTERVAL DAY TO MINUTE literals are partly ANSI SQL:2011 compliant.

The ANSI definition places the optional sign within the apostrophes; the Teradata implementation places the optional sign outside the apostrophes.

### Syntax

```
INTERVAL [ sign ] 'string' DAY TO MINUTE
```

### Syntax Elements

#### *sign*

An optional minus sign to indicate a negative interval. The default is a positive interval.

The sign must be outside the apostrophes that enclose *string*.

#### *string*

One to four digits representing the number of days followed by a pad character and two digits representing the number of hours, followed by a colon and two digits representing the number of minutes. Besides the pad character, no other spaces or new line characters are allowed between the apostrophes.

#### Note:

For the digits representing the number of days, only digits are parsed and converted to numeric. For example, '1.05' is treated as '105'.

### Examples: INTERVAL DAY TO MINUTE Literal

The following example adds a 30 day, 12 hour, and 30 minute interval to the current system timestamp.



```
SELECT INTERVAL '30 12:30' DAY TO MINUTE + CURRENT_TIMESTAMP;

      ( 30 12:30+Current TimeStamp)
-----
1999-12-04 03:14:26.330000+00:00
```

In the following query, the decimal point is ignored and the result is an interval of 10 days, 10 hours, and 30 minutes.

```
SELECT INTERVAL '1.0 10:30' DAY TO MINUTE;
```

## INTERVAL DAY TO SECOND Literals

Declares an INTERVAL DAY TO SECOND value in an expression.

Result type: [INTERVAL DAY TO SECOND Data Type](#)

### ANSI Compliance

INTERVAL DAY TO SECOND literals are partly ANSI SQL:2011 compliant.

In the ANSI definition, the optional *sign* is within the apostrophes; the Teradata implementation places the optional sign outside the apostrophes.

### Syntax

```
INTERVAL [ sign ] 'string' DAY TO SECOND
```

### Syntax Elements

#### *sign*

An optional minus sign to indicate a negative interval. The default is a positive interval.

The sign must be outside the apostrophes that enclose *string*.

#### *string*

One to four digits representing the number of days, followed by a pad character and two digits representing the number of hours, followed by a colon and two digits representing the number of minutes. This is followed by a colon and two digits representing the number of seconds, and optionally followed by a decimal point and 1 to 6 digits representing fractional seconds. The decimal point is required if the fractional seconds are included. Besides the pad character, no other spaces or new line characters are allowed between the apostrophes.

**Note:**

For the digits representing the number of days, only digits are parsed and converted to numeric. For example, '1.05' is treated as '105'.

**Examples: INTERVAL DAY TO SECOND Literal**

The following example adds an interval of 30 days, 12 hours, 30 minutes, and 30.5 seconds to the current system timestamp.

```
SELECT INTERVAL '30 12:30:30.5' DAY TO SECOND + CURRENT_TIMESTAMP;
```

The result looks like this:

```
( 30 12:30:30.5+Current TimeStamp)
-----
1999-12-04 03:19:27.780000+00:00
```

In the following query, the first decimal point is ignored and the result is an interval of 10 days, 10 hours, 30 minutes, and 30.5 seconds.

```
SELECT INTERVAL '1.0 10:30:30.5' DAY TO SECOND;
```

**INTERVAL HOUR Literals**

Declares an INTERVAL HOUR value in an expression.

Result type: [INTERVAL HOUR Data Type](#)

**ANSI Compliance**

INTERVAL HOUR literals are partly ANSI SQL:2011 compliant.

The ANSI definition places the optional sign within the apostrophes; the Teradata implementation places the optional sign outside the apostrophes.

**Syntax**

```
INTERVAL [ sign ] 'string' HOUR
```

**Syntax Elements*****sign***

An optional minus sign to indicate a negative interval. The default is a positive interval.

The sign must be outside the apostrophes that enclose *string*.

### ***string***

One to four digits representing the number of hours. Spaces and new line characters are not allowed between the apostrophes.

---

#### **Note:**

Only digits within the apostrophes are parsed and converted to numeric. For example, '1.05' is treated as '105'.

---

### **Example: INTERVAL HOUR Literal**

This example subtracts an interval of 2000 hours from the current system timestamp.

```
SELECT CURRENT_TIMESTAMP - INTERVAL '2000' HOUR;

      (Current TimeStamp(6)- 2000)
-----
2003-01-28 09:27:08.180000+00:00
```

## **INTERVAL HOUR TO MINUTE Literals**

Declares an INTERVAL HOUR TO MINUTE value in an expression.

Result type: [INTERVAL HOUR TO MINUTE Data Type](#)

### **ANSI Compliance**

INTERVAL HOUR TO MINUTE literals are partly ANSI SQL:2011 compliant.

The ANSI definition places the optional sign within the apostrophes; the Teradata implementation places the optional sign outside the apostrophes.

### **Syntax**

```
INTERVAL [ sign ] 'string' HOUR TO MINUTE
```

### **Syntax Elements**

#### ***sign***

An optional minus sign to indicate a negative interval. The default is a positive interval.

The sign must be outside the apostrophes that enclose *string*.

***string***

One to four digits representing the number of hours followed by a colon and two digits representing the number of minutes. Spaces and new line characters are not allowed between the apostrophes.

**Note:**

For the digits representing the number of hours, only digits are parsed and converted to numeric. For example, '1.05' is treated as '105'.

**Examples: INTERVAL HOUR TO MINUTE Literal**

The following example adds an interval of 12 hours and 37 minutes to the current system time.

```
SELECT INTERVAL '12:37' HOUR TO MINUTE + CURRENT_TIME;

( 12:37+Current Time)
-----
03:34:03+00:00
```

In the following query, the decimal point is ignored and the result is an interval of 10 hours and 30 minutes.

```
SELECT INTERVAL '1.0:30' HOUR TO MINUTE;
```

**INTERVAL HOUR TO SECOND Literals**

Declares an INTERVAL HOUR TO SECOND value in an expression.

Result type: [INTERVAL HOUR TO SECOND Data Type](#)

**ANSI Compliance**

INTERVAL HOUR TO SECOND literals are partly ANSI SQL:2011 compliant.

The ANSI definition places the optional sign within the apostrophes; the Teradata implementation places the optional sign outside the apostrophes.

**Syntax**

```
INTERVAL [ sign ] 'string' HOUR TO SECOND
```

**Syntax Elements*****sign***

An optional minus sign to indicate a negative interval. The default is positive.

The sign must be outside the apostrophes that enclose *string*.

### ***string***

One to four digits representing the number of hours followed by a colon and two digits representing the number of minutes. This is followed by a colon and two digits representing the number of seconds, and optionally followed by a decimal point and 1 to 6 digits representing fractional seconds. The decimal point is required if the fractional seconds are included. Spaces and new line characters are not allowed between the apostrophes.

---

#### **Note:**

For the digits representing the number of hours, only digits are parsed and converted to numeric. For example, '1.05' is treated as '105'.

---

### **Examples: INTERVAL HOUR TO SECOND Literal**

The following example subtracts 12 hours, 37 minutes, and 12.123456 seconds from the current system time.

```
SELECT CURRENT_TIME - INTERVAL '12:37:12.123456' HOUR TO SECOND;

(Current Time(0)- 12:37:12.123456)
-----
                        05:07:11+00:00
```

In the following query, the first decimal point is ignored and the result is an interval of 10 hours, 30 minutes, and 12.123456 seconds.

```
SELECT INTERVAL '1.0:30:12.123456' HOUR TO SECOND;
```

## **INTERVAL MINUTE Literals**

Declares an INTERVAL MINUTE value in an expression.

Result type: [INTERVAL MINUTE Data Type](#)

### **ANSI Compliance**

INTERVAL MINUTE literals are partly ANSI SQL:2011 compliant.

The ANSI definition places the optional sign for the interval within the apostrophes; the Teradata implementation places the optional sign outside the apostrophes.

## Syntax

```
INTERVAL [ sign ] 'string' MINUTE
```

## Syntax Elements

### *sign*

An optional minus sign to indicate a negative interval. The default is a positive interval.

Note that the sign must be outside the apostrophes that enclose *string*.

### *string*

One to four digits representing the number of minutes. Spaces and new line characters are not allowed between the apostrophes.

---

#### Note:

Only digits within the apostrophes are parsed and converted to numeric. For example, '1.05' is treated as '105'.

---

## Example: INTERVAL MINUTE Literal

This example adds an interval of 600 minutes to the current system time.

```
SELECT INTERVAL '600' MINUTE + CURRENT_TIME;

( 600+Current Time)
-----
01:14:19+00:00
```

## INTERVAL MINUTE TO SECOND Literals

Declares an INTERVAL MINUTE TO SECOND value in an expression.

Result type: [INTERVAL MINUTE TO SECOND Data Type](#)

## ANSI Compliance

INTERVAL MINUTE TO SECOND literals are partly ANSI SQL:2011 compliant.

The ANSI definition places the optional sign within the apostrophes; the Teradata implementation places the optional sign outside the apostrophes.

## Syntax

```
INTERVAL [ sign ] 'string' MINUTE TO SECOND
```

## Syntax Elements

### *sign*

An optional minus sign to indicate a negative interval. The default is a positive interval.

The sign must be outside the apostrophes that enclose *string*.

### *string*

One to four digits representing the number of minutes followed by a colon and two digits representing the number of seconds, optionally followed by a decimal point and 1 to 6 digits representing fractional seconds. The decimal point is required if the fractional seconds are included. Spaces and new line characters are not allowed between the apostrophes.

---

#### Note:

For the digits representing the number of minutes, only digits are parsed and converted to numeric. For example, '1.05' is treated as '105'.

---

## Example: INTERVAL MINUTE TO SECOND Literal

The following example subtracts 9 minutes and 30 seconds from the current system timestamp.

```
SELECT CURRENT_TIMESTAMP - INTERVAL '9:30' MINUTE TO SECOND;

      (Current Timestamp(6)- 9:30)
-----
2004-04-01 17:40:19.610000+00:00
```

In the following query, the decimal point is ignored and the result is an interval of 10 minutes and 30 seconds.

```
SELECT INTERVAL '1.0:30' MINUTE TO SECOND;
```

## INTERVAL SECOND Literals

Declares an INTERVAL SECOND value in an expression.

Result type: [INTERVAL SECOND Data Type](#)

## ANSI Compliance

INTERVAL SECOND literals are partly ANSI SQL:2011 compliant.

The ANSI definition places the optional sign for the interval within the apostrophes; the Teradata implementation places the optional sign outside the apostrophes.

## Syntax

```
INTERVAL [ sign ] 'string' SECOND
```

## Syntax Elements

### *sign*

An optional minus sign to indicate a negative interval. The default is a positive interval.

Note that the sign must be outside the apostrophes that enclose *string*.

### *string*

One to four digits representing the number of seconds, optionally followed by a decimal point and 1 to 6 digits representing fractional seconds. The decimal point is required if the fractional seconds are included. Spaces and new line characters are not allowed between the apostrophes.

---

#### Note:

For the digits representing the number of seconds, only digits are parsed and converted to numeric. For example, '1.05' is treated as '105'.

---

## Example: INTERVAL SECOND Literals

The following example adds an interval of 0.000001 seconds to the current system time.

```
SELECT INTERVAL '0.000001' SECOND + CURRENT_TIME;

( 0.0+Current Time)
-----
15:21:23+00.00
```

In the following query, the first decimal point is ignored and the result is an interval of 10.000001 seconds.

```
SELECT INTERVAL '1.0.000001' SECOND;
```



## Period Literals

Specifies a constant value of a Period data type.

### Note:

Both temporal and nontemporal tables support Period data types and literals. However, Period data types and literals are not ANSI standard SQL; therefore, you can use them to define Teradata temporal tables, but not ANSI temporal tables. For more information about Teradata temporal tables, see *Teradata Vantage™ - Temporal Table Support*, B035-1182. For more information about ANSI temporal tables, see *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186.

### Syntax

```
PERIOD '('
  beginning_bound { - | , }
  { ending_bound | UNTIL_CHANGED | UNTIL_CLOSED }
  )'
```

### Syntax Elements

#### *beginning\_bound, ending\_bound*

Date, time, or timestamp values in the same format as Date, Time, or Timestamp literals.

A comma (,) separator between *beginning\_bound* and *ending\_bound* can include optional spaces on either side. A hyphen (-) separator must have a space before and after.

Date values have the following format:

YYYY-MM-DD

Time values have the following formats:

- hh:mi:ss
- hh:mi:sssignhh:mi
- hh:mi:ss.ssssss
- hh:mi:ss.ssssssignhh:mi

Timestamp values have the following formats:

- YYYY-MM-DD hh:mi:ss
- YYYY-MM-DD hh:mi:sssignhh:mi
- YYYY-MM-DD hh:mi:ss.ssssss
- YYYY-MM-DD hh:mi:ss.ssssssignhh:mi

For more information, see [Date Literals](#), [Time Literals](#), and [Timestamp Literals](#).

**UNTIL\_CHANGED**

The ending bound has a value of forever, or until it is changed. UNTIL\_CHANGED may only be specified when *beginning\_bound* is a Date or Timestamp value.

If *beginning\_bound* is a date value, UNTIL\_CHANGED specifies a value of DATE '9999-12-31'. If *beginning\_bound* is a timestamp value, UNTIL\_CHANGED specifies a value of TIMESTAMP '9999-12-31 23:59:59.999999+00:00', with precision truncated to the precision of *beginning\_bound* and the time zone omitted if *beginning\_bound* has no time zone.

**UNTIL\_CLOSED**

An ending bound for the Period value of a transaction-time column of a temporal table that indicates that the row is an open row. UNTIL\_CLOSED has a value of TIMESTAMP '9999-12-31 23:59:59.999999+00:00'.

For more information about Teradata temporal tables, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

**Data Types**

IF the format of the Period literal is ...	THEN the data type is ...
hh:mi:ss	PERIOD(TIME(0)).
hh:mi:sssignhh:mi	PERIOD(TIME(0) WITH TIME ZONE).
hh:mi:ss.ssssss	PERIOD(TIME( <i>n</i> )), where <i>n</i> is the maximum number of fractional seconds digits in the beginning and ending bound values, or, if the ending bound value is UNTIL_CHANGED, the number of fractional seconds digits in the beginning bound value.
hh:mi:ss.ssssssignhh:mi	PERIOD(TIME( <i>n</i> ) WITH TIME ZONE), where <i>n</i> is the maximum number of fractional seconds digits in the beginning and ending bound values, or, if the ending bound value is UNTIL_CHANGED, the number of fractional seconds digits in the beginning bound value.
YYYY-MM-DD	PERIOD(DATE).
YYYY-MM-DD hh:mi:ss	PERIOD(TIMESTAMP(0)).
YYYY-MM-DD hh:mi:sssignhh:mi	PERIOD(TIMESTAMP(0) WITH TIME ZONE).
YYYY-MM-DD hh:mi:ss.ssssss	PERIOD(TIMESTAMP( <i>n</i> )), where <i>n</i> is the maximum number of fractional seconds digits in the beginning and ending bound values, or, if the ending bound value is UNTIL_CHANGED, the number of fractional seconds digits in the beginning bound value.
YYYY-MM-DD hh:mi:ss.ssssssignhh:mi	PERIOD(TIMESTAMP( <i>n</i> ) WITH TIME ZONE), where <i>n</i> is the maximum number of fractional seconds digits in the beginning and ending bound values, or, if the ending bound value is UNTIL_CHANGED, the number of fractional seconds digits in the beginning bound value.

## Element Types

The element type of a Period literal is derived from the format of the DateTime values specified in the string literal.

IF...	THEN the element type of the literal is...
the beginning bound value only has a date value	DATE.
the beginning bound value only has a time value and a time zone interval is not specified after either the beginning or ending bound value	TIME(n).
the beginning bound value only has a time value and a time zone interval is specified after the beginning or ending bound value	TIME(n) WITH TIME ZONE.
the beginning bound value has both a date value and time value and a time zone interval is not specified after either the beginning or ending bound value	TIMESTAMP(n).
the beginning bound value contains both a date value and time value and a time zone interval is specified after the beginning or ending bound value	TIMESTAMP(n) WITH TIME ZONE.

## Usage Notes

A Period literal starts with keyword PERIOD, followed by a string literal that indicates the start (beginning bound value) and end (ending bound value) of the Period. Some of the syntactic elements and their usage that are not obvious are described below.

IF...	THEN...
the years, months, days, hours, minutes, seconds, and time zone interval do not conform to the rules of a DateTime literal with respect to the number of digits allowed and the valid range	a syntax error is reported.
the ending bound value is not UNTIL_CHANGED	the following must be true: <ul style="list-style-type: none"> <li>The beginning and ending bound values must both have a date value or both not have a date value.</li> <li>The beginning and ending bound values must both have a time value or both bound values must not have a time value. Otherwise, a syntax error is reported.</li> </ul>
the ending bound value is UNTIL_CHANGED and the beginning bound is a Timestamp value	the ending bound value must not be followed by a time zone interval; otherwise, a syntax error is reported.
the ending bound value is UNTIL_CLOSED (Teradata temporal tables only)	the following must be true: <ul style="list-style-type: none"> <li>The data type of the beginning bound value must be comparable with TIMESTAMP(6) WITH TIME ZONE.</li> </ul>

IF...	THEN...
	<ul style="list-style-type: none"> <li>The literal is only valid in an assignment operation where the target column to which the result is assigned is a transaction-time column.</li> <li>Because the only way to set the value of a transaction-time column is by using nontemporal DML, the literal is only valid in a nontemporal DML statement.</li> </ul> <p>Performing nontemporal operations on temporal tables requires the NONTEMPORAL privilege. For more information see <i>Teradata Vantage™ - Temporal Table Support</i>, B035-1182.</p>
the beginning bound value has a time value and the ending bound value is not UNTIL_CHANGED	the precision of the literal, <i>n</i> , is the maximum of the number of fractional digits specified in the beginning bound value and in the ending bound value.
the beginning bound value has a time value and the ending bound value is UNTIL_CHANGED	the precision of the literal, <i>n</i> , is the number of fractional digits specified in the beginning bound value.
only one of them includes a time zone value	the time zone field of the other is set to the current session time zone displacement. If both include time zone values, the result bounds include the corresponding time zone value.
UNTIL_CHANGED is specified for the ending bound value and the element type of the literal is TIME( <i>n</i> ) [WITH TIME ZONE]	a syntax error is reported.
the element type of the literal is TIMESTAMP( <i>n</i> ) WITH TIME ZONE	<p>the result ending element is set to the maximum TIMESTAMP[<i>(n)</i>] WITH TIME ZONE value at UTC (that is, the time zone displacement for the ending bound is INTERVAL '00:00' HOUR TO MINUTE).</p> <p><b>Note:</b></p> <p>If the ending bound value is UNTIL_CHANGED, the result ending element is set to the maximum TIMESTAMP value.</p>
the element type of the literal is TIME or TIMESTAMP and the beginning or ending bound value contains leap seconds	the seconds portion gets adjusted to 59.999999 with the precision truncated to the result precision.
the element type of the literal is DATE	the result ending bound must be greater than the result beginning bound; otherwise, a syntax error is reported.
the element type of the literal is not DATE after adjusting both the beginning and ending bound values to UTC using the specified time zone interval or, if not specified, the current session time zone displacement	the result ending bound must be greater than the result beginning bound; otherwise, a syntax error is reported.
the beginning element of a period literal is specified as UNTIL_CHANGED or UNTIL_CLOSED	an error is reported.

## Restrictions

UDFs or external stored procedures that are written in Java do not support arguments or return values that have a Period data type.

A primary index column or partitioning column cannot be a column that has a Period data type.

## Example: PERIOD(DATE) Literal

The following INSERT statement uses a PERIOD(DATE) literal.

```
INSERT INTO Policy
  (Policy_ID, Customer_ID, Policy_Type, Policy_Details, Validity)
VALUES (497201, 304779902, 'AU', 'STD-CH-524-WXY-00',
       PERIOD '(2005-02-03, 2006-02-04)');
```

The following INSERT statement uses a PERIOD(DATE) literal and sets the ending bound to UNTIL\_CHANGED.

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

## Example: PERIOD(TIME[(n)] WITH TIME ZONE) Literal

In the following UPDATE statement, the PERIOD(TIME(0) WITH TIME ZONE) column for flight 243 from Los Angeles, CA to Orlando, FL is updated in the schedule table. The literal's element type is TIME(0) WITH TIME ZONE.

```
UPDATE schedule
  SET flight_period = PERIOD '(08:00:00-08:00 - 15:40:00-05:00)'
  WHERE flight_no = 243;
```

# Character String Literals

Declares a character string literal value in an expression.

## ANSI Compliance

Character literals are ANSI SQL:2011 compliant.

## Syntax

```
[ _character_set ] 'string' [...]
```

## Syntax Elements

### `_character_set`

```
{ _Latin | _Unicode | _KanjiSJIS | _Graphic }
```

The name of the character set to be associated with the string literal:

- `_Latin` indicates that each character in the string is within the repertoire of the LATIN character set.
- `_Unicode` indicates that each character in the string is within the repertoire of the UNICODE character set.
- `_KanjiSJIS` indicates that each character in the string is within the repertoire of the KANJISJIS character set
- `_Graphic` indicates that each character in the string is within the repertoire of the GRAPHIC character set

If `_character_set` is not specified, the default is `_Unicode`.

Do not use an identifier of `_Kanji1`. This will generate an error.

### *string*

The character string literal.

## Definition

Character literals consist of zero or more alphanumeric characters enclosed in apostrophes. The new line character and spaces are allowed between the apostrophes. The total length of a character literal is 0 to 31000 bytes.

A character literal can include pass through characters (PTCs) for use in sessions with Unicode Pass Through enabled. Note that character literals containing PTCs cannot be used in account strings, which are validated as object names, and object names cannot contain PTCs. For information about Unicode Pass Through and PTCs, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

A zero-length character literal is represented by two consecutive apostrophes ( '' ).

To include an apostrophe ( ' ) in a character literal, use two consecutive apostrophes ( '' ).

Multiple consecutive character literals are treated as if the strings are concatenated.

## Character Literal Data Type

The data type of character literals is `VARCHAR(n) CHARACTER SET UNICODE`, where `n` is the length of the literal.

For details on the `VARCHAR(n)` type, see [VARCHAR Data Type](#).

## Character Literal Identifiers and KANJI1 Data [Deprecated]

### NOTICE

KANJI1 support is deprecated. KANJI1 is not allowed as a default character set. The system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

The following character data type literal identifiers can be assigned to a KANJI1 character column.

- `_Latin 'string'`
- `_Unicode 'string'`
- `_Graphic 'string'`
- `_KanjiSJIS 'string'`

Conversion is based on the current session character set, which can be established using the BTEQ command `SET SESSION CHARSET`.

For information on the assignability of these literals, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Case Specification

The CASESPECIFIC attribute of character strings determines the rules for string comparisons. The default case specificity for a character string literal depends on the mode of the session parsing the string for execution.

In this mode ...	The default case specificity for strings is ...
ANSI	CASESPECIFIC
Teradata	NOT CASESPECIFIC The exception is character data of type CHARACTER or VARCHAR CHARACTER SET GRAPHIC, which is always CASESPECIFIC.

## Example: Simple Character Strings

The following are examples of valid character literals:

```
'Los Angeles'
''
```

## Example: Character Strings Containing an apostrophe

To enter an apostrophe in a character literal, double the embedded apostrophe.

```
'He said ''yes'' to her question'
```

### Example: Character Literals Composed of Segments

Multiple strings are treated as though they are concatenated into one character literal.

```
('AA' 'BB') is ('AA' || 'BB')
(_Unicode 'AA' _Latin 'BB') is (_Unicode 'AA' || _Latin 'BB')
```

### Example: Character Set Specification

To indicate that the characters in a literal are within the repertoire of the LATIN character set, use the `_Latin` identifier.

```
CREATE TABLE table1(Name CHAR(30));
INSERT INTO table1 (_Latin 'Sandoval');
```

Note that the `_Latin` identifier does not change the server character set of the character literal data type, which is always UNICODE.

```
SELECT TYPE (_Latin 'Sandoval');
```

returns:

```
Type('Sandoval')
-----
VARCHAR(8) CHARACTER SET UNICODE;
```

### Example: Converting the Server Character Set of a Character Literal

The data type and server character set of a character literal is always `VARCHAR(n) CHARACTER SET UNICODE`, where `n` is the length of the literal.

Certain situations may require you to convert a character literal from UNICODE to a different server character set.

For example, suppose you want to compare the hexadecimal representation of two character strings. To get the hexadecimal representation of a character string, you can use `CHAR2HEXINT`.

Suppose one of the character strings you want to compare is the character literal `'a'`. The following statement returns the hexadecimal representation of `'a'`:

```
SELECT CHAR2HEXINT('a');
```

The result is `'0061'`, the hexadecimal representation of `'a'` in the UNICODE server character set, which uses two bytes per character.



Suppose the other character string you want to compare is in a column defined as CHAR(1) CHARACTER SET LATIN:

```
CREATE TABLE table1
  (column1 CHAR(1) CHARACTER SET LATIN
   ,column2 INTEGER);
INSERT INTO table1 ('a', 1001);
```

The following statement returns the hexadecimal representation of the string in column1:

```
SELECT CHAR2HEXINT(column1);
```

The result is '61', the hexadecimal representation of 'a' in the LATIN server character set, which uses one byte per character.

The following query compares the two strings:

```
SELECT column2 FROM table1
WHERE CHAR2HEXINT('a') = CHAR2HEXINT(column1);
```

The result is an empty set, because the result of CHAR2HEXINT('a') is '0061' and the result of CHAR2HEXINT(column1) is '61'.

For situations such as this, you can use TRANSLATE to convert the server character set of one of the character strings to match the server character set of the other character string. For example:

```
SELECT column2 FROM table1
WHERE CHAR2HEXINT( TRANSLATE('a' USING UNICODE_TO_LATIN) ) =
CHAR2HEXINT(column1);
```

returns:

```
column2
-----
      1001
```

## Example: Using Pass Through Character Literals

This example shows how you can use pass through character (PTC) literals in a session with Unicode Pass Through enabled. The PTC literals are emoji characters, and the session character set is UTF16.

To run these examples, replace the emoji graphics with the corresponding Unicode code points (such as U+1F601) inserted using your client application.

```

/*****
/* Enable Unicode Pass Through
*****/

```

```
SET SESSION CHARACTER SET UNICODE PASS THROUGH ON;
```

```

/*****
/* CREATE TABLE
*****/

```

```

CREATE TABLE upt_tbl0
(
  col1    INTEGER,
  col2    VARCHAR(6) CHARACTER SET UNICODE
) UNIQUE PRIMARY INDEX (col1);

```

```

/*****
/* Insert Pass Through Characters into the table
*****/

```

```
/* U+1F601 = GRINNING FACE WITH SMILING EYES */
```

```
INSERT INTO upt_tbl0 VALUES (1, '😄');
```

```
/* U+1F602 = FACE WITH TEARS OF JOY */
```

```
INSERT INTO upt_tbl0 VALUES (2, '😂');
```

```
/* U+1F603 = SMILING FACE WITH OPEN MOUTH */
```

```
INSERT INTO upt_tbl0 VALUES (3, '😁');
```

```
/* U+1F604 = SMILING FACE WITH OPEN MOUTH AND SMILING EYES */
```

```
INSERT INTO upt_tbl0 VALUES (4, '😆');
```

```
/* U+1F605 = SMILING FACE WITH OPEN MOUTH AND COLD SWEAT */
```

```
INSERT INTO upt_tbl0 VALUES (5, '😓');
```

```

/*****
/* DISPLAY ROWS
*****/

```

```
SELECT '😄';
```

Result:

• 🧐 •

-----

🧐

```
SELECT * FROM upt_tb10 WHERE col2 = '🧐';
```

Result:

col1 col2

-----

2 🧐

```
SELECT * FROM upt_tb10 ORDER BY col2;
```

Result:

col1 col2

-----

1 🧐

2 🧐

3 😊

4 😊

5 😊

## Unicode Delimited Character Literals

Declares a Unicode delimited character literal value in an expression.

The data type of Unicode delimited character literals is `VARCHAR(n) CHARACTER SET UNICODE`, where *n* is the resolved length of the literal in Unicode characters.

For details on the `VARCHAR(n)` type, see [VARCHAR Data Type](#).

### ANSI Compliance

Unicode delimited character literals are partially ANSI SQL:2011 compliant. The ANSI SQL:2011 standard does not require the `UESCAPE` clause and allows more possibilities for the *Unicode\_esc\_char*.

## Syntax

```
[ _Latin | _Unicode | _KanjiSJIS | _Graphic ] U& 'Unicode_string_body'
[ 'Unicode_string_body' [...] ] UESCAPE 'Unicode_esc_char'
```

## Syntax Elements

### 'Unicode\_string\_body'

A string of zero or more characters, enclosed in apostrophes, consisting of a combination of the following:

- Any character other than an apostrophe ( ' ) or the Unicode escape character
- Two consecutive apostrophes  
Use two consecutive apostrophes to include an apostrophe character in the Unicode delimited character literal.
- Two consecutive Unicode escape characters  
Use two consecutive escape characters to include the escape character in the delimited character literal.
- One Unicode escape character followed by 4 hexadecimal digits, where a hexadecimal digit is a character from 0 to 9, a to f, or A to F  
The 4 hexadecimal digits represent a Unicode BMP (Basic Multilingual Plane) character. Leading zeroes are required.
- One Unicode escape character followed by a plus sign (+) followed by 6 hexadecimal digits, where a hexadecimal digit is a character from 0 to 9, a to f, or A to F  
The 6 hexadecimal digits represent a Unicode BMP character or a supplementary character. Leading zeroes are required.

The Unicode escape character is specified by the UESCAPE clause.

If 'Unicode\_string\_body' immediately follows the U& key letters, no pad characters can appear before the first apostrophe.

Use successive occurrences of 'Unicode\_string\_body' to concatenate multiple strings, separated by a SPACE (U+0020).

A Unicode delimited character literal can consist of a maximum of 31000 *Unicode\_string\_body* characters.

### Unicode\_esc\_char

A single character from the session character set to use as the Unicode escape character in the delimited character literal.

The character must be within the printable ASCII range of Unicode characters (U+0021 through U+007E), with the following exceptions:

- SPACE (U+0020)
- QUOTATION MARK (U+0022)
- APOSTROPHE (U+0027)
- PLUS SIGN (U+002B)
- hexadecimal digit (0-9, a-f, or A-F):
  - U+0030 to U+0039
  - U+0041 to U+0046
  - U+0061 to U+0066

You can also specify the YEN SIGN (U+00A5) and WON SIGN (U+20A9) as the Unicode escape character.

## Usage Notes

A Unicode delimited character literal is useful for inserting a character string containing characters that cannot generally be entered directly on the terminal keyboard or is not available in the current session character set.

In a session where Unicode Pass Through is enabled, noncharacters in a Unicode delimited character literal are converted to an FFFD replacement character. In sessions where Unicode Pass Through is not enabled, an error is returned. Noncharacters are Unicode code points that are permanently reserved for internal use.

Examples:

```
SELECT U&'#FFFE' UESCAPE '#';
SELECT U&'#+01FFFF' UESCAPE '#';
```

Surrogate pairs and inappropriate surrogates in Unicode delimited character literals result in an error whether Unicode Pass Through is enabled or not.

Examples:

```
SELECT U&'#D800#DC00' UESCAPE '#';
SELECT U&'#+00D800#+00DC00' UESCAPE '#';
SELECT U&'#+00D800#+00D801' UESCAPE '#';
SELECT U&'#+00D800' UESCAPE '#';
```

Unicode supplementary characters are allowed with any character set when Unicode Pass Through is enabled:

```
.set session charset 'ascii'
set session character set unicode pass through on;
sel u&'#+010000' uescape '#';
```

Unicode supplementary characters are not allowed with any character set when Unicode Pass Through is disabled. The following SELECT statement will return an error:

```
.set session charset 'utf8'
set session character set unicode pass through off;
sel u&'#+010000' uescape '#';
```

For more information about Unicode Pass Through, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

If your application is intended to be ANSI-compliant and portable, you can replace existing hexadecimal character literals of the form '*hexadecimal digits*'XC with Unicode character literals.

### Examples: Unicode Delimited Character Literal

Consider this table:

```
CREATE TABLE TextTable
  (IDNum INTEGER
   ,Ustring VARCHAR(10) CHARACTER SET UNICODE);
```

This statement uses a Unicode delimited character literal to insert the character string '資料倉儲' into the Ustring column, using the number sign (#) as the Unicode escape character:

```
INSERT TextTable (10, _Unicode U&'#8CC7#6599#5009#5132' UESCAPE '#');
```

This statement concatenates two strings to insert '855-34-9729' into the Ustring column:

```
INSERT TextTable (11, U&'855-34-' '9729' UESCAPE '%');
```

This statement selects all rows from the TextTable table where the string in the Ustring column is an empty string:

```
SELECT * FROM TextTable WHERE Ustring = U&' ' UESCAPE '&;
```

### Examples: Literals Containing Unicode BMP and Supplementary Characters

You can specify Unicode BMP characters using either the 4 or 6 digit syntax. Note that leading zeroes are required.

```
SELECT U&'#0041' UESCAPE '#'
SELECT U&'#+000041' UESCAPE '#'
```

You can specify Unicode supplementary characters using the 6 digit syntax. Note that leading zeroes are required.

```
SELECT U&'#+020000' UESCAPE '#'
SELECT U&'#+010000' UESCAPE '#'
```

**Example: Error: Literal Containing Surrogate Code Points**

The character LINEAR B SYLLABLE B008 A (U+010000) can be represented as follows:

```
SELECT U&'#+010000' UESCAPE '#'
```

This character requires two 16 bit surrogate code points in internal UNICODE format (which is based on UTF-16). The query `SELECT CHAR2HEXINT(U&'#+010000' UESCAPE '#');` produces the following string:

```
'D800DC00'
```

However, note that the following query returns an error because surrogate code points are not allowed in a Unicode delimited character literal:

```
SELECT CHAR2HEXINT(U&'#D800#DC00' UESCAPE '#');
```

## Hexadecimal Character Literals

Declares a hexadecimal character literal value.

Hexadecimal literals consist of 0 to 31000 hexadecimal digits delimited by a matching pair of apostrophes, where a hexadecimal digit is a character from 0 to 9, a to f, or A to F.

**ANSI Compliance**

Hexadecimal character literals are Teradata extensions to the ANSI SQL:2011 standard.

**Syntax**

```
[ _Latin | _Unicode | _KanjiSJIS | _Graphic ] 'hexadecimal digits' { XC | XCV  
| XCF }
```

**Syntax Elements*****hexadecimal digits***

A string of hexadecimal digits, where a hexadecimal digit is a character from 0 to 9, a to f, or A to F.

**XC****XCV**

The hexadecimal literal has data type VARCHAR (default).

**XCF**

The hexadecimal literal has data type CHAR.

## Usage Notes

A hexadecimal character literal is useful for inserting character strings, such as TAB or BACKSPACE, that cannot generally be entered directly on the terminal keyboard.

The following apply to hexadecimal literals:

- Hexadecimal character literals are represented by an even number of hexadecimal digits. Hexadecimal literals are extended on the right with zeros when semantically required. For example:

```
_Latin 'C1C'XC = 'C1C0'XC
```

- As with other character literals, hexadecimal character literals are translated to UNICODE internal form, unless the character set is KANJI1, and then the hexadecimal literal is kept in KANJI1 internal form.
- When a hexadecimal character string is interpreted as \_Unicode either explicitly or because it was the default character set, the form of the data is interpreted as UTF-16. Therefore properly paired surrogates, such as \_Unicode 'D800DC00'XC, are treated as supplementary code points and can be used in sessions with Unicode Pass Through enabled. For information about Unicode Pass Through, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.
- Multiple consecutive character literals (character literals, hexadecimal character literals, and Unicode delimited character literals) are treated as if the strings are concatenated.

## Example: Hexadecimal Character Literal

Assume the user default character set is Latin. If the terminal keyboard does not include the brace characters {}, use the following hexadecimal literal to represent "{12}":

```
'7B31327D'XC
```

where 7B represents the brace "{" and 7D represents the brace "}".

## Graphic Literals

Declares a graphic character string literal in an expression.

### ANSI Compliance

Graphic literals are Teradata extensions to the ANSI SQL:2011 standard.

### Syntax

```
G '<string>'
```

### Syntax Elements

,

String delimiter for a graphic character string.



Each apostrophe is a single byte character.

### **<string>**

A string of valid KanjiEBCDIC multibyte characters surrounded by Shift-Out and Shift-In characters.

## **Usage Notes**

- On a *Japanese character site*, the G'*string*' syntax for graphic string literals can only be used when the KanjiEBCDIC character set is in effect for the session. It is equivalent to `_Graphic 'string'`.
- The `_Graphic 'string'` form can be used from any session character set, and is the recommended form for specifying literals required to be within the graphic repertoire.
- For other character sets, you can use the hexadecimal representation of the graphic string (see [Hexadecimal Byte Literals](#)).
- The literal must be enclosed within Shift-Out/Shift-In characters, as normal KanjiEBCDIC encoding requires.
- The maximum length is 15500 logical (double byte) characters.

Note that the Shift-Out/Shift-In characters are *not* included in the byte count and are not stored as part of the graphic literal.

- The contents must be valid graphic data (multibyte characters that are printable under the KanjiEBCDIC character set in effect for the current session).
- The internal representation must be an even number of bytes.
- Character string literals, including graphic string literals, can occur in the following places:
  - SQL query text
  - View and macro definition text
  - CHECK constraint text

In all of these instances, execution of queries using these literals involves parsing the literals and assigning case specific attributes to them at that time.

IF the session is in this mode ...	THEN the default case specificity of character string literals is ...
ANSI	CASESPECIFIC.
Teradata	NOT CASESPECIFIC. The exception is character data of type CHARACTER or VARCHAR CHARACTER SET GRAPHIC, which is always CASESPECIFIC.

This means that for views with WHERE constraints and for CHECK, if character string literals are included, they compare differently when executed by users with sessions in ANSI and Teradata mode.

To override the default case specificity attribute, apply the CASESPECIFIC or NOT CASESPECIFIC phrase to the character string literal.

You should modify *existing* views having WHERE constraints to explicitly control case specificity for comparisons.

Note that the collation in effect for the session can also cause character string literals to compare differently.

### Related Information

See ALTER TABLE and CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for explanations on the use of CHECK.

Also see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

You should explicitly control case specificity in character string comparisons. For information about how to explicitly control case specificity, see [Character and CLOB Data Types](#) of this volume.

# Numeric Data Types

This section describes the numeric data types.

Each numeric type can be combined with a CHECK constraint when the data type is defined or modified.

The section also describes several usage considerations for the system default DECIMAL size definition.

With respect to FLOAT data types, this section covers some of the comparison and computation inaccuracies associated with floating point values.

## BYTEINT Data Type

Represents a signed binary integer value in the range -128 to 127.

### ANSI Compliance

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

### Syntax

```
BYTEINT [ attributes [...] ]
```

### Syntax Elements

#### *attributes*

Appropriate data type attributes, column storage attributes, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

### Usage Notes

- Storage  
BYTEINT is stored as one byte.
- External Representation of BYTEINT

The following table lists the client representations for the BYTEINT data type.

Determining the application definitions and client data types is the responsibility of the application programmer.

Client CPU Architecture	Client Representation
IBM mainframe	One byte 8-bit signed 2's complement binary number.
◦ UTS	One byte 8-bit signed 2's complement binary number.

Client CPU Architecture	Client Representation
<ul style="list-style-type: none"> <li>◦ RISC</li> <li>◦ Motorola 68000</li> <li>◦ WE 32000</li> <li>◦ Intel</li> </ul>	

- Format

BYTEINT format is described in [Data Type Default Formats](#).

### Example: BYTEINT Data Type

In the following table definition, column EdLev is a BYTEINT data type:

```
CREATE TABLE Education
  (Id CHAR(9)
  ,LastName CHAR(26)
  ,EdLev BYTEINT FORMAT 'Z9'
  CHECK (EdLev BETWEEN 0 AND 22) NOT NULL);
```

## SMALLINT Data Type

Represents a signed binary integer value in the range -32768 to 32767.

### Syntax

```
SMALLINT [ attributes [...] ]
```

### Syntax Elements

#### *attributes*

Appropriate data type attributes, column storage attributes, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

### Usage Notes

- Storage

SMALLINT values are stored as two bytes, with the least significant byte first.

- External Representation of SMALLINT

The following table lists client representations for the SMALLINT data type.

Determining the application definitions and client data types is the responsibility of the application programmer.

Client CPU Architecture	Client Representation
IBM mainframe	Two byte 16-bit signed 2's complement binary number, most significant byte first.
<ul style="list-style-type: none"> <li>◦ UTS</li> <li>◦ RISC</li> <li>◦ Motorola 68000</li> <li>◦ WE 32000</li> </ul>	Two byte 16-bit signed 2's complement binary number, most significant byte first.
Intel	Two byte 16-bit signed 2's complement binary number, least significant byte first.

- Format

For information on the default display format, see [Data Type Default Formats](#).

### Example: SMALLINT Data Type

In the following table definition, column DeptNo is a SMALLINT data type:

```
CREATE TABLE Departments
  (DeptNo SMALLINT FORMAT '999' BETWEEN 100 AND 900
  ,ManagerName CHAR(26)
  ,ManagerID CHAR(9));
```

## INTEGER Data Type

Represents a signed, binary integer value from -2,147,483,648 to 2,147,483,647.

### Syntax

```
{ INTEGER | INT } [ attributes [...] ]
```

### Syntax Elements

#### *attributes*

Appropriate data type attributes, column storage attributes, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

### Usage Notes

- Storage  
INTEGER values are stored as four bytes with the least significant byte first.
- External Representation of INTEGER

The following table lists the client representations for the Teradata INTEGER data type.

Determining the application definitions and client data types is the responsibility of the application programmer.

Client CPU Architecture	Client Representation
IBM mainframe	Four byte 32-bit signed 2's complement binary number, most significant byte first.
<ul style="list-style-type: none"> <li>◦ UTS</li> <li>◦ RISC</li> <li>◦ Motorola 68000</li> <li>◦ WE 32000</li> </ul>	Four byte 32-bit signed 2's complement binary number, most significant byte first.
Intel	Two byte 16-bit signed 2's complement binary number, least significant byte first.

- Format

For information on the default display format, see [Data Type Default Formats](#).

### Example: INTEGER Data Type

In the following table definition, column TelNo is an INTEGER data type:

```
CREATE TABLE Contact
(Id CHAR(9)
,LastName CHAR(26)
,TelNo INTEGER);
```

## BIGINT Data Type

Represents a signed, binary integer value from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

### Syntax

```
BIGINT [ attributes [...] ]
```

### Syntax Elements

#### *attributes*

Appropriate data type attributes, column storage attributes, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

- Storage

BIGINT values are stored as eight bytes with the least significant byte first.

- External Representation of BIGINT

The following table lists the client representations for the Teradata BIGINT data type.

Determining the application definitions and client data types is the responsibility of the application programmer.

Client CPU Architecture	Client Representation
IBM mainframe	Eight byte 64-bit signed 2's complement binary number, most significant byte first.
<ul style="list-style-type: none"> <li>UTS</li> <li>RISC</li> <li>Motorola 68000</li> <li>WE 32000</li> </ul>	Eight byte 64-bit signed 2's complement binary number, most significant byte first.
Intel	Eight byte 64-bit signed 2's complement binary number, least significant byte first.

- Format

For information on the default display format, see [Data Type Default Formats](#).

### Example: BIGINT Data Type

In the following table definition, column Total is a BIGINT data type:

```
CREATE TABLE RelevantNumbers
  (Id CHAR(9)
  ,LastSummary INTEGER
  ,Total BIGINT);
```

## DECIMAL/NUMERIC Data Types

Represents a decimal number of  $n$  digits, with  $m$  of those  $n$  digits to the right of the decimal point.

### ANSI Compliance

NUMERIC is in the ANSI SQL:2011 standard. DECIMAL is a Teradata synonym for NUMERIC.

### Syntax

```
{ DECIMAL | DEC | NUMERIC } [ (  $n$  [,  $m$  ] ) ] [ attributes [...] ]
```

## Syntax Elements

***n***

The precision (the maximum number of digits that can be stored).

The range is from 1 through 38.

***m***

The scale (the number of fractional digits).

The range is from 0 through *n*.

When values are not specified for *n*, *m*, then the default is DECIMAL(5, 0).

When a value is not specified for *m*, then the default is DECIMAL(*n*, 0).

***attributes***

Appropriate data type attributes, column storage attributes, or column constraint attributes.

See [Core Data Type Attributes](#) and [Constraint Attributes](#) for specific information.

## Usage Notes

### Storage

Decimal numbers are scaled by the power of ten equal to the number of fractional digits. The number is stored as a two's complement binary number in 1, 2, 4, 8, or 16 bytes. The number of bytes used for a decimal value depends on the total number of digits in that value.

The following list shows the number of bytes used to store decimal values.

Number of Digits	Number of Bytes
1 to 2	1
3 to 4	2
5 to 9	4
10 to 18	8
19 to 38	16

For example, DECIMAL(3,2) requires 2 bytes and the value -2 would be represented as -200 in two's complement binary form.

The maximum value for DECIMAL(*n*,*m*) is a value consisting of *n* 9's with the decimal point *m* digits from the right. The minimum value would be the negative of the maximum value.

Examples:



<i>n</i>	<i>m</i>	maximum	minimum
3	2	9.99	-9.99
4	4	.9999	-.9999
9	1	99999999.9	-99999999.9

## External Representation of DECIMAL/NUMERIC Numbers

The following table lists the client representations for the DECIMAL/NUMERIC data type.

Determining the application definitions and client data types is the responsibility of the application programmer.

Client CPU Architecture	Client Representation
<ul style="list-style-type: none"> <li>• RISC</li> <li>• Motorola 68000</li> <li>• WE 32000</li> </ul>	<p>Signed two's complement binary number, most significant byte first.</p> <p>For these values of <i>n</i>:</p> <ul style="list-style-type: none"> <li>• 1 or 2, the number is 8-bit</li> <li>• 3 or 4, the number is 16-bit</li> <li>• 5 to 9, the number is 32-bit</li> <li>• 10 to 18, the number is 64-bit</li> <li>• 19 to 38, the number is 128-bit</li> </ul>
Intel	<p>Signed two's complement binary number, least significant byte first.</p> <p>For these values of <i>n</i>:</p> <ul style="list-style-type: none"> <li>• 1 or 2, the number is 8-bit</li> <li>• 3 or 4, the number is 16-bit</li> <li>• 5 to 9, the number is 32-bit</li> <li>• 10 to 18, the number is 64-bit</li> <li>• 19 to 38, the number is 128-bit</li> </ul>
<ul style="list-style-type: none"> <li>• IBM mainframe</li> <li>• UTS</li> </ul>	<p>Twenty bytes (maximum) <i>n</i>-digit (where <i>n</i> represents the precision of the number and must be less than 38), signed, packed decimal numbers.</p> <p>The rightmost nibble represents the sign.</p> <p>The + sign has the following hexadecimal representation:</p> <ul style="list-style-type: none"> <li>• X'A'</li> <li>• X'C'</li> <li>• X'E'</li> <li>• X'F'</li> </ul> <p>The - sign has the following hexadecimal representation:</p> <ul style="list-style-type: none"> <li>• X'B'</li> <li>• X'D'</li> </ul> <p>The remaining nibbles represent the digits X'0' - X'9', left-padded with 0 digits when <i>n</i> is even, giving a total of <math>(n+2)/2</math> bytes, or 20 bytes maximum.</p>

## Application Requirements for the Size of DECIMAL Types

Some applications require DECIMAL types to have 18 or fewer digits or possibly 15 or fewer digits.

Applications with such requirements may need to access DECIMAL columns that have more digits or use expressions that may produce DECIMAL results with more digits. To help with DECIMAL type size requirements, you can use the following:

- CAST function to convert to a DECIMAL type of 18 or fewer digits or 15 or fewer digits
- MaxDecimal field in DBS Control to set the maximum number of digits in a DECIMAL result for an expression containing DECIMAL arguments
- Max-decimal-returned field in the DBCAREA data area to set the maximum precision for a DECIMAL data type result column for CLIV2 for mainframe-attached systems
- Maximum Decimal Precision field in the DBCAREA data area to set the maximum precision for a DECIMAL data type result column for CLIV2 for workstation-attached systems
- DECIMALDIGITS BTEQ command to set the maximum precision for decimal values associated with subsequent SQL requests in nonfield mode. The maximum decimal digits to return applies to all of the record modes (record, indicator, and multipart indicator), but does not apply to field mode. In field mode, you must perform a CAST or use the FORMAT phrase.

## Size of DECIMAL Expression Result Type

You can set the MaxDecimal field in DBS Control to control the maximum number of digits in a DECIMAL result for an expression containing DECIMAL arguments.

There are four valid values for the MaxDecimal field.

IF the value of MaxDecimal is ...	THEN the maximum number of digits for a DECIMAL result of an expression is ...
0	15, if the operands have 15 or less digits.
15	18, if one operand has between 16 and 18 digits and the other operand has 18 or less digits.
18	18, if operands have 18 or less digits.
38	38

The number of digits in a DECIMAL result for an expression containing DECIMAL arguments depends on the value of the MaxDecimal in DBS Control and the number of digits in the DECIMAL arguments.

For example, suppose the value of MaxDecimal is 15. An arithmetic expression that adds a DECIMAL(15) argument and a DECIMAL(15) argument results in a DECIMAL(15). An arithmetic expression that adds a DECIMAL(15) argument and a DECIMAL(18) argument results in a DECIMAL(18).

For more information on the number of digits in a DECIMAL result for an expression containing DECIMAL arguments, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Examples

**Example: DECIMAL Data Type**

Consider the number 256.78. Its type is DECIMAL (5,2) and its default format is expressed either as -(4).9(2) or the equivalent ----.99

The default DECIMAL or NUMERIC display formats are described in [Data Type Default Formats](#).

**Example: Defining a Table Column as DECIMAL Type**

In the following table definition, column Salary is assigned the type DECIMAL.

```
CREATE TABLE Salaries
  (Id CHAR(9)
  ,Salary DECIMAL(8,2) FORMAT 'ZZZ,ZZ9.99'
  CHECK (Salary BETWEEN 1.00 AND 999000.00) );
```

**Related Information**

FOR more information on ...	SEE ...
changing the value of MaxDecimal	<i>Teradata Vantage™ - Database Utilities</i> , B035-1102.
Max-decimal-returned in DBCAREA	<i>Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems</i> , B035-2417.
Maximum Decimal Precision in DBCAREA	<i>Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems</i> , B035-2418.
DECIMALDIGITS BTEQ command	<i>Basic Teradata® Query Reference</i> , B035-2414.
rounding DECIMAL types	<a href="#">Rounding</a> .
the result of expressions containing DECIMAL arguments	<i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145.
using CAST to convert to a DECIMAL type of 15 or fewer digits or 18 or fewer digits	

**FLOAT/REAL/DOUBLE PRECISION Data Types**

Represent values in sign/magnitude form ranging from  $2.226 \times 10^{-308}$  to  $1.797 \times 10^{308}$ .

**ANSI Compliance**

REAL and DOUBLE PRECISION are in the ANSI SQL:2011 standard. FLOAT is a Teradata synonym for REAL and DOUBLE PRECISION.

## Syntax

```
{ FLOAT | REAL | DOUBLE PRECISION } [ attributes [...] ]
```

## Syntax Elements

### *attributes*

Appropriate data type attributes, column storage attributes, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Floating Point Value Storage and Manipulation

Eight bytes are used to hold a floating point value.

Floating point values are stored and manipulated internally in IEEE floating point format. Floating point values are stored with the least significant byte first, with one bit for the mantissa sign, 11 bits for the exponent, and 52 bits for the mantissa. The mantissa sign is in the most significant bit position; the exponent sign is a part of the exponent field (excess-1024 notation, in which  $(1024 - \text{exponent}) = \text{sign}$ ).

Negative numbers differ from positive numbers of the same magnitude only in the sign bit.

Vantage supports normalized, but not non-normalized, client floating point values.

The range of IEEE floating point values may be wider than that supported by the client system. Therefore, values can be created and stored in Vantage that return error messages when converted for delivery to the client system.

### External Representation of FLOAT/REAL/DOUBLE PRECISION

The following table lists the client representations for the Teradata FLOAT/REAL/DOUBLE PRECISION data type.

Determining the application definitions and client data types is the responsibility of the application programmer.

Client CPU Architecture	Client Representation
IBM mainframe	Eight byte 64-bit (double precision) floating point number, most significant byte first, with the attributes as follows: <ul style="list-style-type: none"> <li>• 1 bit represents the sign of the fraction.</li> <li>• 7 bits represent the unsigned power of 16 exponent stored as actual plus X'40'.</li> <li>• 56 bits represent the unsigned fraction.</li> </ul>
UTS	Eight byte 64-bit (double precision) floating point number, most significant byte first, with the attributes as follows: <ul style="list-style-type: none"> <li>• 1 bit represents the sign of the fraction.</li> </ul>

Client CPU Architecture	Client Representation
	<ul style="list-style-type: none"> <li>• 7 bits represent the unsigned power of 16 exponent stored as actual plus X'40'.</li> <li>• 56 bits represent the unsigned fraction.</li> </ul>
<ul style="list-style-type: none"> <li>• RISC</li> <li>• Motorola 68000</li> <li>• WE 32000</li> </ul>	Eight byte 64-bit (double precision) floating point number, most significant byte first, with the attributes as follows: <ul style="list-style-type: none"> <li>• 1 bit represents the sign of the fraction.</li> <li>• 11 bits represent the unsigned power of 2 exponent stored as actual plus X'3FFH'.</li> <li>• 52 bits represent the unsigned fraction.</li> </ul>
Intel	Eight byte 64-bit (double precision) floating point number, least significant byte first, with the attributes as follows: <ul style="list-style-type: none"> <li>• 1 bit represents the sign of the fraction.</li> <li>• 11 bits represent the unsigned power of 2 exponent stored as actual plus X'3FFH'.</li> <li>• 52 bits represent the unsigned fraction.</li> </ul>

## Example: FLOAT Data Type

In the following table definition, column SalaryFactor is a FLOAT data type:

```
CREATE TABLE Salaries
  (Id CHAR(9)
  ,SalaryFactor FLOAT BETWEEN .1 AND 1E1 );
```

## Related Information

FOR information on ...	SEE ...
potential problems associated with floating point values in comparisons and computations	<a href="#">Operations on Floating Point Values.</a>
rounding and FLOAT/REAL/DOUBLE PRECISION types	<a href="#">Rounding.</a>

## NUMBER Data Type

Represents a numeric value with optional precision and scale limitations.

### ANSI Compliance

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

## Syntax

```
NUMBER [ ( { n | * } [, m ] ) ] [ attributes [...] ]
```

## Syntax Elements

*n*

The precision.

The range is from 1 to 38.

*m*

The scale. This indicates the maximum number of digits allowed to the right of the decimal point.

If \* is specified, the range of *m* is from 0 to 38.

If *n* is specified, the range of *m* is from 0 to *n*.

*attributes*

Appropriate data type attributes, column storage attributes, or column constraint attributes.

See [Core Data Type Attributes](#) and [Constraint Attributes](#) for specific information.

## Usage Notes

### General

The NUMBER data type provides the following benefits:

- Increased flexibility in defining numeric columns by providing the ability to increase the precision or scale of existing NUMBER columns in tables without modifying the data rows. However, some restrictions apply.
- Increased efficiency in storing numeric data because NUMBER is a variable-length data type that can vary from 0 to 18 bytes, depending on the value stored.
- Increased flexibility in computation compared with the DECIMAL data type because the result is not limited by the precision or scale of the input.
- Greater range than the DECIMAL data type.
- Greater accuracy than the FLOAT data type because NUMBER has greater guaranteed precision and NUMBER can represent common decimals exactly.
- Increased compatibility with other databases. Several database vendors include a similar NUMBER data type.

**Note:**

For many purposes, the NUMBER data type can provide exact results. However, it is a floating point type. Any database vendor supporting floating point types is subject to different answers based on the order of evaluation. There will be cases where Vantage will not produce the same results as other vendors or even the same answer on different executions of the same query.

**Storage**

NUMBER is stored as a variable-length value comprising an exponent and mantissa. The base of the exponent is 10; therefore, common decimal numbers can be represented exactly.

With one exception, the storage layout is as follows:

1 Byte	1 - 17 Bytes
Exponent	Mantissa

The exception is the value zero, which is indicated by a zero-length NUMBER field with no exponent or mantissa.

The following table summarizes the storage needs and range of NUMBER values.

Data Type	Database Storage Required (in bytes)	Range of Numbers
NUMBER( <i>n,m</i> )	0 to 18, based on the actual value stored.	Number of <i>n</i> total digits with <i>m</i> fractional digits.
NUMBER(*, <i>m</i> )	0 to 18, based on the actual value stored.	Number with up to <i>m</i> fractional digits.
NUMBER(*)	0 to 18, based on the actual value stored.	$\pm [1\text{E-}130 \text{ to } 9.99\dots9\text{E}125]$ , including 0

If a COMPRESS clause is specified, the value is stored in the compressed portion of the row and an extra byte is required to indicate the length. If no COMPRESS clause is present, the value is stored in the variable portion of the row, which requires two bytes. For every row that has a variable portion, two bytes are required.

**External Representation of NUMBER**

In Field mode, NUMBER data is returned based on the FORMAT phrase. For information about the default format for NUMBER and using the FORMAT phrase with NUMBER data, see [Data Type Default Formats](#) and [FORMAT Phrase and NUMERIC Formats](#).

For response modes other than Field mode, NUMBER data has the following representation for most values.

1 byte Length	2 bytes Scale	1 to 17 bytes two's complement representation of the unscaled value, in the client-appropriate endianness.
---------------	---------------	--

When NUMBER is zero or null, a length value of zero indicates no scale or unscaled value is required.

### Differences Between NUMBER and DECIMAL

NUMBER ( $n,m$ ) is similar to DECIMAL ( $n,m$ ) in functionality and can be used wherever DECIMAL is supported.

NUMBER differs from DECIMAL in the following ways:

DECIMAL	NUMBER
The data type is fixed-length with 1, 2, 4, 8 or 16 bytes, depending on the precision.	The data type is variable-length and varies from 0 to 18 bytes internally and from 1 to 20 bytes externally.
Numbers are scaled by the power of ten equal to the number of fractional digits. The number is stored as a two's complement binary number.	NUMBER is stored as a variable-length value comprising an exponent and mantissa.
You cannot change the precision or scale of an existing DECIMAL column without modifying data rows.	You can increase the precision or scale of an existing NUMBER column without modifying data rows. However, some restrictions apply. For more information, see ALTER TABLE in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
The intermediate results of DECIMAL arithmetic have a precision of 15, 18, or 38 depending on the arguments used in the arithmetic expression.	Calculations for the NUMBER data type are guaranteed to 38 digits of precision, but are often performed with 39 or 40 digits of precision. If precision and scale are not limited, NUMBER data will retain the full precision and scale of the calculations. Therefore, the arithmetic operation $1./3.$ returns 0. when performed with DECIMAL, but returns .333333333... when performed with NUMBER.
Rounding mode is controlled by the RoundHalfWayMagUp field in DBS Control. For details, see <a href="#">Rounding</a> .	Rounding mode is controlled by the RoundNumberAsDec field in DBS Control. For details, see <a href="#">Rounding</a> .
DECIMAL when specified without precision or scale defaults to DECIMAL(5,0).	NUMBER when specified without precision or scale defaults to NUMBER with the system limits for precision and scale.

### Differences Between NUMBER and FLOAT

Numeric values can be represented as:

- NUMBER(\*, $m$ )
- NUMBER(\*), which is equivalent to NUMBER

NUMBER can be used wherever FLOAT is supported. However, NUMBER differs from FLOAT in the following ways:



FLOAT	NUMBER
The data type is fixed-length and uses 8 bytes of storage.	The data type is variable-length and varies from 0 to 18 bytes internally and from 1 to 20 bytes externally.
FLOAT is stored using the IEEE-754 standard, which represents floating point numbers using base-2 and is generally known as binary float. This storage scheme cannot represent all decimal values exactly.	NUMBER is stored as a variable-length value comprising an exponent and mantissa. This storage scheme stores decimal values exactly because it uses a decimal base.
FLOAT can only provide accuracy of 15 decimal digits.	Calculations for the NUMBER data type are guaranteed to 38 digits of precision, but are often performed with 39 or 40 digits of precision. If precision and scale are not limited, NUMBER data will retain the full precision and scale of the calculations.

## Example: NUMBER Data Type

Consider the following table definition:

```
CREATE TABLE num_tab
(n1 NUMBER(*,3),
 n2 NUMBER,
 n3 NUMBER(*),
 n4 NUMBER(5,1),
 n5 NUMBER(3) );
```

The following table shows the result of inserting various values into a column defined as NUMBER (3).

If you insert this value...	the value of column n5 will be...
1.234	1
123.6789	124
1234.56	An error is returned since the inserted value exceeds the size of column n5.

The following table shows the result of inserting various values into a column defined as NUMBER (\*,3).

If you insert this value...	the value of column n1 will be...
1.234	1.234
1234.6789	1234.679

## Related Information

FOR more information on ...	SEE ...
rounding NUMBER types	<a href="#">Rounding</a> .
the result of expressions containing NUMBER arguments	<i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145.
the default format for NUMBER and using the FORMAT phrase with NUMBER data	<a href="#">Data Type Default Formats</a> and <a href="#">FORMAT Phrase and NUMERIC Formats</a> .

## Operations on Floating Point Values

### Inconsistencies When Used With Common C Library Functions

Use of floating point data in Common C library functions can be inconsistent between different compilers and supporting libraries. For example, trigonometric functions in the C runtime library may return slightly different values depending on the Linux version or distribution.

### Comparison and Computation Inaccuracies

Because floating point numbers are not stored as exact values, some inaccuracy is inherent and unavoidable when they are involved in comparisons and computations.

Here are some of the problems you might encounter:

- Identical computations in floating point arithmetic may produce slightly different results on different machines because internal precision differs from computer to computer, and from model to model in the same series of computer.
- Because floating point decimal values generally do not have exact binary representations, calculations involving floating point values can often produce results that are not what you might expect. For example, the common decimal number 0.1 is a repeating sequence in binary and does not have a precise binary representation. If you perform a calculation and then compare the results against some expected value, it is highly unlikely that you get exactly the result you intended.
- If you add or subtract floating point numbers that differ greatly in size, the contribution of the small numbers can effectively be lost. For example,  $1E20 + 1.0$  and  $1E20 - 1.0$  evaluate to  $1E20$ .
- Operations involving floating point numbers are not always associative due to approximation and rounding errors:  $((A + B) + C)$  is not always equal to  $(A + (B + C))$ .

Although not readily apparent, the non-associativity of floating point arithmetic can also affect aggregate operations: you can get different results each time you use an aggregate function on a given set of floating point data. When SQL Engine performs an aggregation, it accumulates individual terms from each AMP involved in the computation and evaluates the terms in order of arrival to produce the final result. Because the order of evaluation can produce slightly different results, and because the order in which individual AMPs finish their part of the work is unpredictable, the results of an aggregate function on the same data on the same system can vary.

- Conversion of DECIMAL and INTEGER values to FLOAT values might result in a loss of precision or produce a number that cannot be represented exactly.
- GROUP BY on a FLOAT type can produce inconsistent results.

If you need exact results, do not use floating point types.

## Example: Non-Associativity of Floating Point Arithmetic

Consider a table where the same values are inserted into two floating point columns, but in a different order:

```
CREATE TABLE t1 (i INTEGER, a FLOAT, b FLOAT);
INSERT t1 (1, 1000.55, 2000.7);
INSERT t1 (1, 2000.4, 2000.1);
INSERT t1 (1, 2000.1, 2000.4);
INSERT t1 (1, 2000.7, 1000.55);
```

The conditional expression in the following SELECT statement compares the sums of the values in the two columns:

```
SELECT i, SUM(a) as sum_a, SUM(b) as sum_b
FROM t1
GROUP BY i
HAVING sum_a <> sum_b;
```

Because the values that the two SUM calculations uses are the same, the obvious result is that no rows are returned. However, the result is:

i	sum_a	sum_b
1	7.00175000000000E 003	7.00175000000000E 003

What appears to be an invisible error has crept into the calculations. The following statement shows the error:

```
SELECT ABS(SUM(a) - SUM(b)) FROM t1;
```

Here is the result:

```
Abs((Sum(a)-Sum(b)))
-----
1.81898940354586E-012
```

## Example: Comparing Floating Point Values

Calculations involving floating point values often produce results that are not what you expect. If you perform a floating point calculation and then compare the results against some expected value, it is unlikely that you get the intended result. Consider the results of [Example: Non-Associativity of Floating Point Arithmetic](#).

Instead of comparing the results of a floating point calculation, make sure that the result is greater or less than what is needed, with a given error. Here is an example of how to rewrite the statement in [Example: Non-Associativity of Floating Point Arithmetic](#) and achieve the desired results:

```
SELECT i, SUM(a) as sum_a, SUM(b) as sum_b
FROM t1
GROUP BY i
HAVING ABS(sum_a - sum_b) > 1E-10;
```

## Rounding

### Rounding DECIMAL/NUMERIC Data Types

If a value being inserted does not fit into a DECIMAL or NUMERIC column, the value is rounded. SQL Engine rounds using the digit to the right of the *rounding digit*, the last digit that fits into the DECIMAL/NUMERIC field.

IF the value of the digit to the right of the rounding digit is ...	THEN the value of the rounding digit ...
< 5	does not change.
> 5	increases by one.

Additional considerations come into play when the value of the digit to the right of the rounding digit is exactly five. One consideration is the value of the DBS Control Record RoundHalfwayMagUp field.

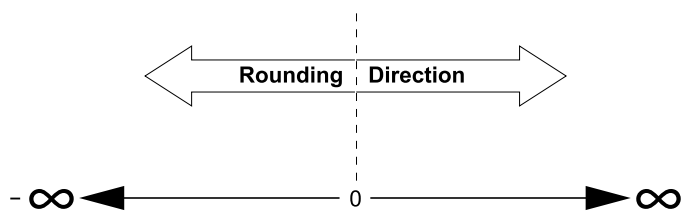
When the DBS Control Record RoundHalfwayMagUp field is set to FALSE, then rounding is performed as explained in the following table. This is the default.

IF the value of the digit to the right of the rounding digit is exactly 5 and ...	THEN ...
there are no trailing nonzero digits	<ul style="list-style-type: none"> <li>If the value of the rounding digit is odd, then the value of the rounding digit increases by one.</li> <li>If the value of the rounding digit is even, then the value of the rounding digit does not change.</li> </ul>
there are trailing nonzero digits	rounding behaves as if the value of the digit to the right of the rounding digit is greater than five.

The following table shows the results of inserting values into a DECIMAL(3,2) column.

WHEN the value of the INSERT is ...	THEN the value is rounded ...	Because the value of the digit to the right of the rounding digit is ...	AND the result is ...
.014	down	< 5	.01
.015	up	5 and last stored position is odd	.02
.0151	up	> 5	.02
.024	down	< 5	.02
.025	down	5 and last stored position is even.	.02
.0251	up	> 5	.03

When the DBS Control Record RoundHalfwayMagUp field is set to TRUE, then the magnitude of all halfway values, both negative and positive, is rounded up, away from zero (see the illustration).



While this is the most common rounding semantics for business applications, it is also nontrivially biased in the upward direction.

For more information about the DBS Control Record and the RoundHalfwayMagUp field, see *Teradata Vantage™ - Database Utilities*, B035-1102.

## Rounding FLOAT, REAL, DOUBLE PRECISION Data Types

Conversion to FLOAT/REAL/DOUBLE PRECISION rounds to the nearest value available. Neither decimal fractions nor numbers greater than 9,007,199,254,740,992 can be guaranteed to be represented exactly, so the nearest representable value is chosen. If there are two representable values that qualify as the nearest value, then the representation with a '0' in the least significant bit is chosen.

For example, 0.1, when stored in a FLOAT column, is rounded to a value slightly higher: 0.100000000000000055511151231257827021181583404541015625.

## Rounding NUMBER Data Type

When rounding a value to NUMBER type, the nearest NUMBER to the value is chosen, taking into account the NUMBER scale. If there are two NUMBER values the same distance from the value, the rounding rule is determined by the DBS Control field RoundNumberAsDec. By default, the RoundNumberAsDec field is FALSE, which indicates that the digit will be rounded up, away from zero. If you want NUMBER rounding to

follow DECIMAL rounding behavior, set the RoundNumberAsDec field to TRUE, in which case the rounding rule will be determined by the DBS Control field RoundHalfwayMagUp.

For more information about the DBS Control Record, and the RoundNumberAsDec and RoundHalfwayMagUp fields, see *Teradata Vantage™ - Database Utilities*, B035-1102.

# Character and CLOB Data Types

This section describes the character and CHARACTER LARGE OBJECT (CLOB) data types.

## Character Data

In general, CHARACTER, VARCHAR, and CLOB data types represent character data.

Character data is automatically translated between the client and the database. Its form-of-use is determined by the client character set or session character set.

The form of character data internal to Vantage is determined by the server character set attribute of the column.

### Terminology

Note the definition of the following expressions used throughout this section.

Expression	Definition
Character data type	The definition for the type of data stored in a column. See: <ul style="list-style-type: none"> <li>• <a href="#">CHARACTER Data Type</a></li> <li>• <a href="#">VARCHAR Data Type</a></li> <li>• <a href="#">CLOB Data Type</a></li> </ul>
<ul style="list-style-type: none"> <li>• Client character set</li> <li>• Session character set</li> <li>• Session charset</li> </ul>	Defines how Vantage translates character data from client form to server form and from server form to client form. Specifically, a client character set consists of a set of translation codes that map each character in the client character set to an equivalent character in a server character set. For details on client character sets, see <i>Teradata Vantage™ - Advanced SQL Engine International Character Set Support</i> , B035-1125. For more information on how to set the client character set for a session, refer to an appropriate tool or connectivity document such as <i>Basic Teradata® Query Reference</i> , B035-2414 or <i>Teradata JDBC Driver Reference</i> , available at <a href="https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html">https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html</a> .
Server character set	The internal character storage handling implied by the ANSI CHARACTER SET phrase. See <a href="#">CHARACTER SET Phrase</a> .

### Options

You can combine CHARACTER and VARCHAR types with the following options:

- Case option ([NOT]CASESPECIFIC or UPPERCASE).
- Server character set literal when the field data type is declared or modified.

Character literals can be prefixed with a server character set to ensure conformity to the given server character set.

## Vantage Output Conversion

Vantage translates output characters (internal-to-external, or I2E) from server form to one of the following:

- The normal character set of the logged-on client.
- The character set defined in the system tables as the default for the logon client.

Query the DBC.HostsInfoV and DBC.CharSetsV views to see the contents of the system tables.

- The character set specified for the current session, such as with the BTEQ `.SET SESSION CHARSET` command or the CLIV2 `CHARSET` call.

## CHARACTER Data Type

Represents a fixed length character string for Vantage internal character storage.

### ANSI Compliance

CHARACTER is ANSI SQL:2011 compliant.

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

### Syntax

```
{ { CHARACTER | CHAR } [ ( n ) ]
  [ { CHARACTER | CHAR } SET server_character_set ] |

  GRAPHIC [ ( n ) ]

} [ attributes [...] ]
```

### Syntax Elements

*n*

The number of characters or bytes allotted to the column defined with this server character set:

- For the LATIN server character set, the maximum value for *n* is 64000 characters.
- For the UNICODE and GRAPHIC server character sets, the maximum value for *n* is 32000 characters.
- For the KANJISJIS server character set, the maximum value for *n* is 32000 bytes.

If a value for *n* is not specified, the default is 1.



**server\_character\_set**

The server character set for the character column being defined. See [CHARACTER SET Phrase](#).

If the CHARACTER SET *server\_character\_set* clause is omitted, the default server character set depends on how the user is defined in the DEFAULT CHARACTER SET clause of the CREATE USER statement. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

**NOTICE**

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

Supported values for *server\_character\_set* are as follows:

- LATIN represents fixed 8-bit characters from the ASCII ISO 8859 Latin1 or ISO 8859 Latin9 repertoires. See [LATIN Server Character Set](#)
- UNICODE represents fixed 16-bit or 32-bit characters from the Unicode® standard. See [UNICODE Server Character Set](#).
- GRAPHIC represents fixed 16-bit UNICODE characters defined by IBM Corporation for DB2. See [GRAPHIC Server Character Set](#).
- KANJISJIS represents mixed single byte/multibyte characters intended for Japanese applications that rely on KanjiShiftJIS characteristics. See [KANJIJSJIS Server Character Set](#).

**attributes**

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Storage

Character data is allocated either in terms of characters or in terms of bytes, depending on the server character set used. The number of bytes of storage per character also varies depending on the server character set, as illustrated by the following table.

Server Character Set	Server Form-of-Use	Server Space Allocation	Sharable Among Heterogeneous Clients?
LATIN	Fixed 8-bit LATIN	Character-based	Yes
UNICODE	Fixed 16-bit UNICODE		
GRAPHIC	Fixed 16-bit UNICODE		
KANJISJIS	Mixed single and multibyte KANJISJIS	Byte-based	Yes

## External Representation of CHARACTER

Whenever a client application communicates with Vantage, it indicates its character set (form-of-use for character data). The server returns all character data to the client application in that form.

Any conversion to or from the client system data types is done by Vantage.

For information on the number of bytes exported for the CHARACTER type, see [Teradata SQL Character Strings and Client Physical Bytes](#).

## Display Format

The default display format of CHARACTER(*n*) is X(*n*). For example, X(5), where data 'HELLO' displays as 'HELLO'.

## GRAPHIC Data

You can use GRAPHIC to represent multibyte character data.

GRAPHIC(*n*) is equivalent to CHARACTER(*n*) CHARACTER SET GRAPHIC. For best practice, define all GRAPHIC(*n*) data as CHARACTER(*n*) CHARACTER SET GRAPHIC.

Each multibyte character in a graphic string is stored assuming two bytes per logical character. Therefore, a graphic data string always represents an *even* multiple of bytes.

If you specify GRAPHIC without the length (*n*), the default is GRAPHIC(1).

The following rules apply to truncation and padding of GRAPHIC data.

IF a graphic string is ...	THEN ...
shorter than the specified length of the column	the remaining space is filled with the graphic pad character.
longer than the specified length of the column	the extra characters are truncated.

## External Representation of GRAPHIC

The following table lists the client representation for the IBM DB2 GRAPHIC type.

Determining the application definitions and client data types is the responsibility of the application programmer.

Client CPU Architecture	Client Internal Data Format
IBM mainframe	2n bytes of n DB2 GRAPHIC characters.

## GRAPHIC Data and Client Character Sets

GRAPHIC types accommodate the following client character sets:

- KanjiEBCDIC double byte graphic data
- KanjiShift-JIS for double byte Shift-JIS codes
- KanjiEUC for fixed-length, double byte EUC characters

Use the following syntax for KanjiEBCDIC graphic string literals:

```
G '<graphic_characters>'
```

Under a KanjiEBCDIC character set, multibyte characters in a graphic string constant must be delimited with the Shift-Out/Shift-In characters; for example:

```
INSERT INTO TableEBCDIC (ColGRAPH)
VALUES (G'<AB>');
```

where *AB* is a valid string of KanjiEBCDIC multibyte characters, *G* specifies the string must be in the Graphic repertoire, and each apostrophe is a single byte character.

## Examples

### Example: CHARACTER Data Type

In the following table definition, the column named *Sex* is assigned the CHARACTER data type with a length of one, and the column named *Frng\_Lang* is assigned the CHARACTER data type with a length of seven.

```
CREATE TABLE PersonalData
(Id INTEGER
, Age INTEGER
, Sex CHARACTER NOT NULL UPPERCASE
, Frng_Lang CHARACTER(7) NULL UPPERCASE );
```

### Example: GRAPHIC Data and Client Character Sets

Consider the following table:

```
CREATE TABLE Product1Data
(id1 INTEGER
,code1 CHARACTER(3) CHARACTER SET GRAPHIC);
```

Assume that column code1 contains the following data:

```
457F4577456D
```

Under a KanjiEBCDIC session in record or indicator mode, the contents of code1 are returned to the user as follows:

```
457F4577456D
```

Under a KanjiEBCDIC session in field mode, the contents of code1 are returned to the user in proper format, as follows:

```
0E457F4577456D0F
```

## Related Information

FOR information on ...	SEE ...
character literals	<a href="#">Character String Literals.</a>
conversion of external-to-internal and internal-to-external character data, including truncation and error handling	<i>Teradata Vantage™ - Advanced SQL Engine International Character Set Support, B035-1125.</i>

## VARCHAR Data Type

Represents a variable length character string of length 0 to *n* for internal character storage. LONG VARCHAR specifies the longest permissible variable length character string for internal character storage.

### ANSI Compliance

VARCHAR is ANSI SQL:2011 compliant.

LONG VARCHAR, VARGRAPHIC, and LONG VARGRAPHIC are Teradata extensions to the ANSI SQL:2011 standard.

### Syntax

```
{
{ VARCHAR | { CHARACTER | CHAR } VARYING } ( n )
[ { CHARACTER | CHAR } SET ] server_character_set |
```

```

LONG VARCHAR |
VARGRAPHIC ( n ) |
LONG VARGRAPHIC
} [ attributes [...] ]

```

## Syntax Elements

*n*

The maximum number of characters or bytes allotted to the column defined with this server character set:

- For the LATIN server character set, the maximum value for *n* is 64000 characters.
- For the UNICODE and GRAPHIC server character sets, the maximum value for *n* is 32000 characters.
- For the KANJISJIS server character set, the maximum value for *n* is 32000 bytes.

### ***server\_character\_set***

The server character set for the character column being defined.

If the CHARACTER SET *server\_character\_set* clause is omitted, the default server character set depends on how the user is defined in the DEFAULT CHARACTER SET clause of the CREATE USER statement. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

### NOTICE

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

Supported values for *server\_character\_set* are as follows:

- LATIN represents fixed 8-bit characters from the ASCII ISO 8859 Latin1 or ISO 8859 Latin9 repertoires. See [LATIN Server Character Set](#).
- UNICODE represents fixed 16-bit or 32-bit characters from the Unicode® standard. See [UNICODE Server Character Set](#).
- GRAPHIC represents fixed 16-bit UNICODE characters defined by IBM Corporation for DB2. See [GRAPHIC Server Character Set](#).

- KANJISJIS represents mixed single byte/multibyte characters intended for Japanese applications that rely on KanjiShiftJIS characteristics. See [KANJISJIS Server Character Set](#).

### attributes

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Storage

Character data is allocated either in terms of characters or in terms of bytes, depending on the server character set used. The number of bytes of storage per character also varies depending on the server character set, as illustrated by the following table.

Server Character Set	Server Form-of-Use	Server Space Allocation	Sharable Among Heterogeneous Clients?
LATIN	Fixed 8-bit LATIN	Character-based	Yes
UNICODE	Fixed 16-bit UNICODE		
GRAPHIC	Fixed 16-bit UNICODE		
KANJISJIS	Mixed single and multibyte KANJISJIS	Byte-based	Yes

Any conversion to or from client system data types is done by Vantage. This data type supports international character sets.

### External Representation of VARCHAR

Whenever a client application talks to Vantage, it indicates its character set (form-of-use for character data). The server returns all character data to the application in that form.

For information on the number of bytes exported for the VARCHAR type, see [Teradata SQL Character Strings and Client Physical Bytes](#).

### External Representation of LONG VARCHAR

For information on the number of bytes exported for the LONG VARCHAR type, see [Teradata SQL Character Strings and Client Physical Bytes](#).

The following table shows how LONG VARCHAR data is represented for the various server character sets. Apart from these definitions, LONG VARCHAR strings behave identically to VARCHAR strings.

FOR this server character set ...	The external representation for LONG VARCHAR is equivalent to ...
<ul style="list-style-type: none"> <li>• LATIN</li> <li>• KANJI1</li> </ul>	VARCHAR(64000)
<ul style="list-style-type: none"> <li>• UNICODE</li> <li>• GRAPHIC</li> <li>• KANJISJIS</li> </ul>	VARCHAR(32000)

## External Representation of VARGRAPHIC

The following table lists the client representation for the IBM DB2 VARGRAPHIC type.

Determining the application definitions and client data types is the responsibility of the application programmer.

Define the length of the VARGRAPHIC( $n$ ) string as  $k$ , where  $0 \leq k \leq n$ .

Client CPU Architecture	Client Internal Data Format
IBM mainframe	Two byte SMALLINT length count $k$ followed by $k$ DB2 GRAPHIC characters for a total of $2+2k$ EBCDIC bytes.

## Display Format

The default display format for VARCHAR( $n$ ) is X( $n$ ).

The default display format for LONG VARCHAR is either X(32000) or X(64000) depending on the server character set in effect.

For more information, see [Data Type Default Formats](#).

## Graphic Data

You can use the VARGRAPHIC or LONG VARGRAPHIC to represent multibyte character data.

VARGRAPHIC( $n$ ) is equivalent to VARCHAR( $n$ ) CHARACTER SET GRAPHIC. For best practice, define all VARGRAPHIC( $n$ ) data as VARCHAR( $n$ ) CHARACTER SET GRAPHIC.

The maximum value for  $n$  in a VARCHAR( $n$ ) CHARACTER SET GRAPHIC definition is 32000. There is no default length; therefore, omitting the length specification results in an error.

LONG VARGRAPHIC is equivalent to LONG VARCHAR CHARACTER SET GRAPHIC. For best practice, define all LONG VARGRAPHIC data as LONG VARCHAR CHARACTER SET GRAPHIC.

## Examples

### Example: VARCHAR Data Type

The following statement creates a table that defines two VARCHAR columns: InfoKey and InfoData.

```
CREATE TABLE InfoTable
  (InfoKey VARCHAR(10) NOT NULL
  ,InfoData VARCHAR(16384) )
UNIQUE PRIMARY INDEX ( InfoKey );
```

The following statements insert character data of varying lengths into the InfoKey and InfoData columns:

```
INSERT INTO InfoTable ('001_5_799', 'Data for key 001_5_799');
INSERT INTO InfoTable ('2', 'Data for key 2');
```

**Example: LONG VARCHAR Data Type**

The following statement creates a table that defines a LONG VARCHAR column called InfoData.

```
CREATE TABLE InfoTable (InfoData LONG VARCHAR);
```

**Related Information**

FOR information on ...	SEE ...
character literals	<a href="#">Character String Literals.</a>
conversion of external-to-internal and internal-to-external character data, including truncation and error handling	<i>Teradata Vantage™ - Advanced SQL Engine International Character Set Support</i> , B035-1125.

**CLOB Data Type**

Represents a large character string. A character large object (CLOB) column can store character data, such as simple text or HTML.

**Note:**

A CLOB column can store XML or JSON documents; however, Teradata recommends that you use the Teradata XML and Teradata JSON data types for that purpose.

**ANSI Compliance**

CLOB is ANSI SQL:2011 compliant.

**Syntax**

```
{ CHARACTER LARGE OBJECT | CLOB }
[ ( n [ K | M | G ] ) ]
```



```
[ { CHARACTER | CHAR } SET { LATIN | UNICODE } ]
[ attribute [...] ]
```

## Syntax Elements

### ***n***

The number of characters to allocate for the CLOB column. The maximum value depends on the server character set:

- For the LATIN server character set, *n* cannot exceed 2097088000.
- For the UNICODE server character set, *n* cannot exceed 1048544000.

If a value for *n* is not specified, the default is the maximum value.

### ***K***

The number of characters to allocate for the CLOB column is *nK*, where K = 1024 and the maximum value for *n* is as follows:

- For the LATIN server character set, *n* cannot exceed 2047937.
- For the UNICODE server character set, *n* cannot exceed 1023968.

### ***M***

The number of characters to allocate for the CLOB column is *nM*, where M = 1024K and the maximum value for *n* is as follows:

- For the LATIN server character set, *n* cannot exceed 1999.
- For the UNICODE server character set, *n* cannot exceed 999.

### ***G***

The number of characters to allocate for the CLOB column is *nG*, where G = 1024M. When G is specified, *n* must be 1 and the server character set must be LATIN.

## CHARACTER SET

The server character set for the CLOB column being defined:

- The LATIN server character set represents fixed 8-bit characters from the ASCII ISO 8859 Latin1 or ISO 8859 Latin9 repertoires.
- The UNICODE server character set represents fixed 16-bit or 32-bit characters from the Unicode® standard.

If the CHARACTER SET clause is omitted, the default server character set depends on how the user is defined in the DEFAULT CHARACTER SET clause of the CREATE USER statement. For details, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Also see [CHARACTER SET Phrase](#).

### **attribute**

Appropriate data type, column storage, or column constraint attributes.

A CLOB column supports the following attributes:

- NOT NULL
- FORMAT
- TITLE

For more information about the NOT NULL attribute, see [Default Value Control Phrases](#).

For more information about the FORMAT and TITLE attributes, see [Data Type Formats and Format Phrases](#).

## Usage Notes

### External Representation

Whenever a client application talks to Vantage, it indicates its character set (form-of-use for character data). Vantage returns all character data to the application in that form.

Whenever multibyte characters are involved, their representation has the length  $(n-2)/2$  multibyte characters exclusive of the preceding Shift-Out and following Shift-In characters.

The following table lists the client representations for the CLOB data type. Determining the application definitions and client data types is the responsibility of the application programmer.

Define the length of the CLOB string as  $k$ , where  $0 \leq k \leq n$ .

Client CPU Architecture	Client Internal Data Format
IBM mainframe	Eight byte (most significant byte first) length count $k$ followed by $k$ bytes of EBCDIC character data for a total of $k+8$ bytes.
<ul style="list-style-type: none"> <li>• UTS</li> <li>• RISC</li> <li>• Motorola 68000</li> <li>• WE 32000</li> </ul>	Eight byte (most significant byte first) length count $k$ followed by $k$ bytes of ASCII character data for a total of $k+8$ bytes.
Intel	Eight byte (least significant byte first) length count $k$ followed by $k$ bytes of ASCII character data for a total of $k+8$ bytes.

### Restrictions

A table can have a maximum of 32 LOB columns.

Queue tables cannot have CLOB columns.

A LOB column cannot be a component of an index. Because of this restriction, a table must define at least one non-LOB column.

The CHARACTER SET clause can specify the following server character sets for a CLOB column:

- LATIN
- UNICODE

## Functions That Operate on CLOBs

The following are some functions and operators that support CLOB types:

- CHARACTERS/CHARS/CHAR
- MCHARACTERS
- CHARACTER\_LENGTH
- TRANSLATE and TRANSLATE\_CHK
- SUBSTRING/SUBSTR
- Concatenation operator (||)
- TYPE
- Explicit data type conversion (CAST and Teradata conversion syntax)
- User-defined functions (UDFs)
- Stored procedures
- External stored procedures

## Examples

### Example: CLOB Data Type

The following example creates a table that defines a CLOB column named clarge:

```
CREATE TABLE t1
(id INTEGER
,clarge CLOB(2K) CHARACTER SET UNICODE);
```

SQL Engine stores the character data using the UNICODE server character set.

### Example: Inserting CLOB Data

The following example shows a stored procedure that inserts part of a CLOB into one table and the remaining part of the CLOB into another table:

```
CREATE TABLE LocalData(ld_ID INTEGER, ld_DATA CLOB);
CREATE TABLE GlobalData (gd_ID INTEGER, gd_DATA CLOB);

CREATE PROCEDURE DataSplitter(IN local_ID  INTEGER,
                              IN global_ID INTEGER,
```

```

                                IN all_DATA CLOB)

BEGIN

    INSERT LocalData (local_ID, SUBSTRING(all_DATA FROM 1 FOR 128546));
    INSERT GlobalData (global_ID, SUBSTRING(all_DATA FROM 128547));

END;
```

Related Information

FOR information on ...	SEE ...
on functions and operators that support CLOB types	<i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145.
implementing UDFs and external stored procedures that operate on CLOB types	<i>Teradata Vantage™ - SQL External Routine Programming</i> , B035-1147.
implementing stored procedures that use CLOB local variables or parameters	<ul style="list-style-type: none"><li>• <i>CREATE PROCEDURE</i> in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li><li>• <i>Teradata Vantage™ - SQL Stored Procedures and Embedded SQL</i>, B035-1148.</li></ul>

Default Case Specificity of Character Columns

Default case specificity varies with mode.

Case specificity does not apply to CLOB types.

ANSI Mode

CHARACTER and VARCHAR data columns defined in ANSI mode are defined as CASESPECIFIC by default. This means that the values 'aa' and 'AA' are not treated as equals.

An option of NOT CASESPECIFIC is implemented to allow columns to be not case specific. NOT CASESPECIFIC is a Teradata extension to the ANSI standard.

Teradata Mode

CHARACTER and VARCHAR data columns defined in Teradata mode (with the exception of character data defined as CHAR or VARCHAR CHARACTER SET GRAPHIC) are defined as NOT CASESPECIFIC by default.

Because Latin characters have canonical representation in all Vantage character sets, the effect of the UPPERCASE function is the same for each character set (with the exception of KANJI1, where multibyte characters are not converted to uppercase).

The CASESPECIFIC and UPPERCASE phrases are described in detail in:

- [CASESPECIFIC Phrase](#)
- [UPPERCASE Phrase](#)

## CASESPECIFIC Phrase

Specifies case for character data comparisons and collations.

### ANSI Compliance

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

### Syntax

```
[ NOT ] { CASESPECIFIC | CS }
```

## Usage Notes

### Case Criteria and Issues

The following table summarizes case criteria and issues.

IF the case specification is ...	THEN ...
CASESPECIFIC (CS)	<p>In sorts or comparisons, lowercase characters are not converted to uppercase and are not equal to uppercase characters.</p> <ul style="list-style-type: none"> <li>• If a column is defined as CASESPECIFIC, characters entered as 'aaa' are not equivalent to 'AAA' when used in a unique index.</li> <li>• Applications that conform to ANSI should define columns for ANSI mode and use the UPPER function to force comparisons that are not case specific.</li> </ul>
NOT CASESPECIFIC (NOT CS)	<p>In sorts or comparisons, lowercase characters are converted to uppercase. 'aaa' is equivalent to 'AAA'.</p> <p>'AAA', 'aaa', 'AaA', and 'aAA' are all equivalent as unique indexes.</p>
Neither CASE SPECIFIC nor NOT CASESPECIFIC	<p>The session mode determines the attribute assigned by default.</p> <ul style="list-style-type: none"> <li>• In ANSI mode, CASESPECIFIC is set.</li> <li>• In Teradata mode (except for CHAR or VARCHAR CHARACTER SET GRAPHIC data), NOT CASESPECIFIC is set.</li> </ul>

### Rules: All Modes

The following set of rules applies to CASESPECIFIC and NOT CASESPECIFIC in both ANSI and Teradata modes.

- CASESPECIFIC specifies that comparisons are case-specific.
- NOT CASESPECIFIC specifies that comparisons are not case-specific.

- If neither CASESPECIFIC nor NOT CASESPECIFIC is specified as part of the column definition, then you can specify the option in an SQL request to ensure that the statement behaves as intended. By including an explicit CASESPECIFIC (or CS) qualifier in your SQL statement, you override the case specificity for a column.
- In character string comparisons, if either of the strings being compared is CASESPECIFIC, then the comparison is always CASESPECIFIC.
- A column typed as CHAR or VARCHAR CHARACTER SET GRAPHIC defaults to CASESPECIFIC when it is created.
- To create a non-CASESPECIFIC CHAR or VARCHAR CHARACTER SET GRAPHIC column, you must specify a NOT CASESPECIFIC clause in the column definition. You can also use NOT CASESPECIFIC in a comparison predicate.
- KANJI1 data only handles NOT CASESPECIFIC for characters in the range A–Z. All other server character sets handle all data correctly.
- The following data types do not support CASESPECIFIC or NOT CASESPECIFIC:
  - CLOBs
  - UDTs
- Character data is stored as typed unless the UPPERCASE phrase is specified. See [UPPERCASE Phrase](#).

Note that the performance of views, macros, and CHECK table constraints can involve parsing character string literals at execution time as well as the more common processing of character string literals in queries.

For more information, see SELECT in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

### Rules: Teradata Mode

The following rules apply to CASESPECIFIC and NOT CASESPECIFIC in Teradata mode.

- All character types except CHAR or VARCHAR CHARACTER SET GRAPHIC default to NOT CASESPECIFIC.

You can override this default in a query by specifying (CASESPECIFIC) immediately following the CHARACTER data to be compared.

For example, the following query is not case specific in Teradata mode:

```
SELECT DataBaseName
FROM DBC.Databases
WHERE DataBaseName = 'dbc';
```

To make it case specific, qualify the predicate with (CASESPECIFIC):

```
SELECT DataBaseName
FROM DBC.Databases
WHERE DataBaseName = 'dbc' (CASESPECIFIC);
```

Note that the first example returns one row and the second example returns no rows.

- CASESPECIFIC supports all letters in the ISO 10646 repertoire. See also [UPPERCASE Phrase](#).

## Rules: ANSI Mode

The following rules apply to CASESPECIFIC and NOT CASESPECIFIC in ANSI mode.

- All character data defaults to CASESPECIFIC.
- To make comparisons that are not case specific using ANSI SQL:2011 syntax, you must apply the UPPER function to any character string value that may contain lower case Latin letters.
- To make comparisons that are not case specific using Teradata syntax, you can apply the NOT CASESPECIFIC specification to the appropriate character string. The UPPER function is preferable because it complies with the ANSI SQL:2011 standard. NOT CASESPECIFIC is a Teradata extension to the standard.

For more information on UPPER, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

- CASESPECIFIC supports all letters in the ISO 10646 repertoire.

If you declare a character column to be NOT CASESPECIFIC and type 'ᾱ' (GREEK SMALL LETTER ALPHA WITH TONOS), then that letter is stored as 'ᾱ' but during comparison compares equal to 'Α' (GREEK CAPITAL LETTER ALPHA WITH TONOS).

The same applies to the (NOT CASESPECIFIC) qualifier in a SQL predicate.

The following SQL predicate evaluates to TRUE.

```
'ᾱ' (NOT CASESPECIFIC) = 'Α'
```

Without the (NOT CASESPECIFIC) qualifier, the same predicate evaluates to FALSE.

## Examples

### Example: CASESPECIFIC Phrase

The following query returns a result only if a case specific comparison of the literal 'Leidner P' finds a match.

```
SELECT Name
FROM Employee
WHERE Name(CS) = 'Leidner P' ;
```

The literal 'Leidner P' might default to CS or NOT CS, depending on the current mode, but because the type modifier of the comparison is CS, the comparison is case specific irrespective of the session mode.

To ensure a comparison that is not case specific irrespective of the session mode, specify the query as follows.

```
SELECT Name
FROM Employee
WHERE Name (NOT CS) = 'Leidner P' (NOT CS) ;
```

Alternatively, you can specify the query using ANSI-compatible syntax as follows.

```
SELECT Name
FROM Employee
WHERE UPPER (Name) = 'LEIDNER P' ;
```

**Example: CASESPECIFIC Comparison and Collation on Mixed Case Data**

CASESPECIFIC comparison and collation on mixed case data can produce unintended results.

```
SELECT Last_Name
FROM SalesReps
ORDER BY Last_Name(CS) ;
```

might return one of the following sorted lists depending on the collation in effect for the session.

EBCDIC	ASCII	MULTINATIONAL
bart	ACME	ACME
fernandez	ALBERT	Albert
hill	Albert	ALBERT
Albert	FARRAH	bart
ACME	Kimble	FARRAH
ALBERT	bart	fernandez
FARRAH	fernandez	hill
Kimble	hill	Kimble

**UPPERCASE Phrase**

Specifies that character data for a column is stored as uppercase.

Also used in CAST and with Teradata conversion syntax to convert a value to uppercase.



### ANSI Compliance

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

### Syntax

```
{ UPPERCASE | UC }
```

### Usage Notes

#### Case Criteria and Issues

The following table summarizes case criteria and issues.

Case Specification	Usage Rules
UPPERCASE <i>not</i> specified NOT CASESPECIFIC specified	The following are all equivalent as unique indexes: <ul style="list-style-type: none"> <li>• AAA</li> <li>• aaa</li> <li>• AaA</li> <li>• aAA</li> </ul>
UPPERCASE (UC)	Character data is stored in uppercase regardless of the case in which it was typed. Because of the conversion, if a column is defined as UPPERCASE, characters typed as 'aaa' are stored as 'AAA' and therefore are equivalent to 'AAA' when used in a unique index.

#### Rules: All Modes

The following rules apply to UPPERCASE in both ANSI and Teradata modes.

- You can specify the UPPERCASE option in an SQL request to override the stored case specificity assignment for a column.
- If a column is declared to be UPPERCASE, then lowercase letters are converted and stored as their uppercase equivalents.

Note that the KANJI1 server character set only converts the 26 letters a-z to uppercase.

- UPPERCASE supports all letters in the ISO 10646 repertoire.

If you declare a character column to be UPPERCASE and type 'ά' (GREEK SMALL LETTER ALPHA WITH TONOS), then that character is translated and stored as 'Α' (GREEK CAPITAL LETTER ALPHA WITH TONOS).

The same applies to the (UC) qualifier.

The following SQL predicate evaluates to TRUE.

```
'ά' (UC) = 'Α'
```

- The following data types do not support UPPERCASE
  - CLOBs
  - UDTs

## UPPERCASE Phrase and UPPER Function

The UPPER function is defined by the ANSI SQL:2011 standard and is *not* the same as declaring a value to be UPPERCASE. For more information, see UPPER in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Examples

### Example: UPPERCASE Phrase

The Gender column is created using the UPPERCASE option.

```
Gender CHAR UPPERCASE
```

If Gender data is entered in lowercase as shown in the following INSERT statement, then it is stored and returned in uppercase.

```
INSERT INTO Employee (Name, EmpNo, Gender ...
VALUES ('Smith', 10021, 'f', ...);
```

```
SELECT Gender
FROM Employee;
      Gender
      F
```

### Example: UPPERCASE Phrase With the Concatenation Operator

If used with the concatenation (||) operator, UC must be enclosed in parentheses, and be placed immediately after the column name.

```
SELECT (City_name (UC)) || ', ' || State, Population
FROM World
WHERE Country = 'USA'
ORDER BY Population ;
```

## Teradata SQL Character Strings and Client Physical Bytes

The server character sets LATIN, UNICODE, and GRAPHIC are declared within Teradata SQL as strings of characters.

Within a client environment, those character data types are expected to be specified as strings of physical bytes rather than as characters.

For example, within client application programs, space is allocated in terms of bytes, the layout of the import files for data loading utilities is expressed in terms of bytes, and so on.

To provide a reasonable bridge between characters in SQL declarations and physical bytes in client environments, clients can specify character strings in bytes even though the destination columns are declared as characters.

Whenever a client is informed about the length of the character expression (as in the `HELP COLUMN` statement, response parcels, and so on), Vantage specifies the length in terms of physical bytes.

### General Rules for Server Character Set Declarations

The rules for declaring data types are as follows:

- The length of a character type is declared in terms of bytes or characters, depending on the server character set.

IF a character type uses this server character set ...	THEN the type is defined in terms of ...
<ul style="list-style-type: none"> <li>LATIN</li> <li>UNICODE</li> <li>GRAPHIC</li> </ul>	characters
<ul style="list-style-type: none"> <li>KANJI1</li> <li>KANJISJIS</li> </ul>	bytes

- The length of the imported CHARACTER data described by a USING clause is in terms of bytes (the CHARACTER data description in the USING clause is in bytes). For more information, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- The definition of the character data in response and request parcels is in terms of bytes.
- Client Load (Teradata FastLoad and Teradata MultiLoad) and Export (Teradata FastExport) utilities use the MultiLoad and FastExport DEFINE or LAYOUT command to specify the layout of the file on the client.

The layout of CHARACTER data within the DEFINE command is in terms of bytes.

The layout of GRAPHIC data within the DEFINE command is in terms of characters. This layout is then translated into the response parcel or the USING clause for communication with Teradata SQL. For more information, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

### Exported Character Data

Export width determines the appropriate number of characters or bytes to export for a client application. System-wide and user-defined export width tables define the export width of characters or bytes during export to a client. The values used for the export width are based on the active export width table, and the server character set and client character set used by the application.

At the system level, the value of the Export Width Table ID field in the DBS Control Record identifies which export width table to use to determine export widths.

The following table lists the system export width tables provided by Teradata:

Export Width Table Name	Export-ID	Description
Expected Default	0	Provides reasonable default widths for the character data type and client form of use. This is the initial default table.
Compatibility Default	1	Enables Unicode data to be processed by applications that are written to process Latin or KANJI1 data.
Maximum Default	2	Provides maximum default width of the character data type and client form of use.

For information about how to use the DBS Control utility to set the system-level export width for your system, see *Teradata Vantage™ - Database Utilities*, B035-1102.

You can override the system-level export width by using the EXPORTWIDTH option in the CREATE USER or MODIFY USER statements. For more information, see:

- CREATE USER and MODIFY USER in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.
- *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## HELP Statements

The export width rules determine HELP TABLE and HELP COLUMN statement results.

## CHARACTER SET Phrase

Specifies the server character set for a character column.

### ANSI Compliance

The CHARACTER SET phrase is entry-level ANSI-compliant.

Implementation-defined character sets are intermediate-level ANSI-compliant.

### Syntax

```
{ CHARACTER | CHAR } SET { LATIN | UNICODE | GRAPHIC | KANJISJIS }
```

## Usage Notes

### Using CHARACTER SET to Define Internal Storage of a Column

To define the server character set for a character column, use the CHARACTER SET phrase of a character column definition in the CREATE TABLE statement.

This optional phrase defines the internal handling of character data stored on the server and has nothing to do with the client, or session, character set.

If you omit the CHARACTER SET clause, then the default server character set for the column depends on how the user accessing the data in the table is defined in the DEFAULT CHARACTER SET clause of the CREATE USER statement. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. If there is no default character set specified for the user, the system defaults to the character set associated with the language mode, that is, LATIN for standard language mode and UNICODE for Japanese language mode.

### Supported Server Character Sets for CHARACTER SET

Teradata SQL supports five server character sets for use with CHARACTER SET. The following topics discuss each server character set in detail:

- [LATIN Server Character Set](#)
- [UNICODE Server Character Set](#)
- [GRAPHIC Server Character Set](#)
- [KANJI1 Server Character Set](#)
- [KANJI1 Server Character Set \[Deprecated\]](#)

#### NOTICE

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

### Restrictions

You cannot use the CHARACTER SET phrase for a UDT column.

### Using the CHARACTER SET Phrase for Translation

The CHARACTER SET phrase is a form of explicit translation.

For information on explicit translation of CHARACTER data, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

If the CHARACTER SET phrase is applied to a character expression, the expression is first translated using the implicit translation rules (see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145), and then truncated or extended as appropriate.

FOR this <i>server_character_set</i> ...	THE <i>n</i> in CHAR( <i>n</i> ) or VARCHAR( <i>n</i> ) indicates the number of these in the translation result ...
<ul style="list-style-type: none"> <li>• UNICODE</li> <li>• LATIN</li> <li>• GRAPHIC</li> </ul>	characters
<ul style="list-style-type: none"> <li>• KANJISJIS</li> <li>• KANJ11</li> </ul>	bytes

## LATIN Server Character Set

### Intended Use

U.S. and European applications using only those characters from ASCII, ISO 8859 Latin1, or proposed ISO 8859 Latin9 repertoires.

### Pad Character

SPACE (0x20)

### SQL Declaration

To specify the LATIN server character set for a character column, use the following syntax.

Data Type	Maximum Value for <i>n</i>
CHARACTER( <i>n</i> ) CHARACTER SET LATIN	64000 (This is also the size of LONG VARCHAR CHARACTER SET LATIN.)
VARCHAR( <i>n</i> ) CHARACTER SET LATIN	
CLOB( <i>n</i> ) CHARACTER SET LATIN	2097088000

### Usage Notes

The semantics of the 8-bit LATIN server character set determine what characters are considered letters for the purpose of uppercasing, and how to translate between server character sets.

Database storage space for this type is allocated on a character basis.

For more information about the LATIN server character set, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

## UNICODE Server Character Set

### Intended Use

International applications using the Unicode repertoire or a subset thereof.

### Pad Character

SPACE (U+0020)

### SQL Declaration

To specify the UNICODE server character set for a character column, use the following syntax.

Data Type	Maximum Value for <i>n</i>
CHARACTER( <i>n</i> ) CHARACTER SET UNICODE	32000 (This is also the size of LONG VARCHAR CHARACTER SET UNICODE.)
VARCHAR( <i>n</i> ) CHARACTER SET UNICODE	
CLOB( <i>n</i> ) CHARACTER SET UNICODE	1048544000

### Usage Notes

The UNICODE server character set supports the 16-bit BMP characters from Unicode® 6.0. For a list of the supported characters, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Unicode also allows the storage and searching of all other 16-bit and 32-bit Unicode characters. For more information, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Each code point represents a distinct character, including non-spacing characters such as diacritical marks and joiners. All characters named as letters in UNICODE are considered as such, and are candidates for uppercasing.

Database storage space for UNICODE is allocated on a character basis. 32-bit Pass Through Characters (for example, emoji) require two 16-bit UTF-16 code units, or 4 bytes. Therefore, VARCHAR(2) or CHAR(2) are the minimum sizes required to store a Pass Through Character.

For details about the UNICODE server character set, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

## GRAPHIC Server Character Set

### Intended Use

DB2 compatibility.

**Pad Character**

IDEOGRAPHIC SPACE (U+3000)

**N-Character SQL Declaration**

CHARACTER(*n*) CHARACTER SET GRAPHIC

or

GRAPHIC(*n*)

**Maximum Value for *n***

32000

**Usage Notes**

GRAPHIC data can also be defined as GRAPHIC(*n*) or VARGRAPHIC(*n*); however, for best practice, define all GRAPHIC data as CHARACTER(*n*) CHARACTER SET GRAPHIC or VARCHAR(*n*) CHARACTER SET GRAPHIC.

All characters named as letters in this subset of UNICODE are considered as such, and are candidates for uppercasing.

For more information about the GRAPHIC server character set, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

**Restrictions**

CLOBs do not support the GRAPHIC server character set.

**KANJISJIS Server Character Set****Intended Use**

Japanese applications that rely on the KANJISJIS characteristics (for example, KANJISJIS physical space allocation).

**Pad Character**

ASCII SPACE (0x20)

**N-Character SQL Declaration**

CHARACTER(*n*) CHARACTER SET KANJISJIS

**Maximum Value for *n***

32000

This also is the size of LONG VARCHAR CHARACTER SET KANJISJIS.



**Usage Notes**

The KANJISJIS type has a mixed single byte/multibyte character form-of-use of the KANJISJIS client character set.

The database storage space allocation for this type is on a byte basis rather than character basis. When updating the field, truncation occurs based on this physical space.

All SQL string functions on this type operate on a logical character basis.

This type is included for third party tools that rely on the semantics of KANJISJIS, such as the string length or physical space allocation.

For more information about the KANJISJIS server character set, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

**Restrictions**

CLOBs do not support the KANJISJIS server character set.

**KANJI1 Server Character Set [Deprecated]****Intended Use**

Japanese applications that must remain compatible with previous Vantage and Teradata Database Kanji releases.

**NOTICE**

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

**Pad Character**

ASCII SPACE (0x20)

**SQL Declaration**

CHARACTER(*n*) CHARACTER SET KANJI1

VARCHAR(*n*) CHARACTER SET KANJI1

**Maximum Value for *n***

64000

This also is the size of LONG VARCHAR CHARACTER SET KANJI1.

## Usage Notes

KANJI1 contains mixed single- and multibyte characters in the KanjiEBCDIC, KANJISJIS, or KanjiEUC client character set, as determined by the current character set for the session.

KANJI1 inherits the semantics and limitations on character types of prior releases. For example, it is possible to generate nonvalid strings with the SUBSTR function.

Note the following peculiarity of KANJI1:

- The single byte character portion of the type is stored in the canonical form defined by JIS X 0201.
- The multibyte character portion of the type is stored in the client form-of-use and thus cannot be shared among heterogeneous clients.

The maximum length in bytes of the exported character data for the KANJI1 server character set is always  $n$  for CHARACTER( $n$ ) and VARCHARACTER( $n$ ).

On a system that is enabled with Japanese language support, the database assumes that *all* character data is mixed single and multibyte characters. Mixed single and multibyte character data is associated with any column defined as CHAR, VARCHAR, or LONG VARCHAR.

Encoding of the KANJI1 character set is based on the KanjiEBCDIC, KanjiEUC, or KanjiShift-JIS client character set, depending on the current character set for the session.

Depending on the character set of the session, single byte and multibyte characters are distinguished as described by the following table.

Client Character Set	Definition
KanjiEBCDIC	Characters are assumed to be single byte until a shift-out character is encountered. Subsequent characters are assumed to be multibyte characters until a Shift-In character is encountered. If the end of the string is reached without finding Shift-In, an error condition occurs.
KanjiEUC	The first byte of a multibyte character always has the most significant bit on. Multibyte characters are two bytes except KanjiEUC cs3 characters, which require three bytes.
KanjiShift-JIS	

If the CASESPECIFIC option is defined for the character column, conversion to uppercase is not performed, and simple Latin letters (A...Z, a...z) are considered to match only if they are the same letters and the same case.

For more information about the KANJI1 server character set, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

## Conversion to Uppercase

The effect of the UPPERCASE function on various Japanese characters is illustrated in the following table.

Client Character Set	Character String	Conversion Result
KanjiEBCDIC	mN<abc>b	MN<abc>B
KanjiEUC	mn a ss <u>2</u> B ss <u>3</u> c	MNa ss <u>2</u> B ss <u>3</u> c
KanjiShift-JIS	mn labc	MN labc

Because simple Latin letters always have the same canonical representation, the effect of converting to uppercase is the same across all the character sets supported by Vantage.

## Restrictions

CLOB types do not support the KANJI1 server character set.

## Padding and Truncation for CHARACTER Types

In Teradata mode, if a character expression is assigned to a CHAR column of a shorter length, the extra bytes are truncated. This may result in an improper string. You are not informed that this truncation has occurred.

In ANSI mode, an error occurs if a nonblank character is truncated.

If a character expression of some length is assigned to a CHAR column of a longer length, the field is padded with the SPACE character.

Shorter strings are padded with single-byte spaces, regardless of whether the mode is Teradata or ANSI. Only truncation differs between the two modes.

## Multibyte Character Data Validation and Storage

Translation and storage of validated multibyte character data on the server depends on the character set of the current session, as explained in the following table.

For this client character set ...	Multibyte characters are translated and stored on the server ...
KanjiEUC	<p>according to the client encoding of the current session, as follows:</p> <ul style="list-style-type: none"> <li>Code set cs0: For each character of this code set, the first byte is translated to all characters, from EUC to the internal representation (based on JIS X 0201) and stored as single byte characters.</li> <li>Code set cs1: For each character of this code set, the first byte is translated to KanjiShift-JIS character data only (based on JIS X 0208), as illustrated in the translation map in <i>Teradata Vantage™ - Advanced SQL Engine International Character Set Support</i>, B035-1125. GRAPHIC data is stored without translation. Subsequent characters are also translated to KanjiShift-JIS.</li> <li>Code set cs2:</li> </ul>

For this client character set ...	Multibyte characters are translated and stored on the server ...
	<p>For each character of this code set, the first byte is translated to character data only. The first byte (ss<sub>2</sub>=0x8E) is translated to 0x80 and the second byte is left unmodified.</p> <ul style="list-style-type: none"> <li>Code set cs3: For each character of this code set, the first byte is translated to character data only. The first byte (ss<sub>3</sub>=0x8F) is translated to 0xFF and the remaining 2 bytes are left unmodified.</li> </ul> <p>GRAPHIC data is stored without translation.</p>
KanjiEBCDIC	as received.
KanjiShift-JIS	<p>They are not translated and remain in the client encoding.</p> <p>For KanjiEBCDIC only, the Shift-Out and Shift-In characters are stored as part of the string after being translated to the same encoding (0x0E and 0x0F, respectively).</p>

### Example: Fixed Length KanjiEBCDIC

Assume that a fixed-length column is to contain the following KanjiEBCDIC data:

```
< S T R I N G > 12
```

The column definition must be at least 16 bytes (CHAR(16) CHARACTER SET KANJI1) to accommodate the internal representation of the data plus the Shift-Out (<) and Shift-In (>) characters, which is as follows:

```
0E 42E2 42E3 42D9 42C9 42D5 42D7 0F 31 32
```

Note that each of the three client representations of multibyte character data could require a different length for the same sequence of symbols.

### Example: Fixed Length KanjiShift-JIS

The same string in KanjiShift-JIS is as follows:

```
S T R I N G 12
```

and requires a length of only 14 bytes (CHAR(14) CHARACTER SET KANJISJIS). The internal equivalent for the KanjiShift-JIS string is as follows:

```
8272 8273 8271 8268 826D 8266 31 32
```

# Byte and BLOB Data Types

This section describes the BYTE, VARBYTE, and BINARY LARGE OBJECT (BLOB) data types.

## Data Storage of Byte and BLOB Types

The BYTE, VARBYTE, and BLOB data types are stored in the client system format—they are never translated by Vantage.

The BYTE, VARBYTE, and BLOB data types store raw data as logical bit streams. For any machine, BYTE, VARBYTE, and BLOB data is transmitted directly from the memory of the client system. The sort order is logical, and values are compared as if they were *n*-byte, unsigned binary integers suitable for digitized images.

Logical bit streams are not translated. However, the interpretation of a hexadecimal string in EBCDIC form depends on the client system from which that string is submitted.

BLOB is ANSI SQL:2011 compliant. BYTE and VARBYTE are Teradata extensions to the ANSI standard.

## BYTE Data Type

Represents a fixed-length binary string.

### ANSI Compliance

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

### Syntax

```
BYTE [ ( n ) ] [ attributes [...] ]
```

### Syntax Elements

***n***

The number of bytes in the string.

Maximum value: 64000

Default: 1

***attributes***

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### External Representation of BYTE

The following table lists the client representations for the BYTE data type.

Determining the application definitions and client data types is the responsibility of the application programmer.

Client CPU Architecture	Client Internal Data Format
IBM mainframe	<i>n</i> bytes.
<ul style="list-style-type: none"> <li>• UTS</li> <li>• RISC</li> <li>• Motorola 68000</li> <li>• WE 32000</li> <li>• Intel</li> </ul>	<i>n</i> bytes.

## Example

In the following example, the column named BinaryData is assigned the BYTE data type and has an upper limit of 1200 bytes:

```
CREATE TABLE DocTable
  (DocType INTEGER
  ,DocName CHAR(26)
  ,BinaryData BYTE(1200));
```

## VARBYTE Data Type

Represents a variable-length binary string.

### ANSI Compliance

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

### Syntax

```
VARBYTE ( n ) [ attributes [...] ]
```

### Syntax Elements

*n*

The number of bytes in the string.

The maximum value for  $n$  is 64000.

### attributes

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### External Representation of VARBYTE

The following table lists the client representations for the Teradata SQL VARBYTE data type.

Define the length of the VARBYTE string as  $k$ , where  $0 \leq k \leq n$ .

Client CPU Architecture	Client Internal Data Format
IBM mainframe	$k + 2$ bytes 16-bit SMALLINT (2 byte) length count $k$ followed by $k$ bytes of BYTE data.
<ul style="list-style-type: none"> <li>• UTS</li> <li>• RISC</li> <li>• Motorola 68000</li> <li>• WE 32000</li> <li>• Intel</li> </ul>	$k + 2$ bytes 16-bit SMALLINT (2 byte) length count $k$ followed by $k$ bytes of BYTE data.

## Example

In the following example, the column named BinaryData is assigned the VARBYTE data type and has an upper limit of 1200 bytes:

```
CREATE TABLE DocTable
  (DocType INTEGER
  ,DocName CHAR(26)
  ,BinaryData VARBYTE(1200));
```

## BLOB Data Type

Represents a large binary string of raw bytes. A binary large object (BLOB) column can store binary objects, such as graphics, video clips, files, and documents.

### ANSI Compliance

BLOB is ANSI SQL:2011 compliant.

## Syntax

```
{ BINARY LARGE OBJECT | BLOB }
  [ ( n [ K | M | G ] ) ]
  [ attribute [...] ]
```

## Syntax Elements

### *n*

The number of bytes to allocate for the BLOB column. The maximum number of bytes is 2097088000, which is the default if *n* is not specified.

### K

*n* is specified in kilobytes (KB). When K is specified, *n* cannot exceed 2047937.

### M

*n* is specified in megabytes (Mb). When M is specified, *n* cannot exceed 1999.

### G

*n* is specified in gigabytes (GB). When G is specified, *n* must be 1.

### *attribute*

Appropriate data type, column storage, or column constraint attributes.

A BLOB column supports the following attributes:

- NOT NULL
- FORMAT
- TITLE

For more information on NOT NULL, see [Default Value Control Phrases](#). For details on FORMAT and TITLE, see [Data Type Formats and Format Phrases](#).

## Usage Notes

### External Representation

The following table lists the client representations for the Teradata SQL BLOB data type.

Define the length of the BLOB string as *k*, where  $0 \leq k \leq n$ .

Client CPU Architecture	Client Internal Data Format
IBM mainframe	Eight bytes (16-bit SMALLINT) length count <i>k</i> followed by <i>k</i> bytes of BYTE data.



Client CPU Architecture	Client Internal Data Format
<ul style="list-style-type: none"> <li>• UTS</li> <li>• RISC</li> <li>• Motorola 68000</li> <li>• WE 32000</li> <li>• Intel</li> </ul>	Eight bytes (16-bit SMALLINT) length count $k$ followed by $k$ bytes of BYTE data.

## Restrictions

A table can have a maximum of 32 LOB columns.

A LOB column cannot be a component of an index. Because of this restriction, a table must define at least one non-LOB column.

Queue tables cannot have BLOB columns.

## Functions That Operate on BLOBs

The following are some functions and operators that support BLOB types:

- BYTES
- Concatenation operator (||)
- SUBSTRING/SUBSTR
- TYPE
- Explicit data type conversion (CAST and Teradata conversion syntax)
- User-defined functions (UDFs)
- Stored procedures
- External stored procedures

## Examples

### Defining BLOB Data

The following example creates a table that defines a BLOB column named blarge:

```
CREATE TABLE t1
(id INTEGER
,blarge BLOB(128K));
```

### Inserting BLOB Data

The following example shows a stored procedure that inserts part of a BLOB into one table and the remaining part of the BLOB into another table:

```

CREATE TABLE LocalData(ld_ID INTEGER, ld_DATA BLOB);
CREATE TABLE GlobalData (gd_ID INTEGER, gd_DATA BLOB);

CREATE PROCEDURE DataSplitter(IN local_ID  INTEGER,
                              IN global_ID INTEGER,
                              IN all_DATA  BLOB)

BEGIN

    INSERT LocalData (local_ID, SUBSTRING(all_DATA FROM 1 FOR 128546));
    INSERT GlobalData (global_ID, SUBSTRING(all_DATA FROM 128547));

END;

```

## Related Information

FOR information on ...	SEE ...
functions and operators that support BLOB types	<i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145.
implementing UDFs and external stored procedures that operate on BLOBs	<i>Teradata Vantage™ - SQL External Routine Programming</i> , B035-1147.
implementing stored procedures that use BLOB local variables or parameters	<ul style="list-style-type: none"> <li>• <i>CREATE PROCEDURE</i> in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li> <li>• <i>Teradata Vantage™ - SQL Stored Procedures and Embedded SQL</i>, B035-1148.</li> </ul>

# LOB Functions

The following sections describe functions that are used with BLOB (Binary Large Object) and CLOB (Character Large Object) data types.

## EMPTY\_BLOB

Returns an empty BLOB (Binary Large Object), that is, one that contains 0 bytes.

EMPTY\_BLOB takes zero input arguments.

EMPTY\_BLOB is a scalar function whose return value data type is BLOB AS LOCATOR.

### ANSI Compliance

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

### Syntax

```
[TD_SYSFNLIB.] EMPTY_BLOB ()
```

### Syntax Elements

**TD\_SYSFNLIB.**

Name of the database where the function is located.

## Example

The following query:

```
UPDATE employee SET picture = EMPTY_BLOB();
```

updates the picture column with an empty BLOB.

## EMPTY\_CLOB

Returns an empty CLOB (Character Large Object), that is, one that contains 0 bytes.

EMPTY\_CLOB takes zero input arguments.

EMPTY\_CLOB is a scalar function whose return value data type is CLOB AS LOCATOR in the LATIN character set.

## ANSI Compliance

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

## Syntax

```
[TD_SYSFNLIB.] EMPTY_CLOB ()
```

## Syntax Elements

**TD\_SYSFNLIB.**

Name of the database where the function is located.

## Example

The following query:

```
UPDATE student SET essay = EMPTY_CLOB();
```

updates the essay column with an empty CLOB.

# DateTime and Interval Data Types

Teradata SQL supports DATE, TIME, TIMESTAMP, TIME WITH TIME ZONE, TIMESTAMP WITH TIME ZONE, and Interval data types for defining periods of time.

---

**Note:**

DATE, TIME, TIMESTAMP, TIME WITH TIME ZONE, and TIMESTAMP WITH TIME ZONE data types are collectively referred to as DateTime data types.

---

For information about Period data types, which are used to define a period with a beginning time and an end time, see [Period Data Types](#).

## DateTime Fields

The basic components of all DateTime and Interval data types are components called DateTime fields. All DateTime fields are represented internally as logical records of integer numbers, though the values are defined as type DateTime, not NUMERIC.

The components, ranked from most to least significant, are:

- YEAR
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND

## DateTime and Interval Data and Character Data Types

Because Unicode is supported, references to all character data lengths and format lengths should be interpreted as logical lengths and not as the number of bytes.

DateTime data exported to host processes are handled as character data. The physical size of values depends on the host character set.

## Time Zones

All TIME and TIMESTAMP data is associated with time zones either explicitly or implicitly. A time zone is represented by a signed displacement from Universal Coordinated Time (UTC). All TIME and TIMESTAMP values are stored internally as UTC by default, but you can use the TimeDateWZControl DBS Control field to specify that DateTime values without time zone information be stored in the database as system local time. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

TIME and TIMESTAMP values submitted for storage can be literals that include time zone information or they can be explicitly specified with syntax that provides time zone displacement information with the submitted value.

WHEN time zone values are defined ...	THEN ...
implicitly	the default time zone displacement of the SQL session is assigned to them.
explicitly	the submitted time zone displacement is stored with the values.

For information on defining session time zones, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

### Storage and Manipulation of Values in UTC Form

By default, Vantage converts all TIME and TIMESTAMP values to UTC prior to storing them. All operations, including hashing, collation, and comparisons that act on TIME and TIMESTAMP values are performed using their UTC forms.

For an example of hashing, consider the following TIMESTAMP literals:

```
TIMESTAMP '1999-07-01 15:00:00-08:00'
TIMESTAMP '1999-07-01 18:00:00-05:00'
```

Both values refer to the same time, which is expressed in UTC as follows:

```
TIMESTAMP '1999-07-01 23:00:00'
```

Because they are equal to one another, both literals hash identically.

For an example of collation, consider the following TIME WITH TIME ZONE literals:

```
TIME '08:00:00-08:00'
TIME '12:00:00-08:00'
TIME '15:00:00-08:00'
TIME '20:00:00-08:00'
```

The correct collation of these values is as follows:

```
TIME '20:00:00-08:00'
TIME '08:00:00-08:00'
TIME '12:00:00-08:00'
TIME '15:00:00-08:00'
```

This nonintuitive outcome becomes more apparent when the values are converted to UTC.

Local Time Value	UTC Time Value
TIME '08:00:00-08:00'	TIME '16:00:00'
TIME '12:00:00-08:00'	TIME '20:00:00'
TIME '15:00:00-08:00'	TIME '23:00:00'
TIME '20:00:00-08:00'	TIME '04:00:00'

As you can see, the TIME WITH TIME ZONE literal '20:00:00-08:00' is actually the lowest value in the sequence.

The counterargument to this explanation is that the time value '20:00:00-08:00' is really '04:00:00' the next day. This is a correct assessment; however, the TIME data type has no concept of day adjustment. Because of this property, you should consider using the TIMESTAMP data type for situations that might have outcomes like the one presented by this example.

For a detailed example of this behavior, see ORDER BY in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146, which contains an example where retrieving rows using an ORDER BY clause on a column with TIME data type may return unexpected results. The example also provides possible workarounds you can use to get the expected result.

## Implicit Time Zone Assignment

When no explicit time zone is specified for TIME or TIMESTAMP data, the time zone for the current session is assigned to the data by default.

The current time zone for a session is defined relative to UTC.

For example, Eastern Standard Time (EST) is five hours earlier than UTC, so EST is indicated by the signed value -05:00. Eastern Daylight Time (EDT) is only four hours earlier than UTC, so EDT is indicated by the signed value -04:00. European time is ahead of UTC by one hour, so it is represented by +01:00.

WHEN a TIME or TIMESTAMP column is defined this way ...	THEN information about its time zone is ...
WITH TIME ZONE	stored explicitly using the fields TIMEZONE_HOUR and TIMEZONE_MINUTE to indicate the offset applicable to the data.
without WITH TIME ZONE	not stored with the data, so it is not possible to know what time zone was in effect at the time the data was stored.

## Behavior of Time Values With UTC Offsets

Storing time values using UTC offsets results in the following standard behavior.

Suppose an installation is in the PST time zone and it is New Years Eve, 1998-12-31 20:30 local time.

The system TIMESTAMP WITH TIME ZONE for the indicated time is '1999-01-01 04:30-08:00' internally.

When you perform the `CURRENT_TIMESTAMP` function, it is in the form that includes `TIME ZONE`, and any external display converts the values into the appropriate values for the indicated time zone.

Should you want to return this value without the time zone, you could use `CAST` to convert the value to `TIMESTAMP` (without time zone). In PST, the result would be '1999-12-31 20:30 ', while the identical query performed with the time zone offset for EST returns the result '1999-12-31 23:30 '.

It is very important to realize that making a time zone adjustment can change the values for Year, Month, and Day fields when the result is displayed.

When timestamps are compared or used in an `ORDER BY` clause, a time zone adjustment does not change the comparison or ordering.

`TIME` adjustments to the time zone can also change what is displayed, just as `TIMESTAMP` adjustments do.

Note that the effect of adding or subtracting a time zone can change the comparison and ordering behavior because there are no higher order fields above the `HOUR` field to mark the crossover into what would be the previous (or next) day.

## Related Information

For more information on...	See...
setting session time zones	<code>SET TIME ZONE</code> , <code>CREATE USER</code> , <code>MODIFY USER</code> in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
setting the system time zone	<i>Teradata Vantage™ - Database Administration</i> , B035-1093.
using the <code>AT LOCAL</code> and <code>AT TIME ZONE</code> time zone specifiers to specify the time zone displacement in a <code>DateTime</code> expression	<i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145.
automatic adjustment of the system time to account for daylight saving time	<i>Teradata Vantage™ - Database Utilities</i> , B035-1102.
<ul style="list-style-type: none"> <li>defining whether <code>DateTime</code> values are stored in the database as UTC or as system local time</li> <li>specifying whether or not the built-in functions <code>CURRENT_TIME</code>, <code>CURRENT_TIMESTAMP</code>, <code>CURRENT_DATE</code>, <code>DATE</code>, and <code>TIME</code> reflect the session time and session time zone</li> </ul>	the <code>TimeDateWZControl</code> DBS Control field in <i>Teradata Vantage™ - Database Utilities</i> , B035-1102.

## Daylight Saving Time

Locales that observe Daylight Saving Time (DST) set their clocks ahead one hour for a portion of the year for DST, then return their clocks to standard time by setting the clocks back.

Vantage can automatically account for DST time switching through the use of time zone strings. Time zone strings allow the definition of named time zones for locales based on their time offset from UTC. For locales that observe DST, time zone strings can include rules that define when DST starts and ends. Vantage uses



this information to determine appropriate handling of time and date information, and to automatically adjust time zone offsets as appropriate for locales that observe DST.

For more information on...	See...
Time zone strings	<ul style="list-style-type: none"> <li>• <i>Teradata Vantage™ - Database Administration</i>, B035-1093</li> <li>• <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i>, B035-1145</li> <li>• <i>Teradata Vantage™ - Database Utilities</i>, B035-1102.</li> </ul>

## DATE Data Type

Identifies a field as a DATE value and simplifies handling and formatting of date variables.

### ANSI Compliance

DATE with a DateForm of ANSIDate is ANSI SQL:2011 compliant.

DATE with a DateForm of IntegerDate is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
DATE [ attributes [...] ]
```

### Syntax Elements

#### *attributes*

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Internal Representation of DATE

Internally, SQL Engine stores each DATE value as a four-byte signed integer using the following formula:

$$(\text{YEAR} - 1900) * 10000 + (\text{MONTH} * 100) + \text{DAY}$$

where the YEAR, MONTH, and DAY components, defined appropriately for the Gregorian calendar, have the following range of values.

Component	Range of Values	
	Minimum	Maximum
YEAR	1	9999

Range of Values		
Component	Minimum	Maximum
MONTH	1	12
DAY	1	28, 29, 30, or 31 (depending on the month and year)

## External Representation of DATE

The external DATE format for the client depends on whether the DateForm option is set to IntegerDate or ANSIDate mode.

IF the DateForm is ...	THEN ...
ANSIDate	the export data type is CHARACTER(10) in the client character set with format 'YYYY-MM-DD'.
IntegerDate	<p>the export data characteristics are as follows:</p> <ul style="list-style-type: none"> <li>• <i>Client CPU Architecture:</i> IBM mainframe Client Internal Data Format: Four byte 32-bit signed 2's complement integer, most significant byte first.</li> <li>• <i>Client CPU Architecture:</i> UTS, RISC, Motorola 68000, WE 32000 Client Internal Data Format: Four byte 32-bit signed 2's complement integer, most significant byte first.</li> <li>• <i>Client CPU Architecture:</i> Intel Client Internal Data Format: Four byte 32-bit signed 2's complement integer, least significant byte first.</li> </ul>

Determining the application definitions and client data types is the responsibility of the application programmer.

For an overview of the operations that set the DateForm option, see [Hierarchy of Date Formats](#).

## DATE Formats

To simplify your work with dates, Vantage assumes that you want dates in a preset format. Your database administrator sets that format for the system with the DateForm setting and the default date format setting in the specification for data formatting (SDF) file.

Those settings determine the following:

- Export data type of DATE values
- Data entry format for DATE values
- Display format of DATE values in macros being executed
- Date format for string-to-DATE comparisons and conversions
- Display format of DATE columns in newly created tables and views

You can change or override the DATE format settings in any of the following ways:

- Change the DateForm at the system level
- Set the DateForm at the user or session level
- Change the system default date format in the SDF
- Specify a format in a FORMAT phrase

For more information about the DATE formats and how to change them, see [Data Type Default Formats](#) and [DATE Formats](#).

## Entering Dates

You can specify dates in SQL statements in three ways:

- String
- Number
- ANSI date literal

The preferred way to specify dates is as an ANSI date literal. A valid ANSI date literal requires no additional format validation for SQL date operations. For example, the following SQL statement works regardless of the DATE format of the referenced column:

```
INSERT INTO DATETAB VALUES (DATE '2001-12-20');
```

Specifying the date as just a string requires matching the DATE format of the referenced column. SQL requires that date string comparisons have the same format. For example, the following statement would result in an error if the DATE format of the column DOB were 'YYYY-MM-DD':

```
SELECT Name FROM Employee WHERE DOB = 'Jan 31 1948';
```

For more information on the format characters that make up a valid DATE format, see [Formatting Characters for Date Information](#).

You can also specify a date as a number. A date as a number requires no format checking, but entering the number for years outside of the 1900's is not obvious and error prone. See the subsequent section about numeric date validation in the subsequent section.

The following is an example of 2001-12-20 entered as a number.

```
INSERT INTO DATETAB VALUES (1011220);
```

## String Date Validation

When converting a string into a date value, Vantage validates the components of the string as described in the following table.

Component	Requirement	Example
Date string - LATIN character set	Enclosed in apostrophes.	'Jan 28, 1960'
Date string - Unicode hexadecimal format	Enclosed in apostrophes and followed by xc.	'79797979C7 AF6D6DB7EE 6464C6FC'xc
Separator	Any non-numeric character can serve as a separator. Separators within a date do not have to match.	/
YY	The year as two or four numeric characters that are valid for the calendar. To support the century change transition, Vantage accepts four-digit years in two-digit year date formats.  <b>Note:</b> The Century Break feature determines the century of a two-digit year. For more information, see <i>Teradata Vantage™ - Database Utilities</i> , B035-1102.	05 2003
YYYY Y4	The year as four numeric characters that are valid for the calendar.	2003
MM	The month as two numeric characters that are valid for the calendar.	02
MMM M3	An abbreviated month name that matches one of the names specified by <i>ShortMonths</i> in the current Specification for Data Formatting (SDF) file.	Jan
MMMM M4	A full month name that matches one of the names specified by <i>LongMonths</i> in the current SDF.	January
DD	The day of the month as two numeric characters that are valid for the calendar.	03
DDD D3	The day of the year as three numeric characters that are valid for the calendar. If the date includes a month, the day must fall in that month.	056
EEE E3	An abbreviated day of the week name that matches one of the names specified by <i>ShortDays</i> in the current SDF.	Fri
EEEE E4	A day of the week name that matches one of the names specified by <i>LongDays</i> in the current SDF.	Friday

For more information on CHARACTER to DATE conversions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Numeric Date Validation

Although not recommended, you can enter dates as numbers in the database storage format. SQL Engine stores each DATE value as a four-byte integer using the following formula:

$$(\text{year} - 1900) * 10000 + (\text{month} * 100) + \text{day}$$

Allowable date values range from AD January 1, 0001 to AD December 31, 9999. For example, December 31, 1985 would be stored as the integer 851231; July 4, 1776 stored as -1239296; and March 30, 2041 stored as 1410330.

The Century Break feature does not affect numeric dates.

Vantage exports dates in this numeric format if the current DateForm setting is set to IntegerDate.

The following table demonstrates how numeric dates are interpreted when inserted into a column. Note the translation of the third date, which was probably intended to be 1990-12-01.

This raw date value ...	Translates to this value ...
901201	1990-12-01
1001201	2000-12-01
19901201	3890-12-01

Notice that this formula best fits two-digit dates in the 1900's. Because of the difficulty of using this format outside of the 1900's, we recommend specifying dates as ANSI date literals instead.

For more information on NUMERIC to DATE conversions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Implicit and Explicit DATE Conversion

Vantage performs implicit conversion on DATE types for certain operations such as assignment and comparison. This includes conversion from DATE to TIMESTAMP types in some cases. You can also use CAST to explicitly convert DATE to TIMESTAMP or other types.

For more information about all the cases in which implicit conversion of DATE is performed, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Arithmetic Operations on DATE

A field that is declared as DATE can be operated on using addition and subtraction. The MIN, MAX, AVERAGE, and COUNT aggregate operators can also be used with DATE values.

You can perform arithmetic operations on DATE data. For example, a number of days can be added to or subtracted from a DATE field, or from the DATE built-in value. The result of such operations is a value that is also a DATE data type.

The number of days between two dates can be calculated by subtracting one DATE value from another DATE value. The result is an integer.

Vantage handles month-to-month, year-to-year, and leap year arithmetic automatically; it does not store invalid dates, such as 1998-02-30.

In the following column definition, DOB is assigned the DATE data type.

```
DOB DATE FORMAT 'YYYY-MM-DD' NOT NULL
```

The following operations can be performed on data defined as DATE.

- Arithmetic (addition and subtraction)

Addition and subtraction of numeric types such as SMALLINT or INTEGER is still permitted, though these are non-ANSI operations because DATE is not a numeric data type in ANSI.

Such arithmetic operations are treated as if the numeric value had been typed as INTERVAL DAY.

For example, DATE - 12 is equivalent to the ANSI expression DATE - INTERVAL '12' DAY.

- Comparison

- =
- <>
- >
- ≥
- <
- ≤
- OVERLAPS

- Format conversion
- ADD\_MONTHS function
- EXTRACT function

For more information on arithmetic operations on DATE types, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Calendar functions and CALENDAR System View

Vantage provides a suite of calendar functions and the SYS\_CALENDAR.CALENDAR view to support DateTime operations that use calendar attributes. For example, the td\_day\_of\_month function returns the number of days from the beginning of the month to the specified date. The calendar functions provide better performance as compared to using the CALENDAR view to obtain similar results.

For more information about the calendar functions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

You can use the CALENDAR view to get attributes for any date between the years 1900 and 2100, such as which day of the week or which week of the month a date falls on.

You are encouraged to define views on the CALENDAR view because of its convenience. A useful view to define on CALENDAR is TODAY, defined as follows.

```
CREATE VIEW Today AS (
SELECT *
FROM SYS_CALENDAR.Calendar
WHERE SYS_CALENDAR.Calendar.calendar_date = DATE
);
```

CALENDAR permits easy specification of arithmetic expressions and aggregation. This is particularly useful in OLAP environments where it is common to request values aggregated by weeks, months, year-to-date, years, and so on.

For more information on the definition of the SYS\_CALENDAR.CALENDAR system view, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

## Examples

### Example: Queries Using Dates

To list all male employees currently over 50 years of age, the following statement might be entered:

```
SELECT name, dob
FROM employee
WHERE CURRENT_DATE > ADD_MONTHS (dob, 12*50)
AND sex = 'M' ;
```

The system returns:

Name	DOB
-----	-----
Russell S	1932-06-05
Carter J	1935-03-12
Inglis C	1938-03-07

Note that CURRENT\_DATE is used in the expression to get the current date from the system.

To project a date three months from the date of birth of employee Russell, enter:

```
SELECT name, ADD_MONTHS (dob,3)
FROM employee
WHERE name = 'Russell S' ;
```

The system returns:

Name	ADD_MONTHS(dob, 3)
-----	-----
Russell S	1932-09-05

**Note:**

Three months is not a specific number of days. In this particular case, “three months” is 92 days.

**Example: Using an Integer to Represent a Date**

To list employees who were born between March 7, 1938, and August 25, 1942, you can specify the date information as follows:

```
SELECT name, dob
FROM employee
WHERE dob BETWEEN 380307
AND DATE '1942-08-25'
ORDER BY dob ;
```

In this example, the first date (380307) is an integer representing *yymmdd*. The values of the dob column are converted to INTEGER to compare with the integer value. The second date is in the preferred DATE literal form. The result returns the date of birth information as specified for the Employee table:

Name	DOB
-----	-----
Inglis C	Mar 07 1938
Peterson J	Mar 27 1942

The display of DOB is controlled by the format for Personnel.Employee.DOB: FORMAT 'MMM DD YYYY'.

**Example: Changing the Date Format**

To change the date format displayed above to an alternate form, change the SELECT to:

```
SELECT name, dob (FORMAT '99-99-99')
FROM employee
WHERE dob BETWEEN 380307 AND DATE '1942-08-25'
ORDER BY dob ;
```

This format specification changes the display to the following:

Name	DOB
-----	-----



Inglis C	38-03-07
Peterson J	42-03-27

**Example: Changing From Date Format To Integer Format**

To change the display from date format to integer format, change the statement in Example 3 to:

```
SELECT name, dob (INTEGER)
FROM employee
WHERE dob BETWEEN 380307 AND 420825
ORDER BY dob ;
```

This format specification changes the display to the following:

Name	DOB
-----	-----
Inglis C	380307
Peterson J	420327

Further examples illustrating arithmetic operations on DATE appear in the following table. Assume the system date is Jan 24, 2001.

For this statement ...	The system returns this value ...
SELECT CURRENT_DATE;	Date ----- 01/01/24
SELECT CURRENT_DATE +3;	(Date+3) ----- 01/01/27
SELECT CURRENT_DATE -3;	(Date-3) ----- 01/01/21
SELECT CURRENT_DATE - CURRENT_DATE;	(Date-Date) ----- 0

**TIME Data Type**

Identifies a field as a TIME value.

This is not the value returned by the TIME function, which is formatted as a REAL number.

## ANSI Compliance

TIME is non-ANSI standard. SQL Engine stores a TIME value in UTC.

## Syntax

```
TIME [ ( fractional_seconds_precision ) ] [ attributes [...] ]
```

## Syntax Elements

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field.

Values for *fractional\_seconds\_precision* range from 0 to 6 inclusive.

The default precision is 6.

### *attributes*

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Internal Representation of TIME

Conceptually, TIME is treated as a record composed of three fields.

Field name	Range of Values		Storage Format
	Minimum	Maximum	
HOUR	00	23	BYTEINT
MINUTE	00	59	BYTEINT
SECOND	00.000000	61.999999 This value accounts for leap seconds that can be added to the clock.	DECIMAL(8,6)

Although the record is composed of numeric fields, it is not treated as a numeric value.

The length of the internal stored form is six bytes.

## External Representation of TIME

TIME types are imported and exported in record and indicator modes as CHARACTER data using the ANSI format string and the site-defined client character set.

WHEN <i>fractional_seconds_precision</i> is ...	THEN the type is ...	AND the format is ...
0	CHAR(8)	'hh:mi:ss'
<i>n</i> where <i>n</i> is 1 - 6	CHAR(9+ <i>n</i> )	'hh:mi:ss.ss...'

The following table shows examples of how TIME types are exported in record and indicator modes.

WHEN <i>fractional_seconds_precision</i> is ...	THEN the length is ...	FOR example ...
6	15	'11:37:58.123456'
0	8	'11:37:58'

## TIME Formats

For information about the TIME formats and how to change them, see [TIME and TIMESTAMP Formats](#).

## Implicit and Explicit TIME Conversion

Vantage performs implicit conversion from CHARACTER to TIME types during assignment and comparison. This conversion is supported for CHAR and VARCHAR types only. You cannot convert a character data type of CLOB or CHAR/VARCHAR CHARACTER SET GRAPHIC to TIME.

Vantage also performs implicit conversion from TIME to TIMESTAMP types in some cases. However, implicit TIME to TIMESTAMP conversion is not supported for comparisons.

You can use CAST to explicitly convert CHARACTER to TIME types or from TIME to TIMESTAMP types.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## TIMESTAMP Data Type

Identifies a field as a TIMESTAMP value.

## ANSI Compliance

TIMESTAMP is non-ANSI standard. SQL Engine stores a TIMESTAMP value in UTC.

## Syntax

```
TIMESTAMP [ ( fractional_seconds_precision ) ] [ attributes [...] ]
```

## Syntax Elements

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field.

Values for *fractional\_seconds\_precision* range from zero through six inclusive.

The default precision is six.

### *attributes*

Appropriate data type, column storage, or column constraint attributes.

For more information, see [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#).

## Usage Notes

### Internal Representation of TIMESTAMP

Conceptually, TIMESTAMP is treated as a record composed of six fields, defined appropriately for the Gregorian calendar.

Field Name	Minimum Value	Maximum Value	Storage Format
SECOND	00.000000	61.999999 This value accounts for leap seconds that can be added to the clock.	INTEGER (DECIMAL(8, 6))
YEAR	0001	9999	SMALLINT
MONTH	01	12	BYTEINT
DAY	01	28, 29, 30, or 31 (depending on the month and year)	BYTEINT
HOUR	00	23	BYTEINT
MINUTE	00	59	BYTEINT

Although the record is composed of numeric fields, it is not treated as a numeric value.

The length of the internal stored form is 10 bytes.

### External Representation of TIMESTAMP

TIMESTAMP types are imported and exported in record and indicator modes as CHARACTER data using the ANSI format string and the site-defined client character set.

WHEN <i>fractional_seconds_precision</i> is ...	THEN the type is ...	AND the format is ...
0	CHAR(19)	'yyyy-mm-dd hh:mi:ss'
<i>n</i> where <i>n</i> is 1 - 6	CHAR(20+ <i>n</i> )	'yyyy-mm-dd hh:mi:ss.ss...'

The following table shows examples of how TIMESTAMP types are exported in record and indicator modes.

WHEN <i>fractional_seconds_precision</i> is ...	THEN length is ...	FOR example ...
6	26	'1999-01-01 23:59:59.999999'
0	19	'1999-01-01 23:59:59'

## TIMESTAMP Formats

For information about the TIMESTAMP formats and how to change them, see [TIME and TIMESTAMP Formats](#).

## Implicit and Explicit TIMESTAMP Conversion

Vantage performs implicit conversion from CHARACTER to TIMESTAMP types during assignment and comparison. This conversion is supported for CHAR and VARCHAR types only. You cannot convert a character data type of CLOB or CHAR/VARCHAR CHARACTER SET GRAPHIC to TIMESTAMP.

Vantage also performs implicit conversions from TIMESTAMP to DATE and TIME types. However, implicit TIMESTAMP to TIME conversion is not supported for comparisons.

You can use CAST to explicitly convert CHARACTER to TIMESTAMP types and from TIMESTAMP to DATE or TIME types.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Examples

### Example: TIMESTAMP Data Type

The following example shows TIMESTAMP used in a query.

```
SELECT item, quantity, saletime
FROM sales
WHERE saletime > TIMESTAMP '2000-08-25 10:14:59'
AND saletime < TIMESTAMP '2000-08-25 10:30:01';
```

### Example: Difference Between Two TIMESTAMP Types

The difference between two TIMESTAMP types is an Interval type.

First define a table:

```
CREATE TABLE BillDateTime
(phone_no CHARACTER(10)
,start_time TIMESTAMP(0)
,end_time TIMESTAMP(0));
```

Now, determine the difference, specifying an Interval unit of DAY TO SECOND for the result:

```
SELECT (end_time - start_time) DAY(4) TO SECOND
FROM BillDateTime;
```

The DAY(4) specifies four digits of precision, and allows for a maximum of 9999 days, or approximately 27 years. The result looks like:

```
5 16:49:20.340000
```

### Example: Comparison Between Two TIMESTAMP Values

The following example compares two TIMESTAMP numbers to find out if they are within 30 minutes of each other.

First define a table:

```
CREATE TABLE PhoneTime
(phone_no CHARACTER(10)
,start_time TIMESTAMP(0)
,end_time TIMESTAMP(0));
```

Note that the difference between two TIMESTAMP types is an Interval type:

```
SELECT phone_no
FROM PhoneTime
WHERE (end_time - start_time) DAY(4) TO MINUTE > INTERVAL '30' MINUTE;
```

## TIME WITH TIME ZONE Data Type

Identifies a field as a TIME value with a displacement from UTC as defined for the system.

### ANSI Compliance

TIME WITH TIME ZONE is ANSI SQL:2011 compliant.

## Syntax

```
TIME [ ( fractional_seconds_precision ) ]
    WITH TIME ZONE [ attributes [...] ]
```

## Syntax Elements

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field.

Values for *fractional\_seconds\_precision* range from zero through six inclusive.

The default precision is six.

### *attributes*

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Internal Representation of TIME WITH TIME ZONE

Conceptually, TIME WITH TIME ZONE is treated as a record composed of five numeric fields.

Field Name	Minimum Value	Maximum Value	Storage Format
HOUR	00	23	BYTEINT
MINUTE	00	59	BYTEINT
SECOND	00.000000	61.999999 This value accounts for leap seconds that can be added to the clock.	DECIMAL(8,6)
TIMEZONE_HOUR	-12.59	+14.00	BYTEINT
TIMEZONE_MINUTE			BYTEINT

Although the record is composed of numeric fields, it is not treated as a numeric value.

The length of the internal stored form is eight bytes.

### External Representation of TIME WITH TIME ZONE

TIME WITH TIME ZONE types are imported and exported in record and indicator modes as CHARACTER data using the ANSI format string and the site-defined client character set.

WHEN <i>fractional_seconds_precision</i> is ...	THEN the type is ...	AND the format is ...
0	CHAR(14)	'hh:mi:ss-hh:mi' or 'hh:mi:ss+hh:mi'
<i>n</i> where <i>n</i> is 1 - 6	CHAR(15+ <i>n</i> )	'hh:mi:ss.ss...-hh:mi' or 'hh:mi:ss.ss...+hh:mi'

The following table shows examples of how TIME WITH TIME ZONE types are exported in record and indicator modes.

WHEN <i>fractional_seconds_precision</i> is ...	THEN length is ...	FOR example ...
6	21	'11:37:58.123456+08:00'
0	14	'11:37:58-08:00'

## TIME WITH TIME ZONE Format

For information about the TIME WITH TIME ZONE format and how to change it, see [TIME and TIMESTAMP Formats](#).

## Implicit and Explicit TIME WITH TIME ZONE Conversion

Vantage performs implicit conversion from CHARACTER to TIME WITH TIME ZONE types during assignment and comparison. This conversion is supported for CHAR and VARCHAR types only. You cannot convert a character data type of CLOB or CHAR/VARCHAR CHARACTER SET GRAPHIC to TIME WITH TIME ZONE.

Vantage also performs implicit conversion from TIME WITH TIME ZONE to TIMESTAMP types in some cases. However, implicit TIME WITH TIMEZONE to TIMESTAMP conversion is not supported for comparisons.

You can use CAST to explicitly convert CHARACTER to TIME WITH TIME ZONE types or from TIME WITH TIME ZONE to TIMESTAMP types.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## TIMESTAMP WITH TIME ZONE Data Type

Identifies a field as a TIMESTAMP value with a displacement from UTC as defined for the system.

## ANSI Compliance

TIMESTAMP WITH TIME ZONE is ANSI SQL:2011 compliant.



## Syntax

```
TIMESTAMP [ ( fractional_seconds_precision ) ]
  WITH TIME ZONE [ attributes [...] ]
```

## Syntax Elements

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field.

Values for *fractional\_seconds\_precision* range from zero through six inclusive.

The default precision is six.

### *attributes*

Appropriate data type, column storage, or column constraint attributes.

For specific information, see [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#).

## Usage Notes

### Internal Representation of **TIMESTAMP WITH TIME ZONE**

Conceptually, **TIMESTAMP WITH TIME ZONE** is treated as a record composed of eight fields, defined appropriately for the Gregorian calendar.

Field Name	Minimum Value	Maximum Value	Storage Format
SECOND	00.000000	61.999999 This value accounts for leap seconds that can be added to the clock.	INTEGER(DECIMAL(8,6))
YEAR	0001	9999	SMALLINT
MONTH	01	12	BYTEINT
DAY	01	28, 29, 30, or 31 (depending on the month and year)	BYTEINT
HOUR	00	23	BYTEINT
MINUTE	00	59	BYTEINT
TIMEZONE_HOUR	-12.59	+14.00	BYTEINT
TIMEZONE_MINUTE			BYTEINT

Although the record is composed of numeric fields, it is not treated as a numeric value.

The length of the internal stored form is 12 bytes.

### External Representation of TIMESTAMP WITH TIME ZONE

TIMESTAMP WITH TIME ZONE types are imported and exported in record and indicator modes as CHARACTER data using the ANSI format string and the site-defined client character set.

WHEN <i>fractional_seconds_precision</i> is ...	THEN the type is ...	AND the format is ...
0	CHAR(25)	'yyyy-mm-dd hh:mi:ss-hh:mi' or 'yyyy-mm-dd hh:mi:ss+hh:mi'
<i>n</i> where <i>n</i> is 1 - 6	CHAR(26+ <i>n</i> )	'yyyy-mm-dd hh:mi:ss.ss...-hh:mi' or 'yyyy-mm-dd hh:mi:ss.ss...+hh:mi'

The following table shows examples of how TIMESTAMP WITH TIME ZONE types are exported in record and indicator modes.

WHEN <i>fractional_seconds_precision</i> is ...	THEN length is ...	FOR example ...
6	32	'2000-01-01 11:37:58.123456+08:00'
0	25	'2000-01-01 11:37:58-08:00'

### TIMESTAMP WITH TIME ZONE Format

For information about the TIMESTAMP WITH TIME ZONE format and how to change it, see [TIME and TIMESTAMP Formats](#).

### Implicit and Explicit TIMESTAMP WITH TIME ZONE Conversion

Vantage performs implicit conversion from CHARACTER to TIMESTAMP WITH TIME ZONE types during assignment and comparison. This conversion is supported for CHAR and VARCHAR types only. You cannot convert a character data type of CLOB or CHAR/VARCHAR CHARACTER SET GRAPHIC to TIMESTAMP WITH TIME ZONE.

Vantage also performs implicit conversion from TIMESTAMP WITH TIME ZONE to TIME and DATE types in some cases. However, implicit TIMESTAMP WITH TIMEZONE to TIME conversion is not supported for comparisons.

You can use CAST to explicitly convert CHARACTER to TIMESTAMP WITH TIME ZONE types and from TIMESTAMP WITH TIME ZONE to TIME or DATE types.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## INTERVAL YEAR Data Type

Identifies a field as an INTERVAL value defining a period of time in years.

### ANSI Compliance

INTERVAL YEAR is ANSI SQL:2011 compliant.

### Syntax

```
INTERVAL YEAR [ ( precision ) ] [ attributes [...] ]
```

### Syntax Elements

#### *precision*

The permitted range of digits for YEAR, ranging from 1 to 4.

The default is 2.

For example, if you specify a precision of 4, the range for YEAR is '-9999' through '9999'.

#### *attributes*

Appropriate data type, column storage, or column constraint attributes.

For specific information, see [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#).

## Usage Notes

### Internal Representation of INTERVAL YEAR

Storage Format	Length
SMALLINT	2 bytes

### External Representation of INTERVAL YEAR

INTERVAL YEAR types are imported and exported in record and indicator modes as CHARACTER data using the client character set.

Type	Format
CHARACTER(precision + 1)	'-y( <i>precision</i> )'

The value includes a leading blank or minus sign followed by the indicated number of digits. An example value for INTERVAL YEAR(3) is '-125'.

## Range of Values

The range of values for INTERVAL YEAR is as follows.

Type and Precision	Minimum Value	Maximum Value
INTERVAL YEAR(1)	-'9'	'9'
INTERVAL YEAR(2)	-'99'	'99'
INTERVAL YEAR(3)	-'999'	'999'
INTERVAL YEAR(4)	-'9999'	'9999'

### Note:

Decimal values are not allowed for Interval data types except for second intervals.

## Implicit and Explicit INTERVAL YEAR Conversion

Vantage performs implicit conversion from one Interval data type to another Interval type in some cases. You can also use CAST to explicitly convert one Interval type to another.

Conversions are possible only within the same INTERVAL family. For example, you may convert a YEAR interval to months, but not to days or hours.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## INTERVAL YEAR TO MONTH Data Type

Identifies a field as an INTERVAL value defining a period of time in years and months.

### ANSI Compliance

INTERVAL YEAR TO MONTH is ANSI SQL:2011 compliant.

### Syntax

```
INTERVAL YEAR [ ( precision ) ] TO MONTH [ attributes [...] ]
```

### Syntax Elements

#### *precision*

The permitted range of digits for YEAR, ranging from one to four.

The default is two.

#### *attributes*

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Internal Representation of INTERVAL YEAR TO MONTH

Conceptually, INTERVAL YEAR TO MONTH is a record with two fields.

Field Name	Storage Format	Total Length
YEAR	SMALLINT	4 bytes
MONTH	SMALLINT	

### External Representation of INTERNAL YEAR TO MONTH

INTERVAL YEAR TO MONTH types are imported and exported in record and indicator modes as CHARACTER data using the client character set.

Type	Format
CHARACTER(precision + 4)	'-y( <i>precision</i> )-mm'

An example value for INTERVAL YEAR(3) TO MONTH is ' 013-11'.

### Range of Values

The range of values for the INTERVAL YEAR TO MONTH is as follows.

Type and Precision	Minimum Value	Maximum Value
INTERVAL YEAR(1) TO MONTH	'-9-11'	'9-11'
INTERVAL YEAR(2) TO MONTH	'-99-11'	'99-11'
INTERVAL YEAR(3) TO MONTH	'-999-11'	'999-11'
INTERVAL YEAR(4) TO MONTH	'-9999-11'	'9999-11'

#### Note:

Decimal values are not allowed for Interval data types except for second intervals.

### Implicit and Explicit INTERVAL YEAR TO MONTH Conversion

Vantage performs implicit conversion from one Interval data type to another Interval type in some cases. You can also use CAST to explicitly convert one Interval type to another.

Conversions are possible only within the same INTERVAL family. For example, you may convert a YEAR interval to months, but not to days or hours.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Example

The following statement creates a table with an INTERVAL YEAR TO MONTH column:

```
CREATE TABLE TimeInfo
  (Id INTEGER
   ,Offset INTERVAL YEAR (4) TO MONTH);
```

## INTERVAL MONTH Data Type

Identifies a field as an INTERVAL value defining a period of time in months.

### ANSI Compliance

INTERVAL MONTH is ANSI SQL:2011 compliant.

### Syntax

```
INTERVAL MONTH [ ( precision ) ] [ attributes [...] ]
```

### Syntax Elements

#### *precision*

The permitted range of digits for MONTH, ranging from one to four.

The default is two.

#### *attributes*

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Internal Representation of INTERVAL MONTH

Storage Format	Length
SMALLINT	2 bytes

## External Representation of INTERVAL MONTH

INTERVAL MONTH types are imported and exported in record and indicator modes as CHARACTER data using the client character set.

Type	Format
CHARACTER(precision + 1)	'-m( <i>precision</i> )'

For example, for INTERVAL MONTH(4), the value might be something like ' 1300'.

## Range of Values

The range of values for INTERVAL MONTH is as follows.

Type and Precision	Minimum Value	Maximum Value
INTERVAL MONTH(1)	'-9'	'9'
INTERVAL MONTH(2)	'-99'	'99'
INTERVAL MONTH(3)	'-999'	'999'
INTERVAL MONTH(4)	'-9999'	'9999'

### Note:

Decimal values are not allowed for Interval data types except for second intervals.

## Implicit and Explicit INTERVAL MONTH Conversion

Vantage performs implicit conversion from one Interval data type to another Interval type in some cases. You can also use CAST to explicitly convert one Interval type to another.

Conversions are possible only within the same INTERVAL family. For example, you may convert a YEAR interval to months, but not to days or hours.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Example

The following statement creates a table with an INTERVAL MONTH column:

```
CREATE TABLE TimeInfo
  (Id INTEGER
   ,Offset INTERVAL MONTH (4));
```

## INTERVAL DAY Data Type

Identifies a field as an INTERVAL value defining a period of time in days.

### ANSI Compliance

INTERVAL DAY is ANSI SQL:2011 compliant.

### Syntax

```
INTERVAL DAY [ ( precision ) ] [ attributes [...] ]
```

### Syntax Elements

#### *precision*

The permitted range of digits for DAY, ranging from one to four.

The default is two.

#### *attributes*

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Internal Representation of INTERVAL DAY

Storage Format	Length
SMALLINT	2 bytes

### External Representation of INTERVAL DAY

INTERVAL DAY types are imported and exported in record and indicator modes as CHARACTER data using the client character set.

Type	Format
CHARACTER( <i>precision</i> + 1)	'-d( <i>precision</i> )'

For example, for INTERVAL DAY(2), the value might be something like '-39'.

### Range of Values

The range of values for INTERVAL DAY is as follows.



Type and Precision	Minimum Value	Maximum Value
INTERVAL DAY(1)	-'9'	'9'
INTERVAL DAY(2)	-'99'	'99'
INTERVAL DAY(3)	-'999'	'999'
INTERVAL DAY(4)	-'9999'	'9999'

**Note:**

Decimal values are not allowed for Interval data types except for second intervals.

**Implicit and Explicit INTERVAL DAY Conversion**

Vantage performs implicit conversion from one Interval data type to another Interval type in some cases. You can also use CAST to explicitly convert one Interval type to another.

Conversions are possible only within the same INTERVAL family. For example, you may convert a YEAR interval to months, but not to days or hours.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

**Example**

The following statement creates a table with an INTERVAL DAY column:

```
CREATE TABLE TimeInfo
  (Id INTEGER
   ,Offset INTERVAL DAY (4));
```

**INTERVAL DAY TO HOUR Data Type**

Identifies a field as an INTERVAL value defining a period of time in days and hours.

**ANSI Compliance**

INTERVAL DAY TO HOUR is ANSI SQL:2011 compliant.

**Syntax**

```
INTERVAL DAY [ ( precision ) ] TO HOUR [ attributes [...] ]
```

## Syntax Elements

### *precision*

The permitted range of digits for DAY, ranging from one to four.

The default is two.

### *attributes*

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Internal Representation of INTERVAL DAY TO HOUR

Conceptually, INTERVAL DAY TO HOUR is a record with the following fields.

Field Name	Storage Format	Total Length
DAY	SMALLINT	4 bytes
HOURL	SMALLINT	

### External Representation of INTERVAL DAY TO HOUR

INTERVAL DAY TO HOUR types are imported and exported in record and indicator modes as CHARACTER data using the client character set.

Type	Format
CHARACTER(precision + 4)	'-d( <i>precision</i> ) hh'

An example value of INTERVAL DAY (2) TO HOUR is ' 17 11'.

### Range of Values

The range of values for INTERVAL DAY TO HOUR is as follows.

Type and Precision	Minimum Value	Maximum Value
INTERVAL DAY(1) TO HOUR	'-9 23'	'9 23'
INTERVAL DAY(2) TO HOUR	'-99 23'	'99 23'
INTERVAL DAY(3) TO HOUR	'-999 23'	'999 23'
INTERVAL DAY(4) TO HOUR	'-9999 23'	'9999 23'

**Note:**

Decimal values are not allowed for Interval data types except for second intervals.

**Implicit and Explicit INTERVAL DAY TO HOUR Conversion**

Vantage performs implicit conversion from one Interval data type to another Interval type in some cases. You can also use CAST to explicitly convert one Interval type to another.

Conversions are possible only within the same INTERVAL family. For example, you may convert a YEAR interval to months, but not to days or hours.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

**Example**

The following statement creates a table with an INTERVAL DAY TO HOUR column:

```
CREATE TABLE TimeInfo
  (Id INTEGER
   ,Offset INTERVAL DAY (4) TO HOUR);
```

**INTERVAL DAY TO MINUTE Data Type**

Identifies a field as an INTERVAL value defining a period of time in days, hours, and minutes.

**ANSI Compliance**

INTERVAL DAY TO MINUTE is ANSI SQL:2011 compliant.

**Syntax**

```
INTERVAL DAY [ ( precision ) ] TO MINUTE [ attributes [...] ]
```

**Syntax Elements*****precision***

The permitted range of digits for DAY, ranging from one to four.

The default is two.

HOUR is always two digits and MINUTE is always two digits.

***attributes***

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Internal Representation of INTERVAL DAY TO MINUTE

Conceptually, INTERVAL DAY TO MINUTE is a record that has the following fields.

Field Name	Storage Format	Total Length
DAY	SMALLINT	8 bytes (with 2 pad bytes)
HOURL	SMALLINT	
MINUTE	SMALLINT	

### External Representation of INTERVAL DAY TO MINUTE

INTERVAL DAY TO MINUTE types are imported and exported in record and indicator modes as CHARACTER data using the client character set.

Type	Format
CHARACTER(precision + 7)	'-d( <i>precision</i> ) hh:mm'

For example, for INTERVAL DAY (3) TO MINUTE, the value might be something like '-127 12:37'.

### Range of Values

The range of values for INTERVAL DAY TO MINUTE is as follows.

Type and Precision	Minimum Value	Maximum Value
INTERVAL DAY(1) TO MINUTE	'-9 23:59'	'9 23:59'
INTERVAL DAY(2) TO MINUTE	'-99 23:59'	'99 23:59'
INTERVAL DAY(3) TO MINUTE	'-999 23:59'	'999 23:59'
INTERVAL DAY(4) TO MINUTE	'-9999 23:59'	'9999 23:59'

#### Note:

Decimal values are not allowed for Interval data types except for second intervals.

### Implicit and Explicit INTERVAL DAY TO MINUTE Conversion

Vantage performs implicit conversion from one Interval data type to another Interval type in some cases. You can also use CAST to explicitly convert one Interval type to another.

Conversions are possible only within the same INTERVAL family. For example, you may convert a YEAR interval to months, but not to days or hours.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Example

The following statement creates a table with an INTERVAL DAY TO MINUTE column:

```
CREATE TABLE TimeInfo
  (Id INTEGER
   ,Offset INTERVAL DAY (4) TO MINUTE);
```

## INTERVAL DAY TO SECOND Data Type

Identifies a field as an INTERVAL value defining a period of time in days, hours, minutes, and seconds.

### ANSI Compliance

INTERVAL DAY TO SECOND is ANSI SQL:2011 compliant.

### Syntax

```
INTERVAL DAY [ ( precision ) ] TO SECOND
  [ ( fractional_seconds_precision ) ] [ attributes [...] ]
```

### Syntax Elements

#### *precision*

The permitted range of digits for DAY, ranging from one to four.

The default is two.

#### *fractional\_seconds\_precision*

The fractional precision for the values of SECOND.

The default is six.

#### *attributes*

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Internal Representation of INTERVAL DAY TO SECOND

Conceptually, INTERVAL DAY TO SECOND is a record that has the following fields.

Field Name	Storage Format	Total Length
DAY	SMALLINT	10 bytes
HOUR	SMALLINT	
MINUTE	SMALLINT	
SECOND	INTEGER	

### External Representation of INTERVAL DAY TO SECOND

INTERVAL DAY TO SECOND types are imported and exported in record and indicator modes as CHARACTER data using the client character set.

WHEN <i>fractional_seconds_precision</i> is ...	THEN the type and format are ...
0	<b>Type:</b> CHARACTER(precision + 10)
	<b>Format:</b> '-d( <i>precision</i> ) hh:mm:ss'
<i>n</i> where <i>n</i> is 1 - 6	<b>Type:</b> CHARACTER(precision + <i>n</i> + 11)
	<b>Format:</b> '-d( <i>precision</i> ) hh:mm:ss.s( <i>n</i> )'

For example, for INTERVAL DAY (4) TO SECOND(3), the value might be something like ' 125 12:27:31.125'.

The DAY field has two leading blanks. The first is for a sign, which is not present for positive intervals. The second is to preserve the fixed size of the DAY field at four digits.

### Range of Values

The range of values for INTERVAL DAY TO SECOND is as follows.

Type, Precision, and Fractional Seconds Precision	Minimum Value	Maximum Value
INTERVAL DAY(1) TO SECOND(0)	'-9 23:59:59'	'9 23:59:59'
INTERVAL DAY(1) TO SECOND(1)	'-9 23:59:59.9'	'9 23:59:59.9'
INTERVAL DAY(1) TO SECOND(2)	'-9 23:59:59.99'	'9 23:59:59.99'
INTERVAL DAY(1) TO SECOND(3)	'-9 23:59:59.999'	'9 23:59:59.999'

Type, Precision, and Fractional Seconds Precision	Minimum Value	Maximum Value
INTERVAL DAY(1) TO SECOND(4)	-'9 23:59:59.9999'	'9 23:59:59.9999'
INTERVAL DAY(1) TO SECOND(5)	-'9 23:59:59.99999'	'9 23:59:59.99999'
INTERVAL DAY(1) TO SECOND(6)	-'9 23:59:59.999999'	'9 23:59:59.999999'
INTERVAL DAY(2) TO SECOND(0)	-'99 23:59:59'	'99 23:59:59'
INTERVAL DAY(2) TO SECOND(1)	-'99 23:59:59.9'	'99 23:59:59.9'
INTERVAL DAY(2) TO SECOND(2)	-'99 23:59:59.99'	'99 23:59:59.99'
INTERVAL DAY(2) TO SECOND(3)	-'99 23:59:59.999'	'99 23:59:59.999'
INTERVAL DAY(2) TO SECOND(4)	-'99 23:59:59.9999'	'99 23:59:59.9999'
INTERVAL DAY(2) TO SECOND(5)	-'99 23:59:59.99999'	'99 23:59:59.99999'
INTERVAL DAY(2) TO SECOND(6)	-'99 23:59:59.999999'	'99 23:59:59.999999'
INTERVAL DAY(3) TO SECOND(0)	-'999 23:59:59'	'999 23:59:59'
INTERVAL DAY(3) TO SECOND(1)	-'999 23:59:59.9'	'999 23:59:59.9'
INTERVAL DAY(3) TO SECOND(2)	-'999 23:59:59.99'	'999 23:59:59.99'
INTERVAL DAY(3) TO SECOND(3)	-'999 23:59:59.999'	'999 23:59:59.999'
INTERVAL DAY(3) TO SECOND(4)	-'999 23:59:59.9999'	'999 23:59:59.9999'
INTERVAL DAY(3) TO SECOND(5)	-'999 23:59:59.99999'	'999 23:59:59.99999'
INTERVAL DAY(3) TO SECOND(6)	-'999 23:59:59.999999'	'999 23:59:59.999999'
INTERVAL DAY(4) TO SECOND(0)	-'9999 23:59:59'	'9999 23:59:59'
INTERVAL DAY(4) TO SECOND(1)	-'9999 23:59:59.9'	'9999 23:59:59.9'
INTERVAL DAY(4) TO SECOND(2)	-'9999 23:59:59.99'	'9999 23:59:59.99'
INTERVAL DAY(4) TO SECOND(3)	-'9999 23:59:59.999'	'9999 23:59:59.999'
INTERVAL DAY(4) TO SECOND(4)	-'9999 23:59:59.9999'	'9999 23:59:59.9999'
INTERVAL DAY(4) TO SECOND(5)	-'9999 23:59:59.99999'	'9999 23:59:59.99999'
INTERVAL DAY(4) TO SECOND(6)	-'9999 23:59:59.999999'	'9999 23:59:59.999999'

### Implicit and Explicit INTERVAL DAY TO SECOND Conversion

Vantage performs implicit conversion from one Interval data type to another Interval type in some cases. You can also use CAST to explicitly convert one Interval type to another.

Conversions are possible only within the same INTERVAL family. For example, you may convert a YEAR interval to months, but not to days or hours.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Example

The following statement creates a table with an INTERVAL DAY TO SECOND column:

```
CREATE TABLE TimeInfo
  (Id INTEGER
   ,Offset INTERVAL DAY (4) TO SECOND (4));
```

## INTERVAL HOUR Data Type

Identifies a field as an INTERVAL value defining a period of time in hours.

### ANSI Compliance

INTERVAL HOUR is ANSI SQL:2011 compliant.

### Syntax

```
INTERVAL HOUR [ ( precision ) ] [ attributes [...] ]
```

### Syntax Elements

#### *precision*

The permitted range of digits for HOUR, ranging from one to four.

The default is two.

#### *attributes*

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Internal Representation of INTERVAL HOUR

Storage Format	Length
SMALLINT	2 bytes



## External Representation of INTERVAL HOUR

INTERVAL HOUR types are imported and exported in record and indicator modes as CHARACTER data using the client character set.

Type	Format
CHARACTER(precision + 1)	'-h( <i>precision</i> )'

For example, for INTERVAL HOUR(1), the value might be something like '-5'.

## Range of Values

The range of values for INTERVAL HOUR is as follows.

Type and Precision	Minimum Value	Maximum Value
INTERVAL HOUR(1)	-'9'	'9'
INTERVAL HOUR(2)	-'99'	'99'
INTERVAL HOUR(3)	-'999'	'999'
INTERVAL HOUR(4)	-'9999'	'9999'

### Note:

Decimal values are not allowed for Interval data types except for second intervals.

## Implicit and Explicit INTERVAL HOUR Conversion

Vantage performs implicit conversion from one Interval data type to another Interval type in some cases. You can also use CAST to explicitly convert one Interval type to another.

Conversions are possible only within the same INTERVAL family. For example, you may convert a YEAR interval to months, but not to days or hours.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## INTERVAL HOUR TO MINUTE Data Type

Identifies a field as an INTERVAL value defining a period of time in hours and minutes.

## ANSI Compliance

INTERVAL HOUR TO MINUTE is ANSI SQL:2011 compliant.

## Syntax

```
INTERVAL HOUR [ ( precision ) ] TO MINUTE [ attributes [...] ]
```

## Syntax Elements

### *precision*

The permitted range of digits for HOUR, ranging from one to four.

The default is two.

### *attributes*

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Internal Representation of INTERVAL HOUR TO MINUTE

Field	Storage Format	Total Length
HOUR	SMALLINT	4 bytes
MINUTE	SMALLINT	

### External Representation of INTERVAL HOUR TO MINUTE

INTERVAL HOUR TO MINUTE types are imported and exported in record and indicator modes as CHARACTER data using the client character set.

Type	Format
CHARACTER( <i>precision</i> + 4)	'-h( <i>precision</i> ):mm'

For example, for INTERVAL HOUR(2) TO MINUTE, a value might be ' 17:37 '.

### Range of Values

The range of values for INTERVAL HOUR TO MINUTE is as follows.

Type and Precision	Minimum Value	Maximum Value
INTERVAL HOUR(1) TO MINUTE	'-9:59'	'9:59'
INTERVAL HOUR(2) TO MINUTE	'-99:59'	'99:59'

Type and Precision	Minimum Value	Maximum Value
INTERVAL HOUR(3) TO MINUTE	-'999:59'	'999:59'
INTERVAL HOUR(4) TO MINUTE	-'9999:59'	'9999:59'

**Note:**

Decimal values are not allowed for Interval data types except for second intervals.

**Implicit and Explicit INTERVAL HOUR TO MINUTE Conversion**

Vantage performs implicit conversion from one Interval data type to another Interval type in some cases. You can also use CAST to explicitly convert one Interval type to another.

Conversions are possible only within the same INTERVAL family. For example, you may convert a YEAR interval to months, but not to days or hours.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

**Example**

The following statement creates a table with an INTERVAL HOUR TO MINUTE column:

```
CREATE TABLE TimeInfo
  (Id INTEGER
   ,Offset INTERVAL HOUR (4) TO MINUTE);
```

**INTERVAL HOUR TO SECOND Data Type**

Identifies a field as an INTERVAL value defining a period of time in hours, minutes, and seconds.

**ANSI Compliance**

INTERVAL HOUR TO SECOND is ANSI SQL:2011 compliant.

**Syntax**

```
INTERVAL HOUR [ ( precision ) ] TO SECOND
  [ ( fractional_seconds_precision ) ] [ attributes [...] ]
```

**Syntax Elements*****precision***

The permitted range of digits for HOUR, ranging from one to four.

The default is two.

#### ***fractional\_seconds\_precision***

The fractional precision for the values of SECOND, ranging from zero to six.

The default is six.

#### ***attributes***

Appropriate data type, column storage, or column constraint attributes.

For specific information, see [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#).

## Usage Notes

### **Internal Representation of INTERVAL HOUR TO SECOND**

Conceptually, INTERVAL HOUR TO SECOND has the following fields.

Field Name	Storage Format	Total Length
HOUR	SMALLINT	8 bytes
MINUTE	SMALLINT	
SECOND	INTEGER	

### **External Representation of INTERVAL HOUR TO SECOND**

INTERVAL HOUR TO SECOND types are imported and exported in record and indicator modes as CHARACTER data using the client character set.

WHEN <i>fractional_seconds_precision</i> is ...	THEN the type and format are ...
0	<b>Type:</b> CHARACTER(precision + 7) <b>Format:</b> '-h( <i>precision</i> ):mm:ss'
<i>n</i> where <i>n</i> is 1 - 6	<b>Type:</b> CHARACTER(precision + <i>n</i> + 8) <b>Format:</b> '-h( <i>precision</i> ):mm:ss.s( <i>n</i> )'

For example, for INTERVAL HOUR(2) TO SECOND(2), the value might be something like '-13:27:58.17'.

### **Range of Values**

The range of values for INTERVAL HOUR TO SECOND is as follows.

Type, Precision, and Fractional Seconds Precision	Minimum Value	Maximum Value
INTERVAL HOUR(1) TO SECOND(0)	'-9:59:59'	'9:59:59'
INTERVAL HOUR(1) TO SECOND(1)	'-9:59:59.9'	'9:59:59.9'
INTERVAL HOUR(1) TO SECOND(2)	'-9:59:59.99'	'9:59:59.99'
INTERVAL HOUR(1) TO SECOND(3)	'-9:59:59.999'	'9:59:59.999'
INTERVAL HOUR(1) TO SECOND(4)	'-9:59:59.9999'	'9:59:59.9999'
INTERVAL HOUR(1) TO SECOND(5)	'-9:59:59.99999'	'9:59:59.99999'
INTERVAL HOUR(1) TO SECOND(6)	'-9:59:59.999999'	'9:59:59.999999'
INTERVAL HOUR(2) TO SECOND(0)	'-99:59:59'	'99:59:59'
INTERVAL HOUR(2) TO SECOND(1)	'-99:59:59.9'	'99:59:59.9'
INTERVAL HOUR(2) TO SECOND(2)	'-99:59:59.99'	'99:59:59.99'
INTERVAL HOUR(2) TO SECOND(3)	'-99:59:59.999'	'99:59:59.999'
INTERVAL HOUR(2) TO SECOND(4)	'-99:59:59.9999'	'99:59:59.9999'
INTERVAL HOUR(2) TO SECOND(5)	'-99:59:59.99999'	'99:59:59.99999'
INTERVAL HOUR(2) TO SECOND(6)	'-99:59:59.999999'	'99:59:59.999999'
INTERVAL HOUR(3) TO SECOND(0)	'-999:59:59'	'999:59:59'
INTERVAL HOUR(3) TO SECOND(1)	'-999:59:59.9'	'999:59:59.9'
INTERVAL HOUR(3) TO SECOND(2)	'-999:59:59.99'	'999:59:59.99'
INTERVAL HOUR(3) TO SECOND(3)	'-999:59:59.999'	'999:59:59.999'
INTERVAL HOUR(3) TO SECOND(4)	'-999:59:59.9999'	'999:59:59.9999'
INTERVAL HOUR(3) TO SECOND(5)	'-999:59:59.99999'	'999:59:59.99999'
INTERVAL HOUR(3) TO SECOND(6)	'-999:59:59.999999'	'999:59:59.999999'
INTERVAL HOUR(4) TO SECOND(0)	'-9999:59:59'	'9999:59:59'
INTERVAL HOUR(4) TO SECOND(1)	'-9999:59:59.9'	'9999:59:59.9'
INTERVAL HOUR(4) TO SECOND(2)	'-9999:59:59.99'	'9999:59:59.99'
INTERVAL HOUR(4) TO SECOND(3)	'-9999:59:59.999'	'9999:59:59.999'
INTERVAL HOUR(4) TO SECOND(4)	'-9999:59:59.9999'	'9999:59:59.9999'
INTERVAL HOUR(4) TO SECOND(5)	'-9999:59:59.99999'	'9999:59:59.99999'
INTERVAL HOUR(4) TO SECOND(6)	'-9999:59:59.999999'	'9999:59:59.999999'

## Implicit and Explicit INTERVAL HOUR TO SECOND Conversion

Vantage performs implicit conversion from one Interval data type to another Interval type in some cases. You can also use CAST to explicitly convert one Interval type to another.

Conversions are possible only within the same INTERVAL family. For example, you may convert a YEAR interval to months, but not to days or hours.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## INTERVAL MINUTE Data Type

Identifies a field as an INTERVAL value defining a period of time in minutes.

### ANSI Compliance

INTERVAL MINUTE is ANSI SQL:2011 compliant.

### Syntax

```
INTERVAL MINUTE [ ( precision ) ] [ attributes [...] ]
```

### Syntax Elements

#### *precision*

The permitted range of digits for MINUTE, ranging from one to four.

The default is two.

#### *attributes*

Appropriate data type, column storage, or column constraint attributes.

For specific information, see [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#).

## Usage Notes

### Internal Representation of INTERVAL MINUTE

Storage Format	Total Length
SMALLINT	2 bytes

### External Representation of INTERVAL MINUTE

INTERVAL MINUTE types are imported and exported in record and indicator modes as CHARACTER data using the client character set.

Type	Format
CHARACTER(precision + 1)	'-m( <i>precision</i> )'

For example, for INTERVAL MINUTE(2), the value might be something like ' 49 '.

## Range of Values

The range of values for INTERVAL MINUTE is as follows.

Type and Precision	Minimum Value	Maximum Value
INTERVAL MINUTE(1)	'-9'	'9'
INTERVAL MINUTE(2)	'-99'	'99'
INTERVAL MINUTE(3)	'-999'	'999'
INTERVAL MINUTE(4)	'-9999'	'9999'

### Note:

Decimal values are not allowed for Interval data types except for second intervals.

## Implicit and Explicit INTERVAL MINUTE Conversion

Vantage performs implicit conversion from one Interval data type to another Interval type in some cases. You can also use CAST to explicitly convert one Interval type to another.

Conversions are possible only within the same INTERVAL family. For example, you may convert a YEAR interval to months, but not to days or hours.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Example

The following statement creates a table with an INTERVAL MINUTE column:

```
CREATE TABLE TimeInfo
  (Id INTEGER
   ,Offset INTERVAL MINUTE (4));

INSERT TimeInfo (1001, INTERVAL '2400' MINUTE);
```

## INTERVAL MINUTE TO SECOND Data Type

Identifies a field as an INTERVAL value defining a period of time in minutes and seconds.

## ANSI Compliance

INTERVAL MINUTE TO SECOND is ANSI SQL:2011 compliant.

## Syntax

```
INTERVAL MINUTE [ ( precision ) ] TO SECOND
  [ ( fractional_seconds_precision ) ] [ attributes [...] ]
```

## Syntax Elements

### *precision*

The permitted range of digits for MINUTE, ranging from one to four.

The default is two.

### *fractional\_seconds\_precision*

The fractional precision for the values of SECOND, ranging from zero to six.

The default is six.

### *attributes*

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Internal Representation of INTERVAL MINUTE TO SECOND

Conceptually, INTERVAL MINUTE TO SECOND is treated as a record composed of the following fields.

Field Name	Storage Format	Total Length
MINUTE	SMALLINT	6 bytes
SECOND	INTEGER	

### External Representation of INTERVAL MINUTE TO SECOND

INTERVAL MINUTE TO SECOND types are imported and exported in record and indicator modes as CHARACTER data using the client character set.

WHEN <i>fractional_seconds_precision</i> is ...	THEN the type and format are ...
0	<b>Type:</b> CHARACTER(precision + 4)



WHEN <i>fractional_seconds_precision</i> is ...	THEN the type and format are ...
	<b>Format:</b> '-m( <i>precision</i> ):ss'
<i>n</i> where <i>n</i> is 1 - 6	<b>Type:</b> CHARACTER( <i>precision</i> + <i>n</i> + 5) <b>Format:</b> '-m( <i>precision</i> ):ss.s( <i>n</i> )'

For example, for INTERVAL MINUTE(2) TO SECOND(0), the value might be something like ' 17:39'.

## Range of Values

The range of values for INTERVAL MINUTE TO SECOND is as follows.

Type, Precision, and Fractional Seconds Precision	Minimum Value	Maximum Value
INTERVAL MINUTE(1) TO SECOND(0)	'-9:59'	'9:59'
INTERVAL MINUTE(1) TO SECOND(1)	'-9:59.9'	'9:59.9'
INTERVAL MINUTE(1) TO SECOND(2)	'-9:59.99'	'9:59.99'
INTERVAL MINUTE(1) TO SECOND(3)	'-9:59.999'	'9:59.999'
INTERVAL MINUTE(1) TO SECOND(4)	'-9:59.9999'	'9:59.9999'
INTERVAL MINUTE(1) TO SECOND(5)	'-9:59.99999'	'9:59.99999'
INTERVAL MINUTE(1) TO SECOND(6)	'-9:59.999999'	'9:59.999999'
INTERVAL MINUTE(2) TO SECOND(0)	'-99:59'	'99:59'
INTERVAL MINUTE(2) TO SECOND(1)	'-99:59.9'	'99:59.9'
INTERVAL MINUTE(2) TO SECOND(2)	'-99:59.99'	'99:59.99'
INTERVAL MINUTE(2) TO SECOND(3)	'-99:59.999'	'99:59.999'
INTERVAL MINUTE(2) TO SECOND(4)	'-99:59.9999'	'99:59.9999'
INTERVAL MINUTE(2) TO SECOND(5)	'-99:59.99999'	'99:59.99999'
INTERVAL MINUTE(2) TO SECOND(6)	'-99:59.999999'	'99:59.999999'
INTERVAL MINUTE(3) TO SECOND(0)	'-999:59'	'999:59'
INTERVAL MINUTE(3) TO SECOND(1)	'-999:59.9'	'999:59.9'
INTERVAL MINUTE(3) TO SECOND(2)	'-999:59.99'	'999:59.99'
INTERVAL MINUTE(3) TO SECOND(3)	'-999:59.999'	'999:59.999'
INTERVAL MINUTE(3) TO SECOND(4)	'-999:59.9999'	'999:59.9999'
INTERVAL MINUTE(3) TO SECOND(5)	'-999:59.99999'	'999:59.99999'
INTERVAL MINUTE(3) TO SECOND(6)	'-999:59.999999'	'999:59.999999'

Type, Precision, and Fractional Seconds Precision	Minimum Value	Maximum Value
INTERVAL MINUTE(4) TO SECOND(0)	-'9999:59'	'9999:59'
INTERVAL MINUTE(4) TO SECOND(1)	-'9999:59.9'	'9999:59.9'
INTERVAL MINUTE(4) TO SECOND(2)	-'9999:59.99'	'9999:59.99'
INTERVAL MINUTE(4) TO SECOND(3)	-'9999:59.999'	'9999:59.999'
INTERVAL MINUTE(4) TO SECOND(4)	-'9999:59.9999'	'9999:59.9999'
INTERVAL MINUTE(4) TO SECOND(5)	-'9999:59.99999'	'9999:59.99999'
INTERVAL MINUTE(4) TO SECOND(6)	-'9999:59.999999'	'9999:59.999999'

### Implicit and Explicit INTERVAL MINUTE TO SECOND Conversion

Vantage performs implicit conversion from one Interval data type to another Interval type in some cases. You can also use CAST to explicitly convert one Interval type to another.

Conversions are possible only within the same INTERVAL family. For example, you may convert a YEAR interval to months, but not to days or hours.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Example

The following statement creates a table with an INTERVAL MINUTE TO SECOND column:

```
CREATE TABLE TimeInfo
  (Id INTEGER
   ,Offset INTERVAL MINUTE (1) TO SECOND (2));

INSERT TimeInfo (1001, INTERVAL '6:15.24' MINUTE TO SECOND);
```

## INTERVAL SECOND Data Type

Identifies a field as an INTERVAL value defining a period of time in seconds.

### ANSI Compliance

INTERVAL SECOND is ANSI SQL:2011 compliant.

### Syntax

```
INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) ]
  [ attributes [...] ]
```

## Syntax Elements

### *precision*

The permitted range of digits for SECOND, ranging from one to four.

The default is two.

### *fractional\_seconds\_precision*

The fractional precision for the values of SECOND, ranging from zero to six.

The default is six.

### *attributes*

Appropriate data type, column storage, or column constraint attributes.

See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

## Usage Notes

### Internal Representation of INTERVAL SECOND

Conceptually, the internal representation of INTERVAL SECOND consists of the following fields.

Field Name	Storage Format	Total Length
WHOLE SECONDS	SMALLINT	6 bytes
FRACTIONAL SECONDS	INTEGER	

### External Representation of INTERVAL SECOND

INTERVAL SECOND types are imported and exported in record and indicator modes as CHARACTER data using the client character set.

WHEN <i>fractional_seconds_precision</i> is ...	THEN the type and format are ...
0	<b>Type:</b> CHARACTER(precision + 1)
	<b>Format:</b> '-s( <i>precision</i> )'
<i>n</i> where <i>n</i> is 1 - 6	<b>Type:</b> CHARACTER(precision + <i>n</i> + 2)
	<b>Format:</b> '-s( <i>precision</i> ).s( <i>n</i> )'

For example, for INTERVAL SECOND(3,2), the value might be something like '-127.35'.

## Range of Values

The range of values for INTERVAL SECOND is as follows.

Type, Precision, and Fractional Seconds Precision	Minimum Value	Maximum Value
INTERVAL SECOND(1,0)	'-9'	'9'
INTERVAL SECOND(1,1)	'-9.9'	'9.9'
INTERVAL SECOND(1,2)	'-9.99'	'9.99'
INTERVAL SECOND(1,3)	'-9.999'	'9.999'
INTERVAL SECOND(1,4)	'-9.9999'	'9.9999'
INTERVAL SECOND(1,5)	'-9.99999'	'9.99999'
INTERVAL SECOND(1,6)	'-9.999999'	'9.999999'
INTERVAL SECOND(2,0)	'-99'	'99'
INTERVAL SECOND(2,1)	'-99.9'	'99.9'
INTERVAL SECOND(2,2)	'-99.99'	'99.99'
INTERVAL SECOND(2,3)	'-99.999'	'99.999'
INTERVAL SECOND(2,4)	'-99.9999'	'99.9999'
INTERVAL SECOND(2,5)	'-99.99999'	'99.99999'
INTERVAL SECOND(2,6)	'-99.999999'	'99.999999'
INTERVAL SECOND(3,0)	'-999'	'999'
INTERVAL SECOND(3,1)	'-999.9'	'999.9'
INTERVAL SECOND(3,2)	'-999.99'	'999.99'
INTERVAL SECOND(3,3)	'-999.999'	'999.999'
INTERVAL SECOND(3,4)	'-999.9999'	'999.9999'
INTERVAL SECOND(3,5)	'-999.99999'	'999.99999'
INTERVAL SECOND(3,6)	'-999.999999'	'999.999999'
INTERVAL SECOND(4,0)	'-9999'	'9999'
INTERVAL SECOND(4,1)	'-9999.9'	'9999.9'
INTERVAL SECOND(4,2)	'-9999.99'	'9999.99'
INTERVAL SECOND(4,3)	'-9999.999'	'9999.999'
INTERVAL SECOND(4,4)	'-9999.9999'	'9999.9999'
INTERVAL SECOND(4,5)	'-9999.99999'	'9999.99999'

Type, Precision, and Fractional Seconds Precision	Minimum Value	Maximum Value
INTERVAL SECOND(4,6)	-'9999.999999'	'9999.999999'

### Implicit and Explicit INTERVAL SECOND Conversion

Vantage performs implicit conversion from one Interval data type to another Interval type in some cases. You can also use CAST to explicitly convert one Interval type to another.

Conversions are possible only within the same INTERVAL family. For example, you may convert a YEAR interval to months, but not to days or hours.

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

### Example

The following statement creates a table with an INTERVAL SECOND column:

```
CREATE TABLE TimeInfo
  (Id INTEGER
   ,Offset INTERVAL SECOND (2,3));

INSERT TimeInfo (1001, INTERVAL -'15.244' SECOND);
```

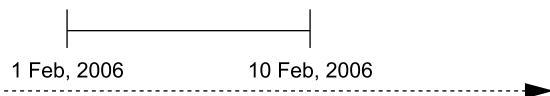
# Period Data Types

This section describes the Period data types. In addition to defining a column whose data type is a Period type, you can also define a derived period column using two DATE or TIMESTAMP columns in a table. For more information about derived period columns, see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Period Data Types: Basic Definitions

### Period

A Period is an anchored duration. It represents a set of contiguous time granules within that duration. It has a beginning bound (defined by the value of a beginning element) and an ending bound (defined by the value of an ending element). The representation of the period is inclusive-exclusive; that is, the period extends from the beginning bound up to but not including the ending bound. The following diagram represents a period of 9 days starting from 1 February, 2006 to 10 February, 2006. The Period includes 1 February, 2006 and extends up to, but does not include, 10 February, 2006.



### Element Type

The element type of a Period data type is the data type of the beginning and ending elements of a value of that Period data type. The element type can be any DateTime data type. The DateTime data types are DATE, TIME, and TIMESTAMP. The TIME and TIMESTAMP data types have a number (from 0 to 6) of fractional seconds in the seconds field which can be specified or defaults to 6 (for example, TIME(3) and TIMESTAMP(6)). They can also explicitly include a time zone field by specifying WITH TIME ZONE (if WITH TIME ZONE is not specified, a time zone field is implicitly not included).

Note that the element type must be the same for both the beginning and ending elements of a period.

### Comparable Period Data Types

Two Period values are comparable if their element types are of same DateTime data type. The DateTime data types are DATE, TIME, and TIMESTAMP.

Data type ...	Is comparable with a ...
PERIOD(DATE)	PERIOD(DATE) data type.
PERIOD(TIME(n) [WITH TIME ZONE])	PERIOD(TIME(m) [WITH TIME ZONE]) data type.
PERIOD(TIMESTAMP(n) [WITH TIME ZONE])	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 operand, these operands will be comparable.

Note that 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 zeroes. Thus, the comparison operations with differing precisions work without any additional logic. Additionally, 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 operands 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 that the time zone values are ignored when comparing values.

## Assignment Operators

A value is only assignable to a Period type target if either one of the following is true:

- The source and target both have Period data types with the same element type. That is, the element types are both DATE, both TIME, or both TIMESTAMP.
- It is possible to implicitly cast the source (which must have a CHARACTER or VARCHAR data type) as the data type of the target. In this case, the source is implicitly cast as the data type before assignment.

In addition, the target precision must not be lower than the source precision.

A Period value is not assignable to a target that does not have a Period data type.

## Time Granule

A time granule or, simply, granule is the minimum interval that can be represented at a given precision. For example, if the element type of a period is DATE, the granule is one day (that is, INTERVAL '1' DAY); if the element type of a period is TIME(0), the granule is one second (that is, INTERVAL '1' SECOND); if the element type of a period is TIMESTAMP(2), the granule is one hundredth of a second (that is, INTERVAL '0.01' SECOND).

## Last Value

The last value of a period is the greatest value of the element type that is less than the value of the ending element of the period. That is, the last value is the ending bound minus one granule of the element type.

## Duration

The duration of a period is the number of granules in a period and is represented as an interval.

## Period Type Input and Output Parameters

You can specify a Period type as a parameter or return type for UDFs written in C, C++, or Java. This includes scalar and aggregate UDFs, table functions, and table operators.

You can specify a Period type as an IN, INOUT, or OUT parameter of stored procedures and external stored procedures written in C, C++, or Java.

You can specify a Period type as a parameter or return type for UDMs written in C or C++.

FNC functions and Java classes and methods are provided to enable a UDF, UDM, or external stored procedure to access and set the value of a Period parameter, or to get information about the Period type parameter. For information about these functions and methods, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## Related Information

For more information on DateTime data types, see [DateTime and Interval Data Types](#).

## PERIOD(DATE) Data Type

A data type that has two DateTime elements associated with it.

The DateTime element...	Specifies...
beginning	the beginning bound of a period.
ending	the ending bound of a period.

The beginning bound is inclusive, and the ending bound is exclusive; that is, the DateTime range starts at the beginning bound and extends up to but not including the ending bound.

### ANSI Compliance

Period types are a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
PERIOD(DATE) [ attribute [...] ]
```

### Syntax Elements

#### *attributes*

Appropriate data type, column storage, or column constraint attributes. See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

The following data type attributes are supported for a PERIOD(DATE) column:

- NULL and NOT NULL
- FORMAT 'format string'
- TITLE
- NAMED
- DEFAULT NULL
- DEFAULT value
- WITH DEFAULT



For more information on these data type attributes, see [Default Value Control Phrases](#) and [Data Type Formats and Format Phrases](#).

The following data type attributes are not supported for a PERIOD(DATE) column:

- DEFAULT USER
- DEFAULT DATE
- DEFAULT TIME
- DEFAULT CURRENT\_DATE
- DEFAULT CURRENT\_TIME[(n)]
- DEFAULT CURRENT\_TIMESTAMP[(n)]
- UPPERCASE or UC
- CASE\_SPECIFIC or CS
- CHARACTER SET

## Usage Notes

### Storage

A PERIOD(DATE) field is a fixed length data type and is saved as two DATE values.

Element Type	Field Size in bytes	Maximum Size in bytes in the row
DATE	8	8

### External Representation of PERIOD(DATE)

In field mode, SQL Engine returns PERIOD(DATE) data as character data.

Assume  $L$  is the maximum length of the formatted character string for the format associated with this Period data type. The resulting character string contains two strings representing the beginning and ending bounds of the period value expression, each up to length  $L$ , and each enclosed in apostrophes ('), separated by comma and a space (,), and then enclosed within a left and right parenthesis [( )]. Thus, the maximum length of the resulting character string is  $2 * L + 8$ .

Assume the actual length is  $K$  which may be less than  $2 * L + 8$ , for example, if the format includes the full names of months and the specific month for a bound is July.

For modes other than field mode, and for input data, the external representation of PERIOD(DATE) consists of two consecutive date values. Each date value is a 4-byte, signed integer flipped to client form. This integer represents a date in the same manner as for a DATE data type, for example,  $(10000 * (\text{year} - 1900)) + (100 * \text{month}) + \text{day}$ .

### Restrictions

A primary index column or partitioning column cannot be a column that has a Period data type.

## Examples

### Example: PERIOD(DATE) Data Type

The following CREATE TABLE statement defines a PERIOD(DATE) column with a default value set using a Period literal.

```
CREATE TABLE t1
(
    employee_id      INTEGER,
    employee_name     CHARACTER(15),
    employee_duration PERIOD(DATE)
    DEFAULT PERIOD '(2005-02-03, 2006-02-03)'
);
```

### Example: Period Parameter in a Java UDF

```
REPLACE FUNCTION PDT_UDF (P1 PERIOD(DATE), P2 PERIOD(DATE))
RETURNS PERIOD (DATE)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.pdt_udf';

public static java.sql.Struct pdt_udf(java.sql.Struct p1, java.sql.Struct p2)
throws SQLException
```

Alternatively, you can define the function as follows:

```
REPLACE FUNCTION PDT_UDF (P1 PERIOD(DATE), P2 PERIOD(DATE))
RETURNS PERIOD (DATE)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.pdt_udf(java.sql.Struct,
java.sql.Struct) returns java.sql.Struct';

public static java.sql.Struct pdt_udf(java.sql.Struct p1, java.sql.Struct p2)
throws SQLException
```

### Example: Period Parameter in a Java External Stored Procedure

```
REPLACE PROCEDURE PDT_XSP(IN P1 PERIOD(DATE), INOUT P2 PERIOD(DATE), OUT P3
PERIOD(DATE))
```

```
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.pdt_xsp';

public static void pdt_xsp(java.sql.Struct p1, java.sql.Struct[] p2,
java.sql.Struct[] p3) throws SQLException
```

## Related Information

For information on functions and operators that apply to Period types, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## PERIOD(TIME) Data Type

A data type that has two DateTime elements associated with it.

The DateTime element...	Specifies...
beginning	the beginning bound of a period.
ending	the ending bound of a period.

The beginning bound is inclusive, and the ending bound is exclusive; that is, the DateTime range starts at the beginning bound and extends up to but not including the ending bound.

A PERIOD(TIME[(n)]) column records the beginning and ending bounds in UTC form in the same manner as exists currently for a TIME column.

### ANSI Compliance

Period types are a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
PERIOD(TIME [ ( fractional_seconds_precision ) ] )
[ attribute [...] ]
```

### Syntax Elements

#### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field.

Values for *fractional\_seconds\_precision* range from 0 to 6 inclusive.

The default precision is 6.

#### *attributes*

Appropriate data type, column storage, or column constraint attributes. See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

The following data type attributes are supported for a PERIOD(TIME) column:

- NULL and NOT NULL
- FORMAT 'format string'
- TITLE
- NAMED
- DEFAULT NULL
- DEFAULT value

For more information on these data type attributes, see [Default Value Control Phrases](#) and [Data Type Formats and Format Phrases](#).

The following data type attributes are not supported for a PERIOD(TIME) column:

- DEFAULT USER
- DEFAULT DATE
- DEFAULT TIME
- DEFAULT CURRENT\_DATE
- DEFAULT CURRENT\_TIME[(n)]
- DEFAULT CURRENT\_TIMESTAMP[(n)]
- UPPERCASE or UC
- CASESPECIFIC or CS
- CHARACTER SET
- WITH DEFAULT

## Usage Notes

### Storage

A PERIOD(TIME[(n)]) is a fixed length data type and is saved as two TIME values.

Element Type	Field Size in bytes	Maximum Size in bytes in the row
TIME(n)	12	16

### External Representation of PERIOD(TIME)

In field mode, SQL Engine returns PERIOD(TIME) data as character data.

Assume  $L$  is the maximum length of the formatted character string for the format associated with this Period data type. The resulting character string contains two strings representing the beginning and ending bounds of the period value expression, each up to length  $L$ , and each enclosed in apostrophes ('), separated by comma and a space (,), and then enclosed within a left and right parenthesis [( )]. Thus, the maximum length of the resulting character string is  $2 * L + 8$ .

For modes other than field mode, and for input data, the external representation of PERIOD(TIME) consists of two consecutive time values. Each time value consists of multiple fields as explained below and returned in the specified order:

- Second: 4-byte, signed integer flipped to client form. This integer represents the number of seconds as a scaled decimal (for example, 12.56 seconds is returned as 12560000).
- Hour: 1 unsigned byte. This byte represents the number of hours.
- Minute: 1 unsigned byte. This byte represents the number of minutes.

## Restrictions

A primary index column or partitioning column cannot be a column that has a Period data type.

## Example

The following CREATE TABLE statement defines a PERIOD(TIME(6)) column since the precision defaults to 6.

```
CREATE TABLE t8
(
    job_id          INTEGER,
    job_desc        CHARACTER(15),
    job_status      CHARACTER(1),
    job_hours       PERIOD(TIME));
```

## Related Information

For information on functions and operators that apply to Period types, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## PERIOD(TIME WITH TIME ZONE) Data Type

A data type that has two DateTime elements associated with it.

The DateTime element...	Specifies...
beginning	the beginning bound of a period.
ending	the ending bound of a period.

The beginning bound is inclusive, and the ending bound is exclusive; that is, the DateTime range starts at the beginning bound and extends up to but not including the ending bound.

A PERIOD(TIME[(n)] WITH TIME ZONE) column records the beginning and ending bounds in UTC form in the same manner as exists currently for a TIME WITH TIME ZONE column.

## ANSI Compliance

Period types are a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
PERIOD(TIME [ ( fractional_seconds_precision ) ] WITH TIME ZONE )
[ attribute [...] ]
```

## Syntax Elements

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field.

Values for *fractional\_seconds\_precision* range from 0 to 6 inclusive.

The default precision is 6.

### *attributes*

Appropriate data type, column storage, or column constraint attributes. See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

The following data type attributes are supported for a PERIOD(TIME WITH TIME ZONE) column:

- NULL and NOT NULL
- FORMAT 'format string'
- TITLE
- NAMED
- DEFAULT NULL
- DEFAULT value

For more information on these data type attributes, see [Default Value Control Phrases](#) and [Data Type Formats and Format Phrases](#).

The following data type attributes are not supported for a PERIOD(TIME WITH TIME ZONE) column:

- DEFAULT USER
- DEFAULT DATE
- DEFAULT TIME
- DEFAULT CURRENT\_DATE
- DEFAULT CURRENT\_TIME[(n)]
- DEFAULT CURRENT\_TIMESTAMP[(n)]
- UPPERCASE or UC
- CASESPECIFIC or CS
- CHARACTER SET
- WITH DEFAULT

## Usage Notes

### Storage

A PERIOD(TIME[(n)] WITH TIME ZONE) is a fixed length data type and is saved as two TIME WITH TIME ZONE values.

Element Type	Field Size in Bytes	Maximum Size in Bytes in the Row
TIME(n) WITH TIME ZONE	16	16

### External Representation of PERIOD(TIME WITH TIME ZONE)

In field mode, Vantage returns PERIOD(TIME WITH TIME ZONE) data as character data.

Assume  $L$  is the maximum length of the formatted character string for the format associated with this Period data type. The resulting character string contains two strings representing the beginning and ending bounds of the period value expression, each up to length  $L$ , and each enclosed in apostrophes ('), separated by comma and a space (,), and then enclosed within a left and right parenthesis [( )]. Thus, the maximum length of the resulting character string is  $2 * L + 8$ .

For modes other than field mode, and for input data, the external representation of PERIOD(TIME WITH TIME ZONE) consists of two consecutive time with time zone values. Each time value consists of multiple fields as explained below and returned in the specified order:

- Second: 4-byte, signed integer flipped to client form. This integer represents the number of seconds as a scaled decimal (for example, 12.56 seconds is returned as 12560000).
- Hour: 1 unsigned byte. This byte represents the number of hours.
- Minute: 1 unsigned byte. This byte represents the number of minutes.
- Time Zone Hour: 1 unsigned byte. This byte represents the hours portion of the time zone displacement along with whether the displacement is + or -. A value of 16 represents 0 hours. A value less than 16 represents a negative time zone displacement for the specified hours; that is, if this is 10, the time zone is displaced by -10 hours. If greater than 16, it specifies a positive time zone displacement of (Time Zone Hour - 16) hours; that is, a value of 20 implies a +4 hour displacement.
- Time Zone Minute: 1 unsigned byte. This byte represents the minute's portion of the time zone displacement.

### Restrictions

A primary index column or partitioning column cannot be a column that has a Period data type.

### Example

The following CREATE TABLE statement defines a PERIOD(TIME(6) WITH TIME ZONE) column since the precision defaults to 6.



```
CREATE TABLE t8
(
    job_id          INTEGER,
    job_desc        CHARACTER(15),
    job_status      CHARACTER(1),
    job_hours       PERIOD(TIME WITH TIME ZONE));
```

## Related Information

For information on functions and operators that apply to Period types, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## PERIOD(TIMESTAMP) Data Type

A data type that has two DateTime elements associated with it.

The DateTime element...	Specifies...
beginning	the beginning bound of a period.
ending	the ending bound of a period.

The beginning bound is inclusive, and the ending bound is exclusive; that is, the DateTime range starts at the beginning bound and extends up to but not including the ending bound.

A PERIOD(TIMESTAMP[(n)]) column records the beginning and ending bounds in UTC form in the same manner as exists currently for a TIME or TIMESTAMP column.

## ANSI Compliance

Period types are a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
PERIOD(TIMESTAMP [ ( fractional_seconds_precision ) ] )
[ attribute [...] ]
```

## Syntax Elements

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field.

Values for *fractional\_seconds\_precision* range from 0 to 6 inclusive.

The default precision is 6.

**attributes**

Appropriate data type, column storage, or column constraint attributes. See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

The following data type attributes are supported for a PERIOD(TIMESTAMP) column:

- NULL and NOT NULL
- FORMAT 'format string'
- TITLE
- NAMED
- DEFAULT NULL
- DEFAULT value
- WITH DEFAULT

For more information on these data type attributes, see [Default Value Control Phrases](#) and [Data Type Formats and Format Phrases](#).

The following data type attributes are not supported for a PERIOD(TIMESTAMP) column:

- DEFAULT USER
- DEFAULT DATE
- DEFAULT TIME
- DEFAULT CURRENT\_DATE
- DEFAULT CURRENT\_TIME[(n)]
- DEFAULT CURRENT\_TIMESTAMP[(n)]
- UPPERCASE or UC
- CASESPECIFIC or CS
- CHARACTER SET

## Usage Notes

### Storage

A PERIOD(TIMESTAMP[(n)]) is a variable length data type and is saved as two TIMESTAMP values. The ending bound value of UNTIL\_CHANGED occupies a single byte for these variable length data types.

Element Type	Field Size in bytes	Maximum Size in bytes in the row
TIMESTAMP(n)	20 if ending bound is not UNTIL_CHANGED	24 if ending bound is not UNTIL_CHANGED
	11 if the ending bound is UNTIL_CHANGED	16 if the ending bound is UNTIL_CHANGED

## External Representation of PERIOD(TIMESTAMP)

In field mode, Vantage returns PERIOD(TIMESTAMP) data as character data.

Assume  $L$  is the maximum length of the formatted character string for the format associated with this Period data type. The resulting character string contains two strings representing the beginning and ending bounds of the period value expression, each up to length  $L$ , and each enclosed in apostrophes ('), separated by comma and a space (,), and then enclosed within a left and right parenthesis [( )]. Thus, the maximum length of the resulting character string is  $2 * L + 8$ .

Assume the actual length is  $K$  which may be less than  $2 * L + 8$ , for example, if the format includes the full names of months and the specific month for a bound is July.

For modes other than field mode, and for input data, the external representation of PERIOD(TIMESTAMP) consists of 2-bytes representing the length of the data followed by 2 consecutive timestamp values. Each timestamp value consists of multiple fields as explained below and returned in the specified order:

- Second: 4-byte, signed integer flipped to client form. This integer represents the number of seconds as a scaled decimal (for example, 12.56 seconds is returned as 12560000).
- Year: 2-byte, signed short integer flipped to client form. This integer represents the year value.
- Month: 1 unsigned byte. This byte represents the month value.
- Day: 1 unsigned byte. This byte represents the day of the month.
- Hour: 1 unsigned byte. This byte represents the number of hours.
- Minute: 1 unsigned byte. This byte represents the number of minutes.

## Restrictions

A primary index column or partitioning column cannot be a column that has a Period data type.

## Example

The following CREATE TABLE statement defines a PERIOD(TIMESTAMP(3)) column with a default value specified using WITH DEFAULT. The WITH DEFAULT option sets the default value to a Period value constructor with the beginning argument set to CURRENT\_TIMESTAMP(3) and the ending argument set to UNTIL\_CHANGED.

```
CREATE TABLE t4
(
    employee_id      INTEGER,
    employee_name     CHARACTER(15),
    employee_duration PERIOD(TIMESTAMP(3)) WITH DEFAULT
);
```

## Related Information

For information on functions and operators that apply to Period types, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## PERIOD(TIMESTAMP WITH TIME ZONE) Data Type

A data type that has two DateTime elements associated with it.

The DateTime element...	Specifies...
beginning	the beginning bound of a period.
ending	the ending bound of a period.

The beginning bound is inclusive, and the ending bound is exclusive; that is, the DateTime range starts at the beginning bound and extends up to but not including the ending bound.

A PERIOD(TIMESTAMP[(n)] WITH TIME ZONE) column records the beginning and ending bounds in UTC form in the same manner as exists currently for a TIMESTAMP WITH TIME ZONE column.

## ANSI Compliance

Period types are a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
PERIOD(TIMESTAMP [ ( fractional_seconds_precision ) ] WITH TIME ZONE )
[ attribute [...] ]
```

## Syntax Elements

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field.

Values for *fractional\_seconds\_precision* range from 0 to 6 inclusive.

The default precision is 6.

### *attributes*

Appropriate data type, column storage, or column constraint attributes. See [Core Data Type Attributes](#) and [Storage and Constraint Attributes](#) for specific information.

The following data type attributes are supported for a PERIOD(TIMESTAMP WITH TIME ZONE) column:

- NULL and NOT NULL

- FORMAT 'format string'
- TITLE
- NAMED
- DEFAULT NULL
- DEFAULT value
- WITH DEFAULT

For more information on these data type attributes, see [Default Value Control Phrases](#) and [Data Type Formats and Format Phrases](#).

The following data type attributes are not supported for a PERIOD(TIMESTAMP WITH TIME ZONE) column:

- DEFAULT USER
- DEFAULT DATE
- DEFAULT TIME
- DEFAULT CURRENT\_DATE
- DEFAULT CURRENT\_TIME[(n)]
- DEFAULT CURRENT\_TIMESTAMP[(n)]
- UPPERCASE or UC
- CASESPECIFIC or CS
- CHARACTER SET

## Usage Notes

### Storage

A PERIOD(TIMESTAMP[(n)] WITH TIME ZONE) is a variable length data type and is saved as two TIMESTAMP WITH TIME ZONE values. The ending bound value of UNTIL\_CHANGED occupies a single byte for these variable length data types.

Element Type	Field Size in bytes	Maximum Size in bytes in the row
TIMESTAMP(n) WITH TIME ZONE	24 if ending bound is not UNTIL_CHANGED	24 if ending bound is not UNTIL_CHANGED
	13 if the ending bound is UNTIL_CHANGED	16 if the ending bound is UNTIL_CHANGED

### External Representation of PERIOD(TIMESTAMP WITH TIME ZONE)

In field mode, Vantage returns PERIOD(TIMESTAMP WITH TIME ZONE) data as character data.

Assume  $L$  is the maximum length of the formatted character string for the format associated with this Period data type. The resulting character string contains two strings representing the beginning and ending bounds of the period value expression, each up to length  $L$ , and each enclosed in apostrophes ('),

separated by comma and a space (,), and then enclosed within a left and right parenthesis [( )]. Thus, the maximum length of the resulting character string is  $2 * L + 8$ .

Assume the actual length is  $K$  which may be less than  $2 * L + 8$ , for example, if the format includes the full names of months and the specific month for a bound is July.

For modes other than field mode, and for input data, the external representation of PERIOD(TIMESTAMP WITH TIME ZONE) consists of 2-bytes representing the length of the data followed by 2 consecutive timestamp with time zone values. Each timestamp value consists of multiple fields as explained below and returned in the specified order:

- Second: 4-byte, signed integer flipped to client form. This integer represents the number of seconds as a scaled decimal (for example, 12.56 seconds is returned as 12560000).
- Year: 2-byte, signed short integer flipped to client form. This integer represents the year value.
- Month: 1 unsigned byte. This byte represents the month value.
- Day: 1 unsigned byte. This byte represents the day of the month.
- Hour: 1 unsigned byte. This byte represents the number of hours.
- Minute: 1 unsigned byte. This byte represents the number of minutes.
- Time Zone Hour: 1 unsigned byte. This byte represents the time zone displacement in hours along with whether the displacement is + or -. A value of 16 represents 0 hours. A value less than 16 represents a negative time zone displacement for the specified hours; that is, if this is 10, the time zone is displaced by -10 hours. If greater than 16, it specifies a positive time zone displacement of (Time Zone Hour - 16) hours; that is, a value of 20 implies a +4 hour displacement.
- Time Zone Minute: 1 unsigned byte. This byte represents the time zone displacement in minutes.

## Restrictions

A primary index column or partitioning column cannot be a column that has a Period data type.

## Example

The following CREATE TABLE statement defines a PERIOD(TIMESTAMP(3) WITH TIME ZONE) column with a default value specified using WITH DEFAULT. The WITH DEFAULT option sets the default value to a Period value constructor with the beginning argument set to CURRENT\_TIMESTAMP(3) and the ending argument set to UNTIL\_CHANGED.

```
CREATE TABLE t4
(
    employee_id      INTEGER,
    employee_name    CHARACTER(15),
    employee_duration PERIOD(TIMESTAMP(3) WITH TIME ZONE) WITH DEFAULT
);
```

## Related Information

For information on functions and operators that apply to Period types, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

# ARRAY/VARRAY Data Type

This section describes the ARRAY data type.

An ARRAY data type is a named data type that has a user-defined maximum number of elements of the same specific data type. You can define an ARRAY data type to be of one or more dimensions and use it to store many values of the same data type sequentially or in a matrix-like format. This extends the number of data values of the same type that can be stored in a table row.

Vantage supports a one-dimensional (1-D) ARRAY data type and a multidimensional (n-D) ARRAY data type, with up to 5 dimensions.

You can also create an ARRAY data type using the VARRAY keyword and syntax for Oracle compatibility.

A data type used for storing and accessing multidimensional data. The ARRAY data type can store many values of the same specific data type in a sequential or matrix-like format.

## ANSI Compliance

The one-dimensional (1-D) ARRAY data type is partially compliant with the ANSI SQL:2011 standard. The 1-D ARRAY type is not compatible with the ANSI standard in the following areas:

- You must create an ARRAY type, using the CREATE TYPE statement, before you can use the ARRAY type. This is not an ANSI requirement.
- The Teradata ARRAY data type is a user-defined type (UDT). This differs from the ANSI standard which does not consider an ARRAY data type to be a UDT.
- Teradata syntax for the ARRAY value constructor is based on the UDT constructor syntax. This differs from the ANSI syntax.
- ANSI-style comparison of two arrays (A=B) is not supported. However, comparison on individual elements of two arrays, that is A[2]=B[2], is supported. You can use the ARRAY\_COMPARE system function to perform equals/not equals comparison on all the elements of two arrays.
- Vantage provides a built-in CAST operation for ARRAY types that performs a CAST from VARCHAR to ARRAY, and ARRAY to VARCHAR. This differs from ANSI ARRAY CAST functionality, where all of the elements in a source array are cast to the element type of a target array.
- The concatenation operator requires that both operands are the same ARRAY type and that the target type of the concatenation operation is also the same type. This is a deviation from the ANSI standard since it defines the target data type of a 1-D ARRAY concatenation to be a new 1-D ARRAY type with the length defined as the sum of the length of both operands.
- The ARRAY\_AGG system aggregate function includes an additional parameter containing an ARRAY expression of the target ARRAY type. This eliminates ambiguity since you can define multiple ARRAY types that have the same element data type.
- An ARRAY constructor by query is not supported. You can use the ARRAY\_AGG system aggregate function to perform this type of operation.



The multidimensional (n-D) ARRAY data type is a Teradata extension to the ANSI SQL standard. Vantage extends the ANSI SQL:2011 syntax for the 1-D ARRAY type to permit multiple dimensions.

Both 1-D and n-D ARRAY data types store and manage their element values in a way that is compliant with the ANSI SQL:2011 standard.

## Syntax

```
[SYSUDTLIB.] array_type_name [ attribute [...] ]
```

### Syntax Elements

#### **SYSUDTLIB.**

The name of the database in which all ARRAY data types are created.

#### **array\_type\_name**

The name of an ARRAY data type that was created with a CREATE TYPE statement.

#### **attribute**

Appropriate data type attributes.

An ARRAY column supports the following attributes:

- NULL
- NOT NULL
- FORMAT
- TITLE
- NAMED
- DEFAULT NULL

For details on using data type attributes with ARRAY data types, see:

- [Default Value Control Phrases](#)
- [Data Type Formats and Format Phrases](#)

An ARRAY data type column does not support column storage or column constraint attributes.

## One-Dimensional (1-D) ARRAY Data Type

The 1-D ARRAY type is defined as a variable-length ordered list of values of the same data type. It has a maximum number of values that you specify when you create the ARRAY type. You can access each element value in a 1-D ARRAY type using a numeric index value. For more information on referencing an ARRAY element, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

SQL Engine stores the element values of a 1-D ARRAY sequentially starting with the first element, from left-to-right.

The 1-D ARRAY type is compatible with the VARRAY type provided by Oracle Database except for the following areas:

- The syntax for ARRAY methods that have zero parameters requires an empty set of parenthesis.
- Some methods that provide the same functionality as methods defined for Oracle VARRAY types have slightly different names in Teradata. For example, Oracle provides the FIRST method, but Teradata provides the OFIRST method. The renaming was done to avoid conflicts with Teradata reserved words.
- Vantage provides a set of methods that have the same functionality as some methods defined for Oracle VARRAY types. However, not all methods defined for Oracle VARRAY types are available for Teradata ARRAY types.
- The ALTER TYPE statement is not supported for Teradata ARRAY types. It is supported for Oracle VARRAY types.

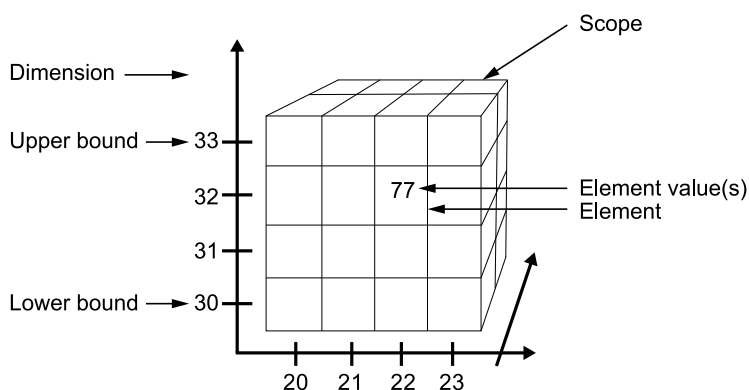
## Multidimensional (n-D) ARRAY Data Type

An n-D ARRAY is a mapping from integer coordinates to an element type. The n-D ARRAY type is defined as a variable-length ordered list of values of the same data type. It has 2-5 dimensions, with a maximum number of values for each dimension, which you specify when you create the ARRAY type.

You define an n-D ARRAY type with a pair of lower and upper boundaries [*n:m*] for each of its dimensions. Alternatively, you can specify a single value [*n*] to signify the maximum size of a dimension, which implicitly defines the lower bound of the dimension to be 1. For more information, see [Creating an ARRAY Data Type](#).

You can access each element value in an n-D ARRAY type using numeric index values for each dimension. For more information on referencing an ARRAY element, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

The figure below shows the constituents of a 3-D ARRAY data type.



Vantage stores the element values of an n-D ARRAY in row-major order. Row-major means that the first dimension, which is leftmost in the scope specification, is the most major dimension, and as you move toward the last dimension, the dimensions become less and less major.

## Privileges Required for Creating an ARRAY Data Type

An ARRAY data type is a UDT, which means that Vantage will store any ARRAY type that you create in the SYSUDTLIB database along with its autogenerated constructor UDF, by default. Therefore, in order to create an ARRAY data type, you must have either the UDTTYPE or UDTMETHOD privilege on the SYSUDTLIB database.

## Creating an ARRAY Data Type

Before you can use an ARRAY data type, you must first create it using the CREATE TYPE statement. When creating a new ARRAY type, you must specify:

- A name for the ARRAY type.
- The data type of the ARRAY elements.

For a 1-D ARRAY type, you also specify a maximum number of elements in the ARRAY.

For an n-D ARRAY type, you specify the number of dimensions from 2 to 5, with a pair of lower and upper boundaries for each dimension. You can specify the array boundary dimensions using any combination of the following two methods:

- Explicitly specify lower and upper bounds for each dimension, separating the two with a colon. For example,  $[n:m]$  where  $n$  and  $m$  are signed integer values, meaning that negative numbers are allowed.
- Specify a single value to signify the maximum size of the dimension using ANSI-style syntax, which implicitly defines the lower bound of the array to be 1. For example,  $[n]$  where  $n$  is an unsigned (positive) integer value.

If you specify the optional DEFAULT NULL clause when creating the ARRAY type, all elements are set to NULL when an instance of the ARRAY type is constructed. Otherwise, all elements are set to an uninitialized state. You will receive an error if you try to access an element that is in an uninitialized state.

For more information, see CREATE TYPE (ARRAY Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Supported Data Types for ARRAY Elements

The data type of an ARRAY element can be any Teradata data type except for the following:

- BLOB
- CLOB
- LOB UDTs (both distinct and structured types)
- Geospatial
- Parameter data types, such as TD\_ANYTYPE or VARIANT\_TYPE
- ARRAY

Note that you cannot specify an ARRAY data type as the element type of an ARRAY type.

The following online HELP statement lists all of the valid element data types for an ARRAY data type:

```
HELP 'array_data_type declarations';
```

## NOTICE

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

## Autogenerated Functionality for an ARRAY Data Type

Vantage automatically generates the following routines for each ARRAY data type you create.

Autogenerated Routine	Description
Constructor function	Allocates an ARRAY type instance and sets all the elements to an uninitialized state if the DEFAULT NULL clause was not specified. If DEFAULT NULL was specified, all elements are set to null.
Constructor method	Takes one or more arguments and initializes each element of the array with the corresponding value passed to the method.

Use the ARRAY constructor expression to create a new instance of an ARRAY data type and initialize it using the autogenerated constructor method or function. For more information about the ARRAY constructor expression, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Vantage automatically generates the following functionality for each ARRAY data type you create.

Autogenerated Functionality	Description
ARRAY type transform	from-sql and to-sql functionality associated with the transform of an ARRAY type. The ARRAY values are transformed to/from a VARCHAR( <i>length</i> ) value, where <i>length</i> depends on the element type and the total number of elements defined in the ARRAY. For detailed information about transform input/output strings for ARRAY types, see <a href="#">External Representations for UDTs</a> .
ARRAY type ordering	Basic ordering functionality for an ARRAY type.
ARRAY type cast	Casting functionality for an ARRAY type. Two autogenerated casts are provided: VARCHAR to ARRAY, and ARRAY to VARCHAR.

For more information, see CREATE TYPE (ARRAY Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Rules and Guidelines

- The 1-D ARRAY type only supports a lower bound of 1. This means that every 1-D ARRAY type that is created has a first array element indexed by the number 1.
- The n-D ARRAY type supports an optional user-specified lower bound that can be a negative or positive integer number.
- The maximum size of an ARRAY type and its autogenerated transform string must not exceed 64 KB because SQL Engine stores its data within the row and because the maximum size of its autogenerated transform string is limited to the maximum size of a VARCHAR type, which is 64,000 Teradata Latin bytes.

Note that the size of the array is influenced not only by the number of elements, but also by the element type of the array.

- For an n-D ARRAY type, you must first consider the row size limit of 64 KB. Then there is a limit to the maximum number of dimensions that can be declared within the scope of the array. The minimum number of dimensions that you can create is 2, and the maximum number of dimensions that you can create is 5.
- You can use the optional DEFAULT NULL clause when creating an ARRAY type to initialize all elements of the ARRAY to null at the time a new instance of the ARRAY type is constructed.

This clause is particularly useful for ARRAY types that are expected to be fully populated.

When you use this clause, the action prevents all subset operations such as AVERAGE, UPDATE, or COUNT for arrays from aborting and returning an error because they refer to an element that is not initialized.

## ARRAY Functions, Operators, and Expressions

Vantage provides system functions, operators, and expressions that operate on either 1-D, n-D, or both ARRAY data types. These functions allow you to perform arithmetic, relational, and aggregate operations on an array in its entirety or on a subset of the elements composing the array. For information about these functions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## ARRAY Type Input and Output Parameters

You can specify the ARRAY type as parameters and return types for UDFs written in C, C++, or Java. This includes scalar and aggregate UDFs, table functions, and table operators.

You can specify the ARRAY type as IN, INOUT, and OUT parameters of stored procedures and external stored procedures written in C, C++, or Java.

You can specify the ARRAY type as parameters and return types for UDMs written in C or C++.

ARRAY FNC functions and Java classes and methods are provided to enable a UDF, UDM, or external stored procedure to access and set the values of the elements in an ARRAY, or to get information about the

ARRAY type parameter. For information about these functions and methods, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## Restrictions

- You cannot pass ARRAY values as arguments to the search condition in a WHERE clause. You can pass individual elements of an ARRAY to the search condition in a WHERE clause. Also, you may use system functions provided by Vantage to do relational comparison on ARRAY data. For information about these functions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.
- You cannot specify an ARRAY column in any of the following SQL DML clauses:
  - INTERSECT, MINUS, or UNION set operator
  - DISTINCT operator
  - ORDER BY, GROUP BY, or HAVING clause of a SELECT statement
- You cannot create UDMs for an ARRAY type. The only valid methods for an ARRAY type are the methods that Vantage creates automatically for an ARRAY type. Therefore, you cannot specify an ARRAY type name in the FOR clause of a CREATE/REPLACE METHOD statement.
- An ARRAY column cannot be a component of an index.
- Any restrictions that apply to a UDT also apply to an ARRAY type.

## Example: Creating an ARRAY Data Type

The following statement uses Teradata syntax to create a 1-D ARRAY type with 5 elements of type CHAR(10). All elements of the array are initialized to null.

```
CREATE TYPE phonenumbers_ary AS CHAR(10) ARRAY[5] DEFAULT NULL;
```

The following statement uses Oracle-compatible syntax to create a 1-D ARRAY type with 5 elements of type CHAR(10). All elements of the array are set to an uninitialized state.

```
CREATE TYPE phonenumbers_ary AS VARRAY(5) OF CHAR(10);
```

Consider the following structured UDT:

```
CREATE TYPE measures_UDT AS(amplitude INTEGER,
                           phase      INTEGER,
                           frequency  INTEGER);
```

The following statement uses Teradata syntax to create a 3-D ARRAY type with elements of type measures\_UDT. The scope of this array is composed of three dimensions:

- Dimension one has a lower bound of 1 and an upper bound of 5.

- Dimension two has a lower bound of 1 and an upper bound of 7.
- Dimension three has a lower bound of 1 and an upper bound of 20.

```
CREATE TYPE seismic_cube AS measures_UDT ARRAY [1:5][1:7][1:20];
```

The following statement uses Oracle-compatible syntax to create the same 3-D ARRAY type.

```
CREATE TYPE seismic_cube AS VARRAY (1:5)(1:7)(1:20) OF measures_UDT;
```

## Example: Creating a Table with an ARRAY Column

Consider the following 1-D ARRAY type:

```
CREATE TYPE phonenumbers_ary AS CHAR(10) ARRAY[5] DEFAULT NULL;
```

The following statement creates a table with a column named ephone that has a data type of phonenumbers\_ary, which is a 1-D ARRAY type.

```
CREATE TABLE my_table (eno INTEGER, ephone phonenumbers_ary);
```

## Example: ARRAY Parameter in a Java UDF

```
CREATE TYPE phonenumbers_ary AS CHAR(10) ARRAY[5];

REPLACE FUNCTION getPhoneNums(A1 phonenumbers_ary)
RETURNS INTEGER
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.getPhoneNums';

public static int getPhoneNums(java.sql.Array a1) throws SQLException
```

Alternatively, you can define the function as follows:

```
REPLACE FUNCTION getPhoneNums(A1 phonenumbers_ary)
RETURNS INTEGER
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME
'UDF_JAR:UserDefinedFunctions.getPhoneNums(java.sql.Array) returns int';
```

```
public static int getPhoneNums(java.sql.Array a1) throws SQLException
```

## Example: ARRAY Parameter in a Java External Stored Procedure

```
CREATE TYPE phonenumbers_ary AS CHAR(10) ARRAY[5];

REPLACE PROCEDURE getPhoneNums(IN A1 phonenumbers_ary,
                                IN EMPID INTEGER, OUT myphonenum INTEGER)

LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.getPhoneNums';

public static void getPhoneNums(java.sql.Array A1, int empid, int[] myphonenum)
```

## Related Information

FOR information on ...	SEE ...
creating an ARRAY data type	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
creating a new instance of an ARRAY data type and initializing it accessing the element values of an ARRAY data type system functions, operators, and expressions that operate on ARRAY data types	<i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145.
writing routines that use ARRAY input and output parameters	<i>Teradata Vantage™ - SQL External Routine Programming</i> , B035-1147.
transform input/output strings for ARRAY types	<a href="#">External Representations for UDTs</a> .



# ARRAY/VARRAY Functions and Operators

## ARRAY/VARRAY Functions and Operators

The following sections describe functions, operators, expressions and methods that operate on either one-dimensional (1-D), multidimensional (n-D), or both ARRAY data types. These functions allow you to perform arithmetic, relational, and aggregate operations on an array in its entirety or on a subset of the elements composing an array.

For information on creating one-dimensional and multi-dimensional arrays, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## ARRAY Element Reference

Accesses the value of a specified element in an ARRAY data type value.

### ANSI Compliance

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

### Syntax

```
array_expression [ index_value [...] ]
```

#### Note:

You must type the colored or bold brackets.

### Syntax Elements

#### *array\_expression*

An expression that evaluates to an ARRAY data type.

#### *index\_value*

Index to the element in the array whose value you want to access.

For a 1-D ARRAY type, *index\_value* must be a positive integer in the range from 1 to *n*, where *n* is the declared size of the ARRAY type.

For an n-D ARRAY type, *index\_value* must be a positive or negative integer in the range from *m* to *n*, where *m* is the declared lower bound of a dimension of the array, and *n* is the declared upper bound.

## Usage Notes

The element of the array being referenced must be initialized to a value or to null. If the referenced array element is in an uninitialized state, an error is returned. You can use the DEFAULT NULL clause to initialize all elements of an array to null at the time the ARRAY data type is created, or you can use the ARRAY constructor expression to initialize the array. For information about the DEFAULT NULL clause, see CREATE TYPE (Array Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. For information about the ARRAY constructor expression, see [ARRAY\\_Constructor\\_Expression](#).

For a 1-D ARRAY type, you can reference a single element using the ARRAY element reference syntax with one set of square brackets. For example, MyArray[n], where MyArray is a 1-D ARRAY data type.

For an n-D ARRAY type, you can reference a single element using the ARRAY element reference syntax with *n* sets of square brackets, where *n* corresponds to the number of dimensions of the n-D ARRAY type. For example, if my3DArray defines a 3-D ARRAY data type, then the ARRAY element reference syntax for accessing one element of this array would be my3DArray[x][y][z]. Note that the maximum number of sets of square brackets is five, which corresponds to the maximum number of dimensions that Teradata supports for n-D ARRAY types.

The number of dimensions referenced in an ARRAY element reference is validated when you attempt to access the element in the array value using the element reference. This requires that an ARRAY value to be referenced already exists, and that the SQL statement actually executes the element reference. For this reason, there are some rare cases where an SQL statement that includes an ARRAY element reference may not actually be executed. For example, a SELECT from a table where the table contains no rows, or a WHERE clause in a SELECT which contains an element reference on the right-hand side of an OR clause, where the left-hand side evaluates to true. In these types of cases, the behavior of an ARRAY element reference is similar to a UDF invocation.

## Using ARRAY Element Reference with a SET Clause

You can use an ARRAY element reference as the target value in a SET clause of an UPDATE or MERGE statement to set the value of an individual element in an ARRAY.

If you set an element in the middle of a 1-D array to a specific value, and the elements previous to this one are not yet initialized, then these elements will be automatically initialized to null. For an n-D ARRAY type, if you set an element in the middle of the array to a specific value, and the elements previous to this one, across all dimensions in row-major order, are not yet initialized, then these elements will be automatically initialized to null.

Note that if the element type of the array is a variable-length type, and if the value passed to initialize the element is larger than the maximum size of the element type, Vantage automatically truncates the passed value to the maximum size of the element type. This truncation behavior occurs with transaction processing in Teradata session mode only. In ANSI session mode, Vantage does not truncate the passed value, and returns an error instead. This behavior is identical to that of distinct and structured UDTs.

In addition, existing implicit CAST functionality, such as casting from CHAR to Timestamp or Timestamp to DATE, is not supported for the ARRAY element reference. The source data type for the SET clause must be the same as the element data type of the array. When the target of the SET clause is an ARRAY element reference, you must use an explicit CAST for the source value of a SET clause that has a different data type than the element type of the array.

## Examples

### Example: Showing the ARRAY Element Reference Syntax

If a table contains a column named ephone, and the data type of the ephone column is a 1-D ARRAY, then the following shows the ARRAY element reference syntax to access the value of the 5th element of the array:

```
ephone[5]
```

### Example: Creating Tables and Set Elements in a 1-D Array

Consider the following 1-D ARRAY data type:

```
CREATE TYPE phonenumbers_ary AS CHAR(10) ARRAY[5];
```

The following statement creates a table with a column named ephone with a data type of phonenumbers\_ary.

```
CREATE TABLE my_table (eno INTEGER, ephone phonenumbers_ary);
```

The following statement uses an ARRAY element reference to set element 3 of the array to the value '5551234567'. If elements 1 and 2 in the array were not initialized, this UPDATE statement automatically sets those elements to null.

```
UPDATE my_table
SET ephone[3] = '5551234567';
```

### Example: Creating Tables and Set Elements in a 3-D Array

Consider the following structured UDT:

```
CREATE TYPE measures_UDT AS(amplitude INTEGER,
                           phase      INTEGER,
                           frequency  INTEGER);
```

The following statement creates a 3-D ARRAY data type named shots with an element type of measures\_UDT.

```
CREATE TYPE shots AS measures_UDT ARRAY[-2:2][-5:5][-3:3];
```

The following statement creates a table with a column named shot\_ary. The data type of the shot\_ary column is shots, which is a 3-D ARRAY type.

```
CREATE TABLE seismic_table (id INTEGER, shot_ary shots);
```

The following query selects the value of an element from the 3-D array.

```
SELECT shot_ary[-1][1][3] FROM seismic_table;
```

## Related Information

For more information, see: [ARRAY\\_Constructor\\_Expression](#).

## ARRAY\_Constructor\_Expression

Constructs a new instance of an ARRAY type and initializes it using the ARRAY constructor method or function.

### ANSI Compliance

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

## ARRAY\_Constructor\_Expression Syntax

```
[ NEW ] array_name ( [ element_expression [, ...] ] )
```

### Syntax Elements

#### *array\_name*

The name of the ARRAY data type that you want to initialize.

***element\_expression***

An expression that evaluates to a literal value. The data type of the value must be the same as the element type of the *array\_name* array.

*element\_expression* can be repeated *n* times, where *n* is the declared size of the *array\_name* array. The maximum limit for *n* is 2559.

## Argument Type and Rules

Expressions passed to the ARRAY constructor must be able to be implicitly converted to the data type of the element type of the *array\_name* array. Existing implicit CAST functionality, such as casting from CHAR to Timestamp or Timestamp to DATE, is supported for the ARRAY constructor expression.

The ARRAY constructor expression accepts zero or more arguments. If there are zero arguments, the ARRAY constructor initializes an empty ARRAY type value and sets all the element values of the array to an uninitialized state if the ARRAY data type was created without the DEFAULT NULL clause. If the ARRAY data type was created with a DEFAULT NULL clause, then the ARRAY constructor initializes all the element values of the array to null. Note that if you try to access an array element that is in an uninitialized state, you will get an error.

You can pass up to *n* arguments to the ARRAY constructor expression, where *n* is the declared size of the ARRAY data type. For a one-dimensional ARRAY type, the ARRAY constructor fills the elements of the array with the argument values sequentially starting from the first element, in the order that the arguments are passed in. For a multidimensional ARRAY type, pass the arguments to the ARRAY constructor expression in row-major order because elements in a multidimensional array are filled in row-major order.

If the number of arguments are less than the number of elements in the array, then the remaining elements for which an argument value is not provided are set to an uninitialized state. However, if the ARRAY data type was created with the DEFAULT NULL clause, then those elements are set to null.

A constructor method for a UDT may not accept more than 128 arguments; however, this limit does not apply to ARRAY data types. The number of arguments that you can specify for the ARRAY constructor is limited only by the maximum number of elements that you can define for the ARRAY type or 2559, whichever comes first.

Note that if the element type of the array is a variable-length type, and if the value passed to initialize the element is larger than the maximum size of the element type, Vantage automatically truncates the passed value to the maximum size of the element type. This truncation behavior occurs with transaction processing in Teradata session mode only. In ANSI session mode, Vantage does not truncate the passed value, and returns an error instead. This behavior is identical to that of distinct and structured UDTs.

## Usage Notes

### Invoking the ARRAY Constructor Expression Without the NEW Keyword

You can invoke the ARRAY constructor expression without using the NEW keyword. This is compatible with Oracle syntax. When invoked without the NEW keyword, Vantage handles the expression like a UDF expression. This restricts the places an ARRAY constructor expression can be used to the same usage as a scalar UDF. This means that in some scenarios, the ARRAY constructor expression with NEW keyword can be used in an SQL statement where the ARRAY constructor expression without NEW keyword cannot. For example, you can use the ARRAY constructor expression with NEW keyword in the DEFAULT clause of a CREATE TABLE statement. However you cannot use the ARRAY constructor expression without NEW keyword in the DEFAULT clause.

Additionally, since the ARRAY constructor expression without NEW keyword is treated like a scalar UDF, the existing search order rules for UDFs will apply in this case. Vantage searches for UDFs in the following databases in the following order:

1. Default database
2. SYSLIB database
3. TD\_SYSFNLIB database
4. SYSUDTLIB database

If a UDF exists in the default database, SYSLIB, or TD\_SYSFNLIB and has the same name as a defined ARRAY data type, and if you invoke the ARRAY constructor expression without the NEW keyword, then Vantage invokes the UDF instead of the ARRAY constructor. To avoid this conflict, use the NEW keyword when invoking the ARRAY constructor.

### Example: Using an ARRAY Constructor Expression

Consider the following 1-D ARRAY data type:

```
CREATE TYPE phonenumbers_ary AS CHAR(10) ARRAY[5];
```

The following shows an ARRAY constructor expression that sets the first two elements in *phonenumbers\_ary* to the provided values, and the rest of the elements in the array are set to an uninitialized state.

```
NEW phonenumbers_ary('5551234567', '8585551234')
```

or

```
phonenumbers_ary('5551234567', '8585551234')
```

## ARRAY Scope Reference

References one or more elements of an ARRAY data type when invoking an ARRAY function.

### ANSI Compliance

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

### Syntax

```
array_lower_bounds, array_upper_bounds
```

### Syntax Elements

#### *array\_lower\_bounds*

An integer value or an array instance of the predefined ARRAY type ArrayVec, with a comma-separated list of integer values to define the lower bounds.

#### *array\_upper\_bounds*

An integer value or an array instance of the predefined ARRAY type ArrayVec, with a comma-separated list of integer values to define the upper bounds.

## Usage Notes

To reference one or more elements of an ARRAY data type when invoking an ARRAY function, you can use an integer value for 1-D arrays, but you must use the ArrayVec ARRAY data type for n-D arrays. The ArrayVec ARRAY data type is automatically created by the system and is defined as:

```
CREATE TYPE ArrayVec AS INTEGER ARRAY[1000];
```

For all ARRAY functions that contain an optional *scope\_reference* or *array\_bound* argument, you can use one or more instances of the ArrayVec ARRAY type to describe the boundaries of each dimension of an n-D ARRAY data type.

## Rules

When the optional *scope\_reference* parameter is specified in an ARRAY function, the following rules apply:

- You can pass two integer values or two ArrayVec values for the *scope\_reference*. If the ARRAY function is called with one-dimensional ARRAY arguments, you can pass a combination of integer and ArrayVec values for the *scope\_reference*. For example, the *scope\_reference* can be *<integer\_value, ArrayVec\_value >* or *<ArrayVec\_value, integer\_value >*.

- If you use an ArrayVec instance for specifying the upper or lower bounds of a *scope\_reference*, the following are not allowed:
  - Passing in NULL for any of the ArrayVec elements that are needed to determine the bound of the array argument.
  - Passing an insufficient number of elements in the ArrayVec to determine the bound of the array argument.
  - Passing too many elements in the ArrayVec to determine the bound of the array argument.
- For all ARRAY functions that define the RETURNS NULL ON NULL INPUT clause, if one of the optional parameters to define the *scope\_reference* is passed as NULL, then NULL is returned as the result value.
- ARRAY functions that are called with a *scope\_reference* specified are considered to have their scope explicitly defined, so these functions return an error if any uninitialized elements fall within the range of the *scope\_reference* in any of the ARRAY input arguments. The CARDINALITY function is an exception to this rule because an explicit scope that includes uninitialized elements is an acceptable input to the function.

If your application requires operations between 2 arrays, consider creating these arrays using the DEFAULT NULL clause at creation time. This will prevent the system from returning errors due to illegal element references since all elements of the array will be initialized.

When the optional *scope\_reference* parameter is not specified in an ARRAY function, the following rules apply:

- If you call an ARRAY function without specifying a *scope\_reference*, the default scope used is the number of initialized elements in the input ARRAY argument. If an ARRAY function accepts two ARRAY arguments and the number of initialized elements is different between the two arrays, then an error is returned.
- When an ARRAY function returns an ARRAY value, the resultant ARRAY value will have the same number of initialized elements as the input ARRAY argument.

## Example: Returning the Number of Distinct Elements within a Range

The following query returns the number of distinct elements within the range from 5 to 10 on each dimension of the phonelist array.

```
SELECT CARDINALITY(phonelist, NEW ArrayVec(5,5), NEW ArrayVec(10,10));
```

## Related Information

For more information, see [ARRAY\\_Constructor\\_Expression](#).



## ARRAY\_AGG

An aggregate function for use with ARRAY data types that allows arrays to be created as part of the SELECT list of a query.

The data type information from the *array\_expression* which resolves to an ARRAY data type is used to set the return type of ARRAY\_AGG.

### ANSI Compliance

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

## ARRAY\_AGG Syntax

```
[TD_SYSFNLIB.] ARRAY_AGG (
  element_value_expr
  [ ORDER BY value_expression [ ASC | DESC ] ] ,
  array_expression
)
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *element\_value\_expr*

An expression that evaluates to an ARRAY element type.

#### *value\_expression*

An expression that evaluates to a Teradata data type that can be compared.

#### ASC

Results are to be ordered in ascending sort order.

If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session.

The default order is ASC.

#### DESC

Results are to be ordered in descending sort order.

If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.

***array\_expression***

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

**Argument Types**

ARRAY\_AGG accepts two arguments. The first argument is an expression evaluating to a data type that is currently being used as the element value of a defined ARRAY data type. The second argument is an expression evaluating to a defined ARRAY data type whose element type matches that of the first argument.

**Usage Notes**

Because you can pass any *array\_expression*, the ARRAY type value that you pass may or may not contain any initialized elements. However, ARRAY\_AGG ignores any initialized elements in the *array\_expression*. The *array\_expression* is used only for providing the data type of the result value.

Recommendation: Pass a blank instance of the desired ARRAY type value for this parameter, such as NEW myArray().

If the *array\_expression* argument passed to ARRAY\_AGG is a NULL literal, then an error is returned. If the *array\_expression* argument can be resolved to an ARRAY data type, then the function executes normally, even if the data contains a NULL.

If the *value\_expression* argument for the current row being processed evaluates to NULL, then that expression will not be eliminated from the list of values being aggregated, but will be used to set the appropriate ARRAY element value just as would be done with a non-NULL element. This follows the ANSI SQL:2011 standard and is noted in the standard as a difference between ARRAY\_AGG and other aggregate functions.

ARRAY\_AGG is created with a CLASS AGGREGATE value of 64000, meaning that the intermediate aggregate storage will be allocated at that size. Since the intermediate aggregate storage is a fixed-length value, it may be possible in cases of processing a very large input set that this buffer may overflow. All aggregate UDFs have this limitation.

**Result Value and Rules**

ARRAY\_AGG returns an ARRAY type value whose elements are the aggregated set of element expression values that were passed in as the first parameter. For an n-D ARRAY result value, the *value\_expression* values that are aggregated into the array are entered in row-major order.

If you specify the ORDER BY clause in the first argument, then the values input to the resultant array are ordered by the column values specified in this clause. The following rules apply to the ORDER BY clause of ARRAY\_AGG:

- Only one value may be passed in the ORDER BY clause of ARRAY\_AGG. This is different from an ORDER BY in a SELECT statement, where multiple values may be passed for ORDER BY.
- An ARRAY column value may not be passed as the sort key in the ORDER BY clause. You cannot make relational comparisons of ARRAY values. Any other data type that cannot be compared, such as a BLOB or CLOB, may not be passed as the sort key in the ORDER BY clause.
- For the single value passed to ORDER BY, any expression that may be relationally comparable may be passed. However, if the value passed is not a column value, then sorting of the resultant array elements will not occur and the results will be written to the resultant array in random order. Use a column value from one of the referenced tables in the SQL statement for proper sorting to occur.

## Example: Inserting Data from a Source Table Into a New Table

This example takes data from a source table which records a phone number associated with an employee and inserts it into a new table which records phone numbers in an ARRAY type. In many cases, an employee may have multiple phone numbers, such as office phone, mobile phone, home phone, etc. A better way to represent this type of data is with a 1-D ARRAY type which records multiple phone numbers. This reduces the number of rows in the table.

Note that the same type of behavior can be illustrated with an n-D ARRAY type. The only difference is that the storage of elements is done in row-major order.

The following statement creates a 1-D ARRAY type that can hold up to 100 phone number values:

```
CREATE TYPE emp_phone_array AS VARCHAR(14) ARRAY[100];
```

The following source table contains one row per employee phone number.

```
CREATE SET TABLE employee
  (emp_id INTEGER,
   emp_name VARCHAR(30),
   emp_phone CHAR(14));
SELECT * FROM employee;
```

emp_id	emp_name	emp_phone
1	Beth	(619) 619-6190
1	Beth	(619) 620-6200
1	Beth	(619) 720-7200
2	Greg	(858) 858-8580
2	Greg	(858) 859-8590
2	Greg	(858) 860-8600
3	Louise	(421) 421-4210

3	Louise	(421) 422-4220
3	Louise	(421) 423-4230

The following target table contains one row per employee and stores all phone numbers associated with an employee in an ARRAY type.

```
CREATE SET TABLE employeePhoneInfo
(emp_id INTEGER,
 emp_name VARCHAR(30),
 emp_phone emp_phone_array);
INSERT INTO employeePhoneInfo
SELECT emp_id, emp_name,
       ARRAY_AGG(emp_phone, NEW emp_phone_array())
FROM employee GROUP BY emp_id,emp_name
WHERE emp_id < 100;
SELECT * FROM employeePhoneInfo;
```

The result is:

emp_id	emp_name	emp_phone
-----	-----	-----
1	Beth	( (619) 619-6190, (619) 620-6200,
(619) 720-7200 )		
2	Greg	( (858) 858-8580, (858) 859-8590,
(858) 860-8600 )		
3	Louise	( (421) 421-4210, (421) 422-4220,
(421) 423-4230 )		

# UNNEST

A table function for use with ARRAY data types that allows arrays to be converted into column tables.

UNNEST returns a table that contains one, two, or three columns. The first output column corresponds to the optional key value, if it was specified. The next output column has the same data type as the element type of the ARRAY type that was passed in. If the optional WITH ORDINALITY clause was specified, then UNNEST generates an additional integer column that contains the position associated with each element.

The table is populated with one row for each element of the ARRAY value passed to UNNEST. For an n-D ARRAY input value, the column containing the element values from the input n-D ARRAY are output in row-major order.

## ANSI Compliance

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

## UNNEST Syntax

```
[TD_SYSFNLIB.] UNNEST ( [ key_expr, ] array_expr ) [ WITH ORDINALITY ]
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *key\_expr*

An expression that evaluates to a Teradata data type, except for ARRAY, UDT, Period data type, BLOB, or CLOB.

#### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

## Argument Types

UNNEST accepts either 1 or 2 arguments. The optional key value argument allows for grouping the rows produced in the result table. This is especially helpful when using UNNEST as a complement of the ARRAY\_AGG function, because ARRAY\_AGG may be invoked with the aggregate GROUP BY clause to produce multiple ARRAY values based on the grouping.

The *array\_expr* argument is an expression that evaluates to an ARRAY data type that is currently defined in the system.

## Usage Notes

UNNEST is a table function; therefore, it can only appear in the FROM clause of an SQL SELECT statement.

If the ARRAY expression argument evaluates to NULL, then no rows will be returned in the result table. If the ARRAY expression argument is not NULL but has one or more elements that are not present, then the column value for the array element in the corresponding result row will be NULL.

## Example: Calling UNNEST

This example calls UNNEST, passing the optional key value argument and using the optional WITH ORDINALITY clause, using a column of 1-D ARRAY type myarray. The example returns 20 rows, one for each value stored in the myarray array for each row of the tt2 table. The result has these three columns:

- out\_key is the optional key value passed to UNNEST. In this example, it corresponds to the value of the int\_key column in the tt2 table, so it identifies the array values in the results that come from each row of the tt2 table.
- pos is the position of an element within the array. It is included in the results because the example uses the WITH ORDINALITY clause.
- val is the value of the element in the array.

```
CREATE TYPE myarray AS INTEGER ARRAY[10];
CREATE TABLE tt2(
    pkey INTEGER,
    int_key INTEGER,
    vc_key VARCHAR(20),
    myarr myarray);
INS INTO tt2 VALUES(0, 0, 'item 0', NEW myarray(10, 20, 30, 40, 50));
INS INTO tt2 VALUES(1, 1, 'item 1', NEW myarray(11, 21, 31, 41, 51));
INS INTO tt2 VALUES(2, 2, 'item 2', NEW myarray(12, 22, 32, 42, 52));
INS INTO tt2 VALUES(3, 3, 'item 3', NEW myarray(NULL, 23, 33, 43, NULL));
INS INTO tt2 VALUES(4, 4, 'item 4', NULL);
SELECT out_key, tf.pos, tf.val
FROM tt2,
    TABLE (UNNEST(tt2.int_key, tt2.myarr) WITH ORDINALITY) AS tf(out_key,
val, pos) WHERE tt2.int_key = tf.out_key ORDER BY 1,2;
*** Query completed. 20 rows found. 3 columns returned.
```

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

out_key	pos	val
0	1	10
0	2	20
0	3	30
0	4	40
0	5	50
1	1	11
1	2	21
1	3	31
1	4	41
1	5	51
2	1	12
2	2	22

2	3	32
2	4	42
2	5	52
3	1	?
3	2	23
3	3	33
3	4	43
3	5	?

## Related Information

For more information, see [ARRAY Constructor Expression](#).

## CARDINALITY

Returns an integer representing the number of elements in an ARRAY data type that currently have assigned values, or the number of elements that are initialized within a specific scope reference.

CARDINALITY returns an integer value that represents the number of elements in the ARRAY that currently have assigned values. This count includes elements that are NULL. Since an ARRAY value may have fewer element values assigned than defined for its maximum size  $n$ , the CARDINALITY function may return a value that is smaller than  $n$ .

The function returns 0 if the array argument is empty (that is, it does not have any elements assigned). If the array argument is NULL, then CARDINALITY returns NULL as the result.

Even though an ARRAY function normally returns an error if any uninitialized elements fall within the range of the specified scope\_reference in any of the ARRAY input arguments, an ARRAY expression specified within a CARDINALITY function behaves differently: an explicit scope that includes uninitialized elements is an acceptable input when an ARRAY expression is specified in a CARDINALITY function.

## ANSI Compliance

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

## CARDINALITY Syntax

### System Function

```
CARDINALITY ( array_expr [ , scope_reference ] )
```

### Method-Style

```
array_expr.CARDINALITY ( [ scope_reference ] )
```

## Syntax Elements

### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

### *scope\_reference*

An ARRAY scope reference.

An explicit scope that includes uninitialized elements is acceptable input when an ARRAY expression is specified in a CARDINALITY function.

## Argument Type

The *array\_expr* argument is an expression that evaluates to an ARRAY data type that is currently defined in the system.

## Example: Return an Integer that Represents the Number of Elements in the 1-D ARRAY

In the following example, an integer value is returned that represents the number of elements in the 1-D ARRAY that are currently filled in.

```
CREATE TYPE phonenumbers AS CHAR(10) ARRAY[20];
CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
SELECT CARDINALITY(phonelist)
FROM employee_info;
```

The CARDINALITY function can also be used to return the number of elements that are currently initialized within a specific scope reference. For example, the following query returns the number of initialized elements within the range from 5 to 10 of the phonelist array.

```
SELECT CARDINALITY(phonelist, 5, 10);
```

Consider the following 2-D ARRAY data type:



```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];
CREATE TABLE seismic_table(shots shot_ary);
CREATE TABLE seismic_data (
  id INTEGER,
  shot1 shot_ary,
  shot2 shot_ary);
```

In the following example, an integer value is returned that represents the number of elements in the 2-D ARRAY that are currently filled in.

```
SELECT CARDINALITY(shot1) FROM seismic_data;
```

The following shows the same query using method-style syntax:

```
SELECT shot1.CARDINALITY() FROM seismic_data;
```

The following query returns the number of initialized elements within the range from 5 to 10 on each dimension of the phonelist array.

```
SELECT CARDINALITY(phonelist, NEW ArrayVec(5,5), NEW ArrayVec(10,10));
```

## Related Information

For more information, see [ARRAY\\_Constructor\\_Expression](#).

## ARRAY\_CONCATENATION\_OPERATOR

Concatenates one-dimensional ARRAY data types.

The result of the concatenation operation is a new 1-D ARRAY value of the same type as the data type of the two arguments, where all of the elements present in *array\_expr1* are followed by all the elements that are present in *array\_expr2*. Elements that are present include NULL elements, but do not include elements that are in an uninitialized state.

Since the result value and the argument values are of the same 1-D ARRAY type, they have the same maximum size *n*. Therefore, if the number of elements that results from the concatenation of *array\_expr1* and *array\_expr2* is greater than the maximum size *n* of the defined 1-D ARRAY type, the operation will abort with an error.

If either argument is NULL, the result of the operation is NULL.

The ARRAY concatenation operator cannot be used with multidimensional ARRAY data types.

## ANSI Compliance

The ARRAY concatenation operator for one-dimensional (1-D) ARRAY data types is partially ANSI SQL:2011 compliant.

The ARRAY concatenation operator requires that both operands are instances of the same ARRAY type and that the target type of the concatenation operation is also the same type. This is a deviation from the ANSI standard because it defines the target data type of a 1-D ARRAY concatenation to be a new 1-D ARRAY type with the length defined as the sum of the length of both operands.

## ARRAY\_CONCATENATION\_OPERATOR Syntax

```
array_expr1 || array_expr2
```

### Syntax Elements

**array\_expr1**

**array\_expr2**

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

**||**

The concatenation operator.

### Argument Type

*array\_expr1* and *array\_expr2* must have the same 1-D ARRAY data type as defined by a CREATE TYPE statement.

## Example: Concatenating 1-D ARRAY values

This example takes two 1-D ARRAY values of the same data type and concatenates them together.

```
CREATE TYPE address AS CHAR(10) ARRAY[5];
CREATE TABLE employee_info (eno INTEGER, addressval address);
/* Assume one row in table employee_info contains the following value
for "addressval":
Addressval[1] = '123 Main St.'
Addressval[2] = 'San Diego'
```

```

Addressval[3] = 'CA'
*/
/* The following select statement concatenates the current value in
"addressval" with ZIP code and country information, which are stored as
additional elements of the 1-D array. */
SELECT addressval || NEW address('92101', 'USA');
FROM employee_info;
/* Result value is the following:
Addressval[1] = '123 Main St.'
Addressval[2] = 'San Diego'
Addressval[3] = 'CA'
Addressval[4] = '92101'
Addressval[5] = 'USA'
*/

```

## ARRAY\_CONCATENATION\_FUNCTION

Concatenates one-dimensional ARRAY data types and can be applied to a subset of the elements of the array.

The result of the concatenation operation is a new 1-D ARRAY value of the same type as the data type of the two arguments, where all of the elements present in *expr1* are followed by all the elements that are present in *expr2*. Elements that are present include NULL elements, but do not include elements that are in an uninitialized state.

Since the result value and the argument values are of the same 1-D ARRAY type, they have the same maximum size *n*. Therefore, if the number of elements that results from the concatenation of *expr1* and *expr2* is greater than the maximum size *n* of the defined 1-D ARRAY type, the operation will abort with an error.

If either argument is NULL, the result of the operation is NULL.

The ARRAY concatenation function cannot be used with multidimensional ARRAY data types.

### ANSI Compliance

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

## ARRAY\_CONCATENATION\_FUNCTION Syntax

### System Function

```

ARRAY_CONCAT ( expr_1, expr_2 [, scope_reference ] )

```

## Method-Style

```
expr_1.ARRAY_CONCAT ( expr_2 [, scope_reference ] )
```

## Syntax Elements

### *expr1*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

### *scope\_reference*

An ARRAY scope reference.

## Argument Type

*expr1* and *expr2* must be ARRAY expressions, and they must have the same 1-D ARRAY data type as defined by a CREATE TYPE statement.

## Example: Concatenating Two 1-D ARRAY Values

This example takes two 1-D ARRAY values of the same data type and concatenates them together. PhoneNum and OldPhoneNum are columns in the employee\_info table, and both columns have the same 1-D ARRAY data type.

```
/* The following select statement concatenates the current value in "PhoneNum"
with the value in "OldPhoneNum". */
/* Assume one row in table employee_info contains the following value
for "PhoneNum":
PhoneNum[1] = '6197211000'
PhoneNum[2] = '6197221000'
PhoneNum[3] = '6197231000'
Also, assume one row in table employee_info contains the following value
for "OldPhoneNum":
OldPhoneNum[1] = '8582001000'
OldPhoneNum[2] = '8582002000'
OldPhoneNum[3] = '8582003000'
*/
SELECT ARRAY_CONCAT(PhoneNum, OldPhoneNum)
```

```

FROM employee_info;
/* Result value is the following:
PhoneNum[1] = '6197211000'
PhoneNum[2] = '6197221000'
PhoneNum[3] = '6197231000'
PhoneNum[4] = '8582001000'
PhoneNum[5] = '8582002000'
PhoneNum[6] = '8582003000'
*/

```

The following shows the same query using method-style syntax:

```

SELECT PhoneNum.ARRAY_CONCAT(OldPhoneNum)
FROM employee_info;

```

## ARRAY\_COMPARISON\_FUNCTION

Performs comparisons on the individual elements of an ARRAY value, whether referred to in its entirety or over a scope reference.

### ANSI Compliance

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

## ARRAY\_COMPARISON\_FUNCTION Syntax

### System Function

```

{ ARRAY_GT |
  ARRAY_GE |
  ARRAY_LT |
  ARRAY_LE |
  ARRAY_EQ |
  ARRAY_NE
} ( expr1, expr2 [, scope_reference ] )

```

### Method-Style

```

expr1.{ ARRAY_GT |
        ARRAY_GE |
        ARRAY_LT |
        ARRAY_LE |
        ARRAY_EQ |

```

```

ARRAY_NE
} ( expr2 [, scope_reference ] )

```

## Syntax Elements

### *expr1*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

### *scope\_reference*

An ARRAY scope reference.

## Argument Type

Either *expr1* or *expr2* or both arguments must be an ARRAY expression. For details, see [Usage Notes](#).

## Usage Notes

For the relational functions ARRAY\_GT, ARRAY\_GE, ARRAY\_LT, ARRAY\_LE, ARRAY\_EQ, and ARRAY\_NE, two forms are supported for ARRAY types.

In the first form, both left-hand side (LHS) and right-hand side (RHS) operands are two instances of the same ARRAY type.

The relational operation is performed in a pairwise fashion, one by one for each matching pair of elements in the two arguments. If the optional argument *scope\_reference* is defined, then the operation is performed only over the elements within the scope. All elements outside the boundaries of *scope\_reference* are set to NULL.

The result is an ARRAY of the same type as the input array arguments, with three possible values: 1 (true), 0 (false), or NULL for all numeric types except DECIMAL(m,n) where m=n. For DECIMAL(m,n) where m=n, the function cannot store a value of 1 because no digits are permitted to the LHS of the decimal point. In this case, the function returns .9[0]. Any higher level of precision above 1 is padded with zero to the right.

In the second form, one of either LHS or RHS is an ARRAY with a valid Teradata data type, and the other operand is a numeric value of the same data type as the one defined for the ARRAY in LHS.

The relational operation is performed by applying the numeric value argument to each element of the ARRAY argument. If the optional argument *scope\_reference* is defined, then the operation is performed only over the elements within the scope. All elements outside the boundaries of *scope\_reference* are set to NULL.

The result is an ARRAY of the same type as the input array argument, with three possible values: 1 (true), 0 (false), or NULL for all numeric types except DECIMAL(m,n) where m=n. For DECIMAL(m,n) where m=n, the function cannot store a value of 1 because no digits are permitted to the LHS of the decimal point. In this case, the function returns .9[0]. Any higher level of precision above 1 is padded with zero to the right. If RHS evaluates to NULL, then NULL is returned.

For both forms of the relational comparison functions, if a NULL element is encountered in an ARRAY argument, the resulting ARRAY will contain a NULL element in that position. Comparison involving arrays are not affected by the optional DEFAULT NULL clause in the CREATE TYPE statement. However, the probability of getting an error is increased when comparing arrays that have been created without the DEFAULT NULL clause.

If an ARRAY argument contains any elements that are in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the ARRAY that contains uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See [OEXTEND](#).

## Supported ARRAY Comparison Functions

The following table provides a description of the supported ARRAY comparison functions.

Relational Function	Description
ARRAY_GT	Greater than function. Compares two expressions and determines whether <i>expr1</i> is greater than <i>expr2</i> . If it is, the function returns a non-zero value in the corresponding result ARRAY element. If <i>expr1</i> is less than or equal to <i>expr2</i> , the function returns zero in the corresponding result ARRAY element.
ARRAY_GE	Equal or greater than function. Compares two expressions and determines whether <i>expr1</i> is greater than or equal to <i>expr2</i> . If it is, the function returns a non-zero value in the corresponding result ARRAY element. If <i>expr1</i> is less than <i>expr2</i> , the function returns zero in the corresponding result ARRAY element.
ARRAY_LT	Less than function. Compares two expressions and determines whether <i>expr1</i> is less than <i>expr2</i> . If it is, the function returns a non-zero value in the corresponding result ARRAY element. If <i>expr1</i> is greater than or equal to <i>expr2</i> , the function returns zero in the corresponding result ARRAY element.
ARRAY_LE	Equal or less than function. Compares two expressions and determines whether <i>expr1</i> is less than or equal to <i>expr2</i> . If it is, the function returns a non-zero value in the corresponding result ARRAY element. If <i>expr1</i> is greater than <i>expr2</i> , the function returns zero in the corresponding result ARRAY element.
ARRAY_EQ	Equality function. Compares two expressions and determines whether <i>expr1</i> is equal to <i>expr2</i> . If it is, the function returns a non-zero value in the corresponding result ARRAY element. If <i>expr1</i> is not equal to <i>expr2</i> , the function returns zero in the corresponding result ARRAY element.
ARRAY_NE	Non-equality function. Compares two expressions and determines if <i>expr1</i> is not equal to <i>expr2</i> . If the two expressions are not equal, the function returns a non-zero value in the corresponding result ARRAY element. If <i>expr1</i> is equal to <i>expr2</i> , the function returns zero in the corresponding result ARRAY element.

## Restrictions

You cannot make relational comparisons of ARRAY values. You cannot use any of the relational comparison operators on ARRAY data. You can only make relational comparisons on individual elements of an ARRAY.

## Examples

### Example: Querying a 1-D ARRAY Data Type and Table

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS DECIMAL(7,2) ARRAY[10];
CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price item_price);
```

Assume the following element values for the sale\_price and regular\_price arrays:

```
sale_price[1:10] = 50, 100, 200, 90, 250, 550, 200, 200, 50, 75
regular_price[1:10] = 100, 300, 230, 110, 500, 550, 200, 400, 100, 150
```

The following query returns a 1-D ARRAY with scope reference [1:10] of BYTEINT element type. During evaluation, each element within the specified scope, [5:10], in the regular\_price array is compared with the corresponding element of the sale\_price array using the relational function ARRAY\_GT. The resulting values in the output array are 0 or a non-zero number for the elements within the scope reference, and NULL for all other elements of the array.

```
SELECT ARRAY_GT(regular_price, sale_price, 5,10)
FROM inventory;
```

The output from the query is a 1-D ARRAY with the following values:

```
output_array[1:10] = [ NULL, NULL, NULL, NULL, 1, 0, 0, 1, 1, 1]
```

The following query returns a 1-D ARRAY with scope reference [1:10] of type item\_price. During evaluation, each element within the specified scope [5:10] in the regular\_price array is compared with the corresponding element of the sale\_price array using the relational function ARRAY\_GT. The resulting ARRAY of element type BYTEINT value (0 or non-zero number) is multiplied by the corresponding element value of the regular\_price array.



```
SELECT ARRAY_MUL(regular_price, ARRAY_GT(sale_price, regular_price, 5, 10))
FROM inventory;
```

The output of the query is a 1-D ARRAY with the following values:

```
output_array[1:10] = [ NULL, NULL, NULL, NULL, 500, 0, 0, 400, 100, 150]
```

In the following query, the relational function `ARRAY_LT` compares each element within the scope of the 1-D ARRAY `sale_price` with a literal value of 100. If the element value is less than 100, the comparison function returns a non-zero value, otherwise it returns 0 if the condition is not satisfied, or NULL if the element is not initialized. A 1-D ARRAY of `item_price` ARRAY type is returned with the corresponding values for each element of the input 1-D ARRAY.

```
SELECT ARRAY_LT(sale_price,100)
FROM inventory;
```

The output of the query is a 1-D ARRAY with the following values:

```
output_array[1:10] = [ 1, 0 , 0, 1, 0, 0, 0, 0, 1, 1]
```

## Example: Querying a 2-D ARRAY Data Type and Table

Consider the following 2-D ARRAY data type and table:

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER DEFAULT NULL;
CREATE TABLE seismic_data (
  id INTEGER,
  shot1 shot_ary,
  shot2 shot_ary);
```

The following query returns a 2-D ARRAY with an element type of `INTEGER`. The size of the output array is the same as that of the input array argument. During evaluation, the element in position `[5][10]` of the `shot1` array is compared to a value of 5. If the element is greater than 5, the value for the corresponding element in the output array is set to a non-zero value, otherwise it is set to 0. All other elements in the output array are set to NULL.

```
SELECT ARRAY_GT(shot1, 5, NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```

The following is the same query using method-style syntax:

```
SELECT shot1.ARRAY_GT(5, NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```

In the following query, the relational function `ARRAY_LT` compares each element within the scope reference `[3:5][8:10]` of the 2-D ARRAY `shot1` with a literal value of 0. If the element value is less than 0 the comparison function returns 0, otherwise it returns a non-zero value. The resulting array of `shot_ary` type is then multiplied by the `shot1` array.

```
SELECT ARRAY_MUL(shot1, ARRAY_LT(shot1, 0, NEW arrayVec(3,8),
NEW arrayVec(5,10)))
FROM seismic_data;
```

## ARRAY\_ARITHMETIC\_FUNCTION

Performs arithmetic operations (add, subtract, multiply, divide, and mod) on an ARRAY value, whether referred to in its entirety or over a scope reference.

The result is an ARRAY of the same size as the maximum cardinality of the input array arguments, and the element type is the same as that of *expr1*. Note that the size of the result array type refers to its current cardinality.

### ANSI Compliance

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

## ARRAY\_ARITHMETIC\_FUNCTION Syntax

### System Function

```
{ ARRAY_ADD |
  ARRAY_SUB |
  ARRAY_MUL |
  ARRAY_DIV |
  ARRAY_MOD
} ( expr1, expr2 [, scope_reference ] )
```

### Method-Style

```
expr1.{ ARRAY_ADD |
        ARRAY_SUB |
        ARRAY_MUL |
        ARRAY_DIV |
        ARRAY_MOD
} ( expr2 [, scope_reference ] )
```

## Syntax Elements

*expr1*

*expr2*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

The element type of the array must be a numeric data type.

*scope\_reference*

An ARRAY scope reference.

## Argument Types

Valid arguments for the ARRAY arithmetic functions are one of the following:

- *expr1* and *expr2* are each an instance of the same ARRAY type.
- *expr1* is an ARRAY value with a numeric argument type and *expr2* is a numeric value that can be assigned to the element type of *expr1*.

If either *expr1* or *expr2* is NULL, the function returns NULL.

## Usage Notes

For the arithmetic functions ARRAY\_ADD, ARRAY\_SUB, ARRAY\_MUL, ARRAY\_DIV, and ARRAY\_MOD, two forms are supported for ARRAY types.

In the first form, both left-hand side (LHS) and right-hand side (RHS) operands are two instances of the same ARRAY type.

The arithmetic operation is performed in a pairwise fashion, one by one for each matching pair of elements in the two arguments. If the optional argument *scope\_reference* is defined, then the operation is performed only over the elements within the scope. All elements outside the boundaries of *scope\_reference* are set to NULL. The result is an ARRAY of the same data type as the input ARRAY arguments, with cardinality (number of populated elements) the same as the first input ARRAY argument.

In the second form, one of either LHS or RHS is an ARRAY value with a numeric argument type, and RHS is a numeric value that can be assigned to the element type of LHS. The arithmetic operation is performed by applying the RHS value with the arithmetic operation to each element of the ARRAY argument on the LHS. The result is an ARRAY of the same type as the input ARRAY argument, with cardinality (number of populated elements) the same as the input ARRAY argument. If the optional argument *scope\_reference*

is defined, then the operation is performed only over the elements within the scope. All elements outside the boundaries of *scope\_reference* are set to NULL.

For both forms of the arithmetic functions, if a NULL element is encountered in an ARRAY argument, the resulting ARRAY will contain a NULL element in that position. Arithmetic operations involving arrays are not affected by the optional DEFAULT NULL clause in the CREATE TYPE statement. However, the probability of getting an error is increased when comparing arrays that have been created without the DEFAULT NULL clause.

If an ARRAY argument contains any elements that are in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the ARRAY that contains uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See [OEXTEND](#).

## Examples

### Example: Querying a 1-D ARRAY Data Type and Table using ARRAY

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS DECIMAL(7,2) ARRAY[20];
CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price_diff item_price);
```

Assume the following element values for the sale\_price\_diff and regular\_price arrays:

```
sale_price_diff[1:20] = 100, 200, 120, 140, 50, 160, 45, 10, 90, 100
                        50, 100, 200, 90, 250, 550, 200, 200, 50, 75
regular_price[1:20] = 50, 90, 80, 10, 45, 30, 20, 10, 90, 100,
                     100, 300, 230, 110, 500, 550, 200, 400, 100, 150
```

The following query returns a 1-D ARRAY of element type item\_price. During evaluation, each element within the specified scope in the regular\_price array is combined with the corresponding element of the sale\_price\_diff array using the arithmetic function ARRAY\_SUB.

```
SELECT ARRAY_SUB(regular_price, sale_price_diff, 10, 20)
FROM inventory;
```

The query returns a 1-D ARRAY with the following values:

```
output_array[1:20] = [ NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
                      NULL, 50, 200, 30, 20, 250, 0, 0, 200, 50, 75]
```

In the following query, a literal value of 2 is added to all elements in the 1-D ARRAY `sale_price_diff`.

```
SELECT ARRAY_ADD(sale_price_diff, 2) FROM inventory;
```

This query returns a 1-D ARRAY with the following values:

```
output_array[1:20] = [ 102, 202, 122, 142, 52, 162, 47, 12, 92, 102
                      52, 102, 202, 92, 252, 552, 202, 202, 52, 77]
```

The following query shows the use of a filtering condition while performing arithmetic operations on an ARRAY. In this example, all elements within the scope [10:20] that have a value less than 1000 are multiplied by 2.

```
SELECT ARRAY_MUL(regular_price, 2, 10, 20) FROM inventory
WHERE ARRAY_COUNT_DISTINCT(ARRAY_LT(regular_price,1000,10,20),1)>1;
```

This query returns a 1-D ARRAY with the following values:

```
output_array[1:20] = [NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
                     NULL, 200, 600, 460, 220, 1000, 1100, 400, 800, 200, 300]
```

## Example: Querying a 2-D ARRAY Data Type and Table using ARRAY

Consider the following 2-D ARRAY data type and table:

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER DEFAULT NULL;
CREATE TABLE seismic_data (
  id INTEGER,
  shot1 shot_ary,
  shot2 shot_ary);
```

The following query returns an ARRAY of type `shot_ary` with elements in scope reference [10:20][10:20] modified as a result of the operation. During evaluation, each element within the specified scope reference in `shot1` is combined with the corresponding element of `shot2` using the arithmetic function `ARRAY_ADD`.

```
SELECT ARRAY_ADD(shot1, shot2, NEW arrayVec(10, 20),
                NEW arrayVec(10,20))
FROM seismic_data;
```

In the following query, a literal value of 9 is added to all elements within the specified scope reference of the `shot1` array.

```
SELECT ARRAY_ADD(shot1, 9, NEW arrayVec(10,10), NEW arrayVec(20,20))
FROM seismic_data;
```

The following is the same query using method-style syntax:

```
SELECT shot1.ARRAY_ADD(9, NEW arrayVec(10,10), NEW arrayVec(20,20))
FROM seismic_data;
```

The following query shows the use of a filtering condition while performing arithmetic operations on an n-D ARRAY. In this example, all elements within the scope reference [10:20][10:20] that have a negative value are multiplied by zero.

```
SELECT ARRAY_MUL(shot1, 0, NEW arrayVec(10,10), NEW arrayVec(20,20))
FROM seismic_data
WHERE ARRAY_COUNT_DISTINCT(ARRAY_LT(shot1, 0, NEW arrayVec(10,10),
NEW arrayVec(20,20)),0)>1;
```

## Related Information

For more information, see [“ARRAY Constructor Expression”](#).

## ARRAY\_SUM

Returns a value representing the total sum of adding the values of each element in *array\_expr* or the sum of the elements within the specified scope.

The result type of ARRAY\_SUM is NUMBER unless the element type of the array argument is FLOAT. In this case, the result type is FLOAT.

### ANSI Compliance

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

## ARRAY\_SUM Syntax

### System Function

```
ARRAY_SUM ( array_expr [ scope_reference ] )
```

### Method-Style

```
array_expr.ARRAY_SUM ( [ scope_reference ] )
```

## Syntax Elements

### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

The element type of the array must be a numeric data type.

### *scope\_reference*

An ARRAY scope reference.

## Usage Notes

ARRAY\_SUM takes an array expression as an argument and returns a scalar value representing the total sum of adding the values of each element in the array argument. The array argument must have a numeric element type. If you specify a scope reference, the summation is applied only to the elements within the limits of the given scope.

The affected elements cannot be filtered through a conditional expression. Therefore, a SELECT statement involving ARRAY\_SUM must not contain an array relational expression in the WHERE clause.

If a NULL element is encountered in the array argument, it is ignored and not considered when doing the calculation. If the array argument contains any elements that is in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the array with uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See [OEXTEND](#).

If *array\_expr* is NULL, the function returns NULL.

## Examples

### Example: Querying a 1-D ARRAY Data Type and Table using ARRAY\_SUM

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS DECIMAL(7,2) ARRAY[20];
CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price item_price);
```

When evaluating the following query, each element value of the `regular_price` array is added together. The resulting value is divided by 2.

```
SELECT ARRAY_SUM(regular_price) / 2 FROM inventory;
```

The following is the same query using method-style syntax.

```
SELECT regular_price.ARRAY_SUM() / 2 FROM inventory;
```

In the following query, `ARRAY_SUM` adds each element within the specified scope of the `regular_price` array. The query returns a value representing the total sum of adding the affected element values of the `regular_price` array.

```
SELECT ARRAY_SUM(regular_price, NEW arrayVec(5,10))FROM inventory;
```

The following shows an alternate way to specify the same query.

```
SELECT ARRAY_SUM(regular_price, 5, 10) FROM inventory;
```

## Example: Querying a 2-D ARRAY Data Type and Table using `ARRAY_SUM`

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];
CREATE TABLE seismic_data (
  id INTEGER,
  shot1 shot_ary,
  shot2 shot_ary);
```

When evaluating the following query, each element value in the `shot1` array is added together. The resulting value is divided by 2.

```
SELECT ARRAY_SUM(shot1) / 2 FROM seismic_data;
```

In the following query, `ARRAY_SUM` adds each element within the specified scope reference of the `shot1` array. The query returns a value representing the total sum of adding the affected element values of the `shot1` array.

```
SELECT ARRAY_SUM(shot1, NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```



## Related Information

For more information, see [ARRAY\\_Constructor\\_Expression](#).

## ARRAY\_AVG

Returns the average of all the element values in *array\_expr* or the average of the elements within the specified scope.

The result data type of ARRAY\_AVG is FLOAT.

### ANSI Compliance

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

## ARRAY\_AVG Syntax

### System Function

```
ARRAY_AVG ( array_expr [ scope_reference ] )
```

### Method-Style

```
array_expr.ARRAY_AVG ( [ scope_reference ] )
```

### Syntax Elements

#### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

The element type of the array must be a numeric data type.

#### *scope\_reference*

An ARRAY scope reference.

## Usage Notes

ARRAY\_AVG takes an array expression as an argument and returns a scalar value of type FLOAT. This value represents the result of adding the values of each element in the array argument, and dividing this

result by the total number of elements. The array argument must have a numeric element type. If you specify a scope reference, the function is applied only to the elements within the limits of the given scope.

The affected elements cannot be filtered through a conditional expression. Therefore, a SELECT statement involving ARRAY\_AVG must not contain an array relational expression in the WHERE clause.

If a NULL element is encountered in the array argument, it is ignored and not considered when doing the calculation. If the array argument contains any elements that is in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the array with uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See [OEXTEND](#).

If *array\_expr* is NULL, the function returns NULL.

## Examples

### Example: Querying a 1-D ARRAY Data Type and Table using ARRAY\_AVG

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS DECIMAL(7,2) ARRAY[20];
CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price item_price);
```

When evaluating the following query, each element value of the regular\_price array is added. The total sum of all the element values is then divided by the number of elements in the regular\_price array.

```
SELECT ARRAY_AVG(regular_price) FROM inventory;
```

In the following query, ARRAY\_AVG adds each element within the specified scope of the regular\_price array. The result is a scalar value representing the total sum of adding the affected element values of regular\_price divided by the total number of elements composing the specified scope.

```
SELECT ARRAY_AVG(regular_price, 5, 10) FROM inventory;
```

### Example: Querying a 2-D ARRAY Data Type and Table using ARRAY\_AVG

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];
CREATE TABLE seismic_data (
    id INTEGER,
```

```
shot1 shot_ary,  
shot2 shot_ary);
```

When evaluating the following query, each element value in the shot1 array is added. The total sum of all the element values is then divided by the number of elements in the shot1 array.

```
SELECT ARRAY_AVG(shot1) FROM seismic_data;
```

In the following query, ARRAY\_AVG adds each element within the specified scope reference of the shot1 array. The result is a scalar value representing the total sum of adding the affected element values of shot1 divided by the total number of elements composing the specified scope reference.

```
SELECT ARRAY_AVG(shot1, NEW arrayVec(5,5), NEW arrayVec(10,10))  
FROM seismic_data;
```

The following is the same query using method-style syntax.

```
SELECT shot1.ARRAY_AVG(NEW arrayVec(5,5), NEW arrayVec(10,10))  
FROM seismic_data;
```

## ARRAY\_MAX

Returns the maximum of all element values in *array\_expr* or the maximum of the elements within the specified scope.

The result type is the same as the element type of the array argument.

### ANSI Compliance

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

## ARRAY\_MAX Syntax

### System Function

```
ARRAY_MAX ( array_expr [, scope_reference ] )
```

### Method-Style

```
array_expr.ARRAY_MAX ( [ scope_reference ] )
```

## Syntax Elements

### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

The element type of the array must be a numeric data type.

### *scope\_reference*

An ARRAY scope reference.

## Usage Notes

ARRAY\_MAX takes an array expression as an argument and returns an element value that represents the maximum value of all the elements composing the array. If you specify a scope reference, the function is applied only to the elements within the limits of the given scope.

The affected elements cannot be filtered through a conditional expression. Therefore, a SELECT statement involving ARRAY\_MAX must not contain an array relational expression in the WHERE clause.

If a NULL element is encountered in the array argument, it is ignored and not considered when doing the calculation. If the array argument contains any elements that is in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the array with uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See [OEXTEND](#).

If *array\_expr* is NULL, the function returns NULL.

## Examples

### Example: Querying a 1-D ARRAY Data Type and Table using ARRAY\_MAX

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS DECIMAL(7,2) ARRAY[20];
CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price item_price);
```

The following query returns an element value, which is the maximum of all the element values in the `regular_price` array.

```
SELECT ARRAY_MAX(regular_price) FROM inventory;
```

The following is the same query using method-style syntax.

```
SELECT regular_price.ARRAY_MAX() FROM inventory;
```

The following query returns an element value, which is the maximum of the element values within the specified scope of the `regular_price` array.

```
SELECT ARRAY_MAX(regular_price, 5, 10) FROM inventory;
```

## Example: Querying a 2-D ARRAY Data Type and Table using ARRAY\_MAX

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];
CREATE TABLE seismic_data (
  id INTEGER,
  shot1 shot_ary,
  shot2 shot_ary);
```

The following query returns an element value, which is the maximum of all the element values in the `shot1` array.

```
SELECT ARRAY_MAX(shot1) FROM seismic_data;
```

The following query returns an element value, which is the maximum of the element values within the specified scope of the `shot1` array.

```
SELECT ARRAY_MAX(shot1, NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```

## ARRAY\_MIN

Returns the minimum of all element values in *array\_expr* or the minimum of the elements within the specified scope.

The result type is the same as the element type of the array argument.

## ANSI Compliance

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

## ARRAY\_MIN Syntax

### System Function

```
ARRAY_MIN ( array_expr [ , scope_reference ] )
```

### Method-Style

```
array_expr.ARRAY_MIN ( [ , scope_reference ] )
```

### Syntax Elements

#### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

The element type of the array must be a numeric data type.

#### *scope\_reference*

An ARRAY scope reference.

## Usage Notes

ARRAY\_MIN takes an array expression as an argument and returns an element value that represents the minimum value of all the elements composing the array. If you specify a scope reference, the function is applied only to the elements within the limits of the given scope.

The affected elements cannot be filtered through a conditional expression. Therefore, a SELECT statement involving ARRAY\_MIN must not contain an array relational expression in the WHERE clause.

If a NULL element is encountered in the array argument, it is ignored and not considered when doing the calculation. If the array argument contains any elements that are in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the array with uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See [OEXTEND](#).

If *array\_expr* is NULL, the function returns NULL.

## Examples

### Example: Querying a 1-D ARRAY Data Type and Table using ARRAY\_MIN

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS DECIMAL(7,2) ARRAY[20];
CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price item_price);
```

The following query returns an element value, which is the minimum of all the element values in the regular\_price array.

```
SELECT ARRAY_MIN(regular_price) FROM inventory;
```

The following query returns an element value, which is the minimum of the element values within the specified scope of the regular\_price array.

```
SELECT ARRAY_MIN(regular_price, 5, 10) FROM inventory;
```

### Example: Querying a 2-D ARRAY Data Type and Table using ARRAY\_MIN

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];
CREATE TABLE seismic_data (
    id INTEGER,
    shot1 shot_ary,
    shot2 shot_ary);
```

The following query returns an element value, which is the minimum of all the element values in the shot1 array.

```
SELECT ARRAY_MIN(shot1) FROM seismic_data;
```

The following query returns an element value, which is the minimum of the element values within the specified scope of the shot1 array.

```
SELECT ARRAY_MIN(shot1, NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```

The following is the same query using method-style syntax.

```
SELECT shot1.ARRAY_MIN(NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```

## Related Information

For more information, see [ARRAY\\_Constructor\\_Expression](#).

## ARRAY\_COUNT\_DISTINCT

Returns the number of distinct elements in *array\_expr*, optionally matching a specific input value.

The result type is an INTEGER.

### ANSI Compliance

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

## ARRAY\_COUNT\_DISTINCT Syntax

### System Function

```
ARRAY_COUNT_DISTINCT ( array_expr [, scope_reference ] [, matching_expr ] )
```

### Method-Style

```
array_expr.ARRAY_COUNT_DISTINCT ( [ scope_reference ] [, matching_expr ] )
```

### Syntax Elements

#### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

The element type of the array must be a numeric data type.



***scope\_reference***

An ARRAY scope reference.

***matching\_expr***

An expression to be matched by one or more elements in the array.

## Usage Notes

ARRAY\_COUNT\_DISTINCT takes an array expression as an argument and returns a value that represents one of two possible outputs:

- If you specify the optional argument *matching\_expr*, then the function returns the number of elements in *array\_expr* whose values are equal to *matching\_expr*. If *matching\_expr* is NULL, the function returns the number of elements with NULLs.
- If you do not specify *matching\_expr*, then the function returns the number of distinct elements in *array\_expr*. If a NULL element is encountered in the array argument, it is ignored and not considered when doing the calculation.

If you specify a scope reference, the function is applied only to the elements within the limits of the given scope. You can use ARRAY\_COUNT\_DISTINCT with arrays of numeric as well as character element types.

If the array argument contains any elements that are in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the array with uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See [OEXTEND](#).

If *array\_expr* is NULL, the function returns NULL.

## Examples

### Example: Querying a 1-D ARRAY Data Type and Table using ARRAY\_COUNT\_DISTINCT

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS DECIMAL(7,2) ARRAY[20];
CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price item_price);
```

In the following query, ARRAY\_COUNT\_DISTINCT returns the number of elements whose value is equal to 100 within the scope reference of the regular\_price array.

```
SELECT ARRAY_COUNT_DISTINCT(regular_price, 5, 10, 100) FROM inventory;
```

The following is the same query using method-style syntax.

```
SELECT regular_price.ARRAY_COUNT_DISTINCT(5, 10, 100) FROM inventory;
```

The following query returns the number of distinct elements in the `regular_price` array.

```
SELECT ARRAY_COUNT_DISTINCT(regular_price) FROM inventory;
```

## Example: Querying a 2-D ARRAY Data Type and Table using `ARRAY_COUNT_DISTINCT`

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];
CREATE TABLE seismic_data (
  id INTEGER,
  shot1 shot_ary,
  shot2 shot_ary);
```

In the following query, `ARRAY_COUNT_DISTINCT` returns the number of elements whose value is equal to 100 within the specified scope of the `shot1` array.

```
SELECT ARRAY_COUNT_DISTINCT(shot1, NEW arrayVec(5,5), NEW arrayVec(10,10), 100)
FROM seismic_data;
```

## ARRAY\_GET

Returns the element value in *array\_expr* that corresponds to the specified index position.

The result type is the same as the element type of the array argument.

### ANSI Compliance

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

## ARRAY\_GET Syntax

### System Function

```
ARRAY_GET ( array_expr, array_index )
```

## Method-Style

```
array_expr.ARRAY_GET ( array_index )
```

## Syntax Elements

### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

### *array\_index*

Specifies one of the following:

- An integer value.
- An array instance of the predefined array type `ArrayVec` with a comma-separated list of integer values to define the bounds.
- The value of *array\_index* must be within the limits of *array\_expr*.

## Usage Notes

ARRAY\_GET takes an array expression as an argument and returns the element value in *array\_expr* that corresponds to the position specified by *array\_index*.

If *array\_expr* is a one-dimensional ARRAY type, the index of the element to be located is defined by an INTEGER or an `ArrayVec` type that must be within the defined boundaries of *array\_expr*.

If *array\_expr* is a multidimensional ARRAY type, the index of the element to be located is defined using the predefined array type `ArrayVec`. For more information on the `ArrayVec` type, see “ARRAY Scope Reference”. The number of dimensions defined by `ArrayVec` must be between 2 and 5 (the maximum number of dimensions supported) and must correspond to the number of dimensions in *array\_expr*. The values for each dimension are separated by a comma and they must be within the defined boundaries of *array\_expr*.

If the value of *array\_index* references an element of the ARRAY which is in an uninitialized state, an error is returned. To avoid referencing an element of the ARRAY that is uninitialized, set any uninitialized elements to NULL. You can do this with the `OEXTEND` method. See [OEXTEND](#).

If *array\_expr* is NULL, the function returns NULL.

## Examples

### Example: Query a 1-D ARRAY Data Type and Table using ARRAY\_GET

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS VARRAY(50) OF DECIMAL(7,2);
CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price item_price);
```

In the following query, ARRAY\_GET returns the element value located in position 40 of the regular\_price array.

```
SELECT ARRAY_GET(regular_price,40) FROM inventory;
```

### Example: Query a 2-D ARRAY Data Type and Table using ARRAY\_GET

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];
CREATE TABLE seismic_data (
    id INTEGER,
    shot1 shot_ary,
    shot2 shot_ary);
```

In the following query, ARRAY\_GET returns the element value located in position [5][10] of the shot1 array.

```
SELECT ARRAY_GET(shot1, NEW arrayVec(5,10)) FROM seismic_data;
```

The following is the same query using method-style syntax.

```
SELECT shot1.ARRAY_GET(NEW arrayVec(5,10)) FROM seismic_data;
```

## ARRAY\_COMPARE

Performs a pairwise comparison of the elements of two arrays and returns a value of 1, 0, or NULL to indicate whether or not the two array arguments contain the same element values.

## ANSI Compliance

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

The function does a pairwise comparison of the elements of two arrays, following the rules for ARRAY comparison according to the ANSI standard.

## ARRAY\_COMPARE Syntax

### System Function

```
ARRAY_COMPARE ( expr1, expr2 [, scope_reference, nulls_equal_flag ] )
```

### Method-Style

```
expr1.ARRAY_COMPARE ( expr2 [, scope_reference, nulls_equal_flag ] )
```

### Syntax Elements

#### *expr1/expr2*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

#### *scope\_reference*

An ARRAY scope reference.

#### *nulls\_equal\_flag*

A flag that specifies how NULL element values should be compared. Valid values for this parameter are 0 and 1.

## Usage Notes

The ARRAY\_COMPARE function does a pairwise comparison of the two array arguments, and returns a scalar value indicating whether or not the two array expressions contain all of the same element values.

ARRAY\_COMPARE follows the rules for ANSI array comparison as follows:

1. If either *expr1* or *expr2* evaluates to NULL, the result is NULL.
2. *expr1* and *expr2* are equal if and only if all the element values of *expr1* are equal to the element values of *expr2*.

3. An element value of *expr1* is equal to an element value of *expr2* if and only if they both have the same value and both have the same position in the array.
4. If any non-NULL elements do not match for a given element of Array A and Array B, then this is considered a mismatch and ARRAY\_COMPARE returns a value of 0.
5. If either or both values are NULL for a given element, and all other elements in the arrays match, this is considered as a mismatch according to the ANSI standard. The result in this case, according to the ANSI standard, is UNKNOWN. In Teradata, this evaluates to a NULL result.

If rule (1) or (5) above is satisfied, ARRAY\_COMPARE returns NULL. If rules (2) and (3) are satisfied, ARRAY\_COMPARE returns 1. Otherwise, ARRAY\_COMPARE returns 0.

When the optional *nulls\_equal\_flag* is set to 1, then when both values are NULL for a given element, this is considered as a match, which is a deviation from the ANSI standard. If all other elements match, then rule (2) above applies.

If you specify the optional argument *scope\_reference*, then the function is applied only to the elements within the limits of the given scope.

If an array argument contains any elements that are in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the ARRAY that contains uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See [OEXTEND](#).

## Restrictions

You cannot make relational comparisons of ARRAY values. You cannot use any of the relational comparison operators on ARRAY data. You can only make relational comparisons on individual elements of an ARRAY.

### Examples: Querying a Table using ARRAY\_COMPARE

In this example, the inventory table contains two ARRAY columns, *sale\_price\_june* and *sale\_price\_july*, which contain the following element values:

```
sale_price_june[1:10] = 50, 100, 200, 90, 250, 550, 200, 200, 50, 75
sale_price_july[1:10] = 50, 100, 200, 90, 250, 550, 200, 200, 50, 75
```

The output of the following query is 1 since all of the element values in the *sale\_price\_june* array are equal to the element values in the *sale\_price\_july* array.

```
SELECT ARRAY_COMPARE(sale_price_june, sale_price_july)
FROM inventory;
```

The following is the same query using method-style syntax.

```
SELECT sale_price_june.ARRAY_COMPARE(sale_price_july)
FROM inventory;
```

Consider the same arrays but with different element values as follows:

```
sale_price_june[1:10] = 50, 100, 200, 90, 250, 550, 200, 200, 50, 75
sale_price_july[1:10] = 50, 100, 200, 90, 300, 550, 200, 200, 50, 75
```

The output of the following query is 0 since the element value in position 5 differs in the two arrays.

```
SELECT ARRAY_COMPARE(sale_price_june, sale_price_july)
FROM inventory;
```

The following query compares the values of elements in positions 6 through 10. The output of this query is 1 since the element values in the given positions of both arrays are equal.

```
SELECT ARRAY_COMPARE(sale_price_june, sale_price_july, 6, 10)
FROM inventory;
```

## ARRAY\_UPDATE

Updates all or a subset of the elements in *array\_expr* to the specified new value.

ARRAY\_UPDATE returns a new copy of the array specified by *array\_expr* with the specified elements updated to the new value.

### ANSI Compliance

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

## ARRAY\_UPDATE Syntax

### System Function

```
ARRAY_UPDATE ( array_expr, new_value [, { scope_reference | array_index } ] )
```

### Method-Style

```
array_expr.ARRAY_UPDATE ( new_value [, { scope_reference | array_index } ] )
```

### Syntax Elements

***array\_expr***

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

***new\_value***

An expression that evaluates to a value with the same data type as the element type of *array\_expr*.

***scope\_reference***

An ARRAY scope reference.

***array\_index***

Specifies one of the following:

- An integer value.
- An array instance of the predefined array type ArrayVec with a comma-separated list of integer values to define the bounds.
- The value of *array\_index* must be within the limits of *array\_expr*.

## Usage Notes

ARRAY\_UPDATE takes an array expression as an argument and updates all or a subset of the elements in *array\_expr* with the value specified by *new\_value*.

The following rules apply to the ARRAY\_UPDATE function:

- If *array\_expr* is NULL, then the result value is NULL.
- The value of *new\_value* must evaluate to the same data type as the element type of *array\_expr*, or it must be a data type that can be implicitly converted to the ARRAY element type.
- If you do not specify *scope\_reference* or *array\_index*, then all the elements in *array\_expr* are updated to the value of *new\_value*.
- If you specify *scope\_reference*, then only the elements within the scope reference of *array\_expr* are updated to the value of *new\_value*. *scope\_reference* must refer to consecutive elements in the array.
- If you specify *array\_index*, then only the single element specified by *array\_index* is updated to the value of *new\_value* in *array\_expr*.
- If an element to be updated contains a NULL, the NULL is replaced with the value of *new\_value* in *array\_expr*.
- When one or more elements are modified, and there are uninitialized elements in the ARRAY value prior to the elements to be updated, then ARRAY\_UPDATE sets any preceding uninitialized elements to NULL. This behavior is the same as that of a regular SQL UPDATE statement that sets an ARRAY element value.



## Examples

### Example: Query a 1-D ARRAY Data Type and Table using ARRAY\_UPDATE

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS CHAR(10) ARRAY[20];
CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The following query returns an updated copy of the phonelist array, with all the elements updated to the new value:

```
SELECT ARRAY_UPDATE(phonelist, '9095551234')
FROM employee_info;
```

The following query returns an updated copy of the phonelist array, with a subset of the elements updated to the new value, as specified by the scope reference. The result is that elements 2, 3, and 4 are updated to the new value. The rest of the elements in the array retain their original values.

```
SELECT ARRAY_UPDATE(phonelist, '9095551234', 2, 4)
FROM employee_info;
```

The following is the same query using method-style syntax.

```
SELECT phonelist.ARRAY_UPDATE('9095551234', 2, 4)
FROM employee_info;
```

### Example: Query a 2-D ARRAY Data Type and Table using ARRAY\_UPDATE

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];
CREATE TABLE seismic_data (
  id INTEGER,
  shot1 shot_ary,
  shot2 shot_ary);
```

The following query returns an updated copy of the shot1 array, with all the elements updated to the new value.

```
SELECT ARRAY_UPDATE(shot1, 0)
FROM seismic_data;
```

The following query returns an updated copy of the shot1 array, with a subset of the elements updated to the new value, as specified by the scope reference. The result is that the elements in the scope reference range [5:10][5:10] are updated to the new value. The rest of the elements in the array retain their original values.

```
SELECT ARRAY_UPDATE(shot1, 0, NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```

## Related Information

For more information, see [ARRAY\\_Constructor\\_Expression](#).

## ARRAY\_UPDATE\_STRIDE

Updates all or a subset of the elements in *array\_expr* to the specified new value. The *stride* argument indicates how many elements should be skipped between each updated element.

ARRAY\_UPDATE\_STRIDE returns a new copy of the array specified by *array\_expr* with the specified elements updated to the new value.

### ANSI Compliance

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

## ARRAY\_UPDATE\_STRIDE Syntax

### System Function

```
ARRAY_UPDATE_STRIDE ( array_expr, new_value, stride [, { scope_reference |  
array_index } ] )
```

### Method-Style

```
array_expr.ARRAY_UPDATE_STRIDE ( new_value, stride [, { scope_reference |  
array_index } ] )
```

## Syntax Elements

### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

### *new\_value*

An expression that evaluates to a value with the same data type as the element type of *array\_expr*.

### *stride*

An unsigned integer value indicating the number of elements that should be skipped between each updated element.

### *scope\_reference*

An ARRAY scope reference.

### *array\_index*

Specifies one of the following:

- An integer value.
- An array instance of the predefined array type `ArrayVec` with a comma-separated list of integer values to define the bounds.
- The value of *array\_index* must be within the limits of *array\_expr*.

## Usage Notes

`ARRAY_UPDATE_STRIDE` takes an array expression as an argument and updates all or a subset of the elements in *array\_expr* with the value specified by *new\_value*. The *stride* argument indicates how many elements should be skipped between each updated element.

The following rules apply to the `ARRAY_UPDATE_STRIDE` function:

- If *array\_expr* is NULL, then the result value is NULL.
- The value of *new\_value* must evaluate to the same data type as the element type of *array\_expr*, or it must be a data type that can be implicitly converted to the ARRAY element type.
- If you do not specify *scope\_reference* or *array\_index*, then all the elements in *array\_expr* are updated to the value of *new\_value*.

- If you specify *scope\_reference*, then only the elements within the scope reference of *array\_expr* are updated to the value of *new\_value*. *scope\_reference* must refer to consecutive elements in the array.
- If you specify *array\_index*, then only the single element specified by *array\_index* is updated to the value of *new\_value* in *array\_expr*. In this case, *stride\_value* has no effect.
- The *stride\_value* argument specifies the number of elements to skip between each element to be updated. For example, if *stride\_value* is 1, then every other element within the affected range of elements is updated. The affected range of elements depends on whether or not *scope\_reference* is specified. For a multidimensional array, the elements are stored in row-major order so they are traversed and skipped according to *stride\_value* in this order.
- If an element to be updated contains a NULL, the NULL is replaced with the value of *new\_value* in *array\_expr*.
- When one or more elements are modified, and there are uninitialized elements in the ARRAY value prior to the elements to be updated, then ARRAY\_UPDATE\_STRIDE sets any preceding uninitialized elements to NULL. This behavior is the same as that of a regular SQL UPDATE statement that sets an ARRAY element value.
- If an element is skipped that was previously uninitialized, it is updated to NULL.

## Examples

### Example: Query a 1-D ARRAY Data Type and Table using ARRAY\_UPDATE\_STRIDE

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS CHAR(10) ARRAY[20];
CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The following query returns an updated copy of the phonelist array with every other element updated to the new value:

```
SELECT ARRAY_UPDATE_STRIDE(phonelist, '9095551234', 1)
FROM employee_info;
```

The following query returns an updated copy of the phonelist array, with a subset of the elements updated to the new value, as specified by the scope reference and stride value. The result is that elements 2 and 4 are updated to the new value. The rest of the elements in the array retain their original values.

```
SELECT ARRAY_UPDATE_STRIDE(phonelist, '9095551234', 1, 2, 4)
FROM employee_info;
```

## Example: Query a 2-D ARRAY Data Type and Table using ARRAY\_UPDATE\_STRIDE

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];
CREATE TABLE seismic_data (
  id INTEGER,
  shot1 shot_ary,
  shot2 shot_ary);
```

The following query returns an updated copy of the shot1 array, with every other element updated to the new value.

```
SELECT ARRAY_UPDATE_STRIDE(shot1, 0, 1)
FROM seismic_data;
```

The following query returns an updated copy of the shot1 array, with a subset of the elements updated to the new value, as specified by the scope reference and stride value. The result is that every 5th element beginning with the first element in the scope reference range [1:50][1:50] is updated to the new value. The rest of the elements in the array retain their original values.

```
SELECT ARRAY_UPDATE_STRIDE(shot1, 0, 5, NEW arrayVec(1,1),
  NEW arrayVec(50,50))
FROM seismic_data;
```

The following is the same query using method-style syntax.

```
SELECT shot1.ARRAY_UPDATE_STRIDE (0, 5, NEW arrayVec(1,1),
  NEW arrayVec(50,50))
FROM seismic_data;
```

## Related Information

For more information, see [ARRAY Constructor Expression](#).

## OEXISTS

Returns an integer value value of 1 or 0, indicating whether the element specified by the index in *array\_expr* contains a value or is in an uninitialized state.

## ANSI Compliance

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

## OEXISTS Syntax

### System Function

```
[TD_SYSFNLIB.] OEXISTS ( array_expr, { index value | array_bound } )
```

### Method-Style

```
array_expr.OEXISTS ( { index value | array_bound } )
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

#### *index value*

An unsigned integer value.

#### *array\_bound*

An array instance of the predefined array type *ArrayVec* with a comma-separated list of integer values to define the bounds.

## Usage Notes

OEXISTS takes an array expression as an argument and checks the element specified by the index to see if it contains a value or is in an uninitialized state. OEXISTS returns 1 if the specified element contains a non-NULL or NULL. If the element is in an uninitialized state, the method returns 0.

If the value of *index\_value* or *array\_bound* is out of bounds for *array\_expr*, OEXISTS returns 0. If either *index\_value* or *array\_bound* is NULL, a 0 value is returned. If *array\_expr* is NULL, then an error stating that the function does not exists is returned.

If *array\_expr* refers to an empty array with all elements in an uninitialized state, then OEXISTS returns 0. The OEXISTS method with an integer *index\_value* argument is compatible with the Oracle EXISTS method for one-dimensional ARRAY types.

# Examples

## Example: Query a 1-D ARRAY Data Type and Table using OEXISTS

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);
CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
    phonenumbers('1112223333', '6195551234'));
/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
    phonenumbers());
/* Update the empty ARRAY instance such that element 3 is set to a value; then
elements 1 and 2 are set to NULL, the rest are uninitialized. */
UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query checks to see whether element 2 of the phonelist array contains a value.

```
SELECT eno, phonelist.OEXISTS(2)
FROM employee_info;
```

The following is the result of the query.

ENO	phonelist.OEXISTS(2)	
---	-----	
1	1	(element 2 contains a value that is non-NULL)
2	1	(element 2 contains a NULL)

The following query checks to see whether element 3 of the phonelist array contains a value.

```
SELECT eno, phonelist.OEXISTS(3)
FROM employee_info;
```

The following is the result of the query.

ENO	phonelist.OEXISTS(3)
---	-----
1	0 (element 3 is in an uninitialized state)
2	1 (element 3 contains a value that is non-NULL)

The following is the same query using function-style syntax.

```
SELECT eno, OEXISTS(phonelist, 3)
FROM employee_info;
```

## Example: Query a 2-D ARRAY Data Type and Table using OEXISTS

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;
CREATE TABLE seismic_table (
  id INTEGER,
  shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));
/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());
/* Update the empty ARRAY instance such that element [1][3] is set to a value;
then elements [1][1] and [1][2] are set to NULL, the rest are uninitialized */
UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query checks to see whether element [1][2] of the shots array contains a value.

```
SELECT id, shots.OEXISTS(NEW arrayVec(1, 2))
FROM seismic_table;
```



The following is the result of the query.

ID	shots.OEXISTS(new arrayVec(1, 2))	
--	-----	
1	1	(element [1][2] contains a value that is non-NULL)
2	1	(element [1][2] contains a NULL)

The following query checks to see whether element [1][3] of the shots array contains a value.

```
SELECT id, shots.OEXISTS(NEW arrayVec(1, 3))
FROM seismic_table;
```

The following is the result of the query.

ID	shots.OEXISTS(NEW arrayVec(1, 3))	
--	-----	
1	0	(element [1][3] is in an uninitialized state)
2	1	(element [1][3] contains a value that is non-NULL)

## Related Information

For more information, see [ARRAY\\_Constructor\\_Expression](#).

## OCOUNT

Returns the number of elements in *array\_expr* that contain either a NULL or value that is non-NULL.

OCOUNT returns an unsigned integer value.

### ANSI Compliance

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

## OCOUNT Syntax

### System Function

```
[TD_SYSFNLIB.] OCOUNT ( array_expr )
```

### Method-Style

```
array_expr.OCOUNT ( )
```

## Syntax Elements

### TD\_SYSFNLIB.

Name of the database where the function is located.

### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

## Usage Notes

OCOUNT takes an array expression as an argument and returns the number of elements that contain a value that is non-NULL or NULL. Elements that are in an uninitialized state are not counted. If the array is empty (all elements of the array are in an uninitialized state), then OCOUNT returns a value of 0.

If *array\_expr* is NULL, an error is returned.

The OCOUNT method is compatible with the Oracle COUNT method for one-dimensional ARRAY types. However, the empty set of parentheses required by Teradata syntax is a deviation from Oracle syntax.

## Examples

### Example: Query a 1-D ARRAY Data Type and Table using OCOUNT

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);
CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
    phonenumbers('1112223333', '6195551234'));
/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
    phonenumbers());
/* Update the empty ARRAY instance such that element 3 is set to a value; then
```

```
elements 1 and 2 are set to NULL, the rest are uninitialized. */
UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query returns the number of elements in the phonelist array that contain NULL or values that are non-NULL.

```
SELECT eno, phonelist.OCOUNT()
FROM employee_info;
```

The following is the result of the query.

ENO	phonelist.OCOUNT()
---	-----
1	2 (the first 2 elements have values that are non-NULL)
2	3 (the first 3 elements have NULL or values that are non-NULL)

## Example: Query a 2-D ARRAY Data Type and Table using OCOUNT

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;
CREATE TABLE seismic_table (
  id INTEGER,
  shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));
/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());
/* Update the empty ARRAY instance such that element [1][3] is set to a value;
then elements [1][1] and [1][2] are set to NULL, the rest are uninitialized */
UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query returns the number of elements in the shots array that contain NULL or values that are non-NULL.

```
SELECT id, shots.OCOUNT()
FROM seismic_table;
```

The following is the result of the query.

ID	shots.OCOUNT()
--	-----
1	2 (the first 2 elements contain values that are non-NULL)
2	3 (the first 3 elements contain NULL or values that are non-NULL)

The following is the same query using function-style syntax.

```
SELECT id, OCOUNT(shots)
FROM seismic_table;
```

## Related Information

For more information, see [ARRAY\\_Constructor\\_Expression](#).

## OLIMIT

Returns the highest possible subscript value in *array\_expr* as either an unsigned INTEGER value or a new instance of the predefined ARRAY type ArrayVec.

OLIMIT returns an unsigned INTEGER value or a new instance of the predefined ARRAY type ArrayVec.

### ANSI Compliance

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

## OLIMIT Syntax

### System Function

### Method-Style

```
array_expr.OLIMIT ( )
```

### Syntax Elements

**TD\_SYSFNLIB.**

Name of the database where the function is located.

***array\_expr***

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

## Usage Notes

OLIMIT takes an array expression as an argument and returns the highest possible subscript value in the ARRAY type. If *array\_expr* is a one-dimensional ARRAY type, OLIMIT returns an unsigned INTEGER value. If *array\_expr* is a multidimensional ARRAY type, OLIMIT returns a new instance of the predefined ARRAY type ArrayVec, containing the subscript information.

If *array\_expr* is NULL, an error is returned.

The OLIMIT method is compatible with the Oracle LIMIT method for one-dimensional ARRAY types. However, the empty set of parentheses required by Teradata syntax is a deviation from Oracle syntax.

## Examples

### Example: Query a 1-D ARRAY Data Type and Table using OLIMIT

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);
CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
    phonenumbers('1112223333', '6195551234'));
/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
    phonenumbers());
```

The following query returns the highest possible subscript value in the phonelist array.

```
SELECT eno, phonelist.OLIMIT()
FROM employee_info;
```

The following is the result of the query.

ENO	phonelist.OLIMIT()
---	-----
1	20
2	20

The following is the same query using function-style syntax.

```
SELECT eno, OLIMIT(phonelist)
FROM employee_info;
```

## Example: Query a 2-D ARRAY Data Type and Table using OLIMIT

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;
CREATE TABLE seismic_table (
  id INTEGER,
  shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));
/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());
```

The following query returns the highest possible subscript value in the shots array.

```
SELECT id, shots.OLIMIT()
FROM seismic_table;
```

The following is the result of the query.

ID	shots.OLIMIT()
--	-----
1	NEW arrayVec(50,50)
2	NEW arrayVec(50,50)

## Related Information

For more information, see [ARRAY\\_Constructor\\_Expression](#).

## OFIRST

Returns the lowest subscript value in *array\_expr* as either an unsigned INTEGER value or a new instance of the predefined ARRAY type ArrayVec.

OFIRST returns an unsigned INTEGER value or a new instance of the predefined ARRAY type ArrayVec.

### ANSI Compliance

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

## OFIRST Syntax

### System Function

```
[TD_SYSFNLIB.] OFIRST ( array_expr )
```

### Method-Style

```
array_expr.OFIRST ( )
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

## Usage Notes

OFIRST takes an array expression as an argument and returns the lowest subscript value in the ARRAY type. If *array\_expr* is a one-dimensional ARRAY type, OFIRST returns an unsigned INTEGER value. If *array\_expr* is a multidimensional ARRAY type, OFIRST returns a new instance of the predefined ARRAY

type ArrayVec, containing the subscript information for the first element in the array. If the array is empty (all elements of the array are in an uninitialized state), then OFIRST returns NULL.

If *array\_expr* is NULL, an error is returned.

The OFIRST method is compatible with the Oracle FIRST method for one-dimensional ARRAY types. However, the empty set of parentheses required by Teradata syntax is a deviation from Oracle syntax.

## Examples

### Example: Query a 1-D ARRAY Data Type and Table using OFIRST

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);
CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
    phonenumbers('1112223333', '6195551234'));
/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
    phonenumbers());
```

The following query returns the lowest subscript value in the phonelist array.

```
SELECT eno, phonelist.OFIRST()
FROM employee_info;
```

The following is the result of the query.

ENO	phonelist.OFIRST()
---	-----
1	1
2	1

### Example: Query a 2-D ARRAY Data Type and Table using OFIRST

Consider the following 2-D ARRAY data type and table.



```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;
CREATE TABLE seismic_table (
  id INTEGER,
  shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));
/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());
```

The following query returns the lowest subscript value in the shots array.

```
SELECT id, shots.OFIRST()
FROM seismic_table;
```

The following is the result of the query.

ID	shots.OFIRST()
--	-----
1	NEW arrayVec(1,1)
2	NEW arrayVec(1,1)

The following is the same query using function-style syntax.

```
SELECT id, OFIRST(shots)
FROM seismic_table;
```

## Related Information

For more information, see [ARRAY\\_Constructor\\_Expression](#).

## OLAST

Returns the highest subscript value with a populated element in *array\_expr*.

OLAST returns an unsigned INTEGER value or a new instance of the predefined ARRAY type ArrayVec.

## ANSI Compliance

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

## OLAST Syntax

### System Function

```
[TD_SYSFNLIB.] OLAST ( array_expr )
```

### Method-Style

```
array_expr.OLAST ( )
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

## Usage Notes

OLAST takes an array expression as an argument and returns the highest subscript value with a populated element in the ARRAY type. This is different from the highest possible subscript value, as returned by the OLIMIT method. If *array\_expr* is a one-dimensional ARRAY type, OLAST returns an unsigned INTEGER value. If *array\_expr* is a multidimensional ARRAY type, OLAST returns a new instance of the predefined ARRAY type ArrayVec, containing the subscript information. If the array is empty (all elements of the array are in an uninitialized state), then OLAST returns NULL.

If *array\_expr* is NULL, an error is returned.

The OLAST method is compatible with the Oracle LAST method for one-dimensional ARRAY types. However, the empty set of parentheses required by Teradata syntax is a deviation from Oracle syntax.

## Examples

### Example: Query a 1-D ARRAY Data Type and Table using OLAST

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);
CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
    phonenumbers('1112223333', '6195551234'));
/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
    phonenumbers());
/* Update empty ARRAY instance such that element 3 is set to a value;
   Then elements 1 and 2 are set to NULL, the rest are uninitialized */
UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query returns the highest subscript value with a populated element in the phonelist array.

```
SELECT eno, phonelist.OLAST()
FROM employee_info;
```

The following is the result of the query.

ENO	phonelist.OLAST()
1	2
2	3

The following is the same query using function-style syntax.

```
SELECT eno, OLAST(phonelist)
FROM employee_info;
```

## Example: Query a 2-D ARRAY Data Type and Table using OLAST

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;
CREATE TABLE seismic_table (
```

```
id INTEGER,
shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));
/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());
/* Update empty ARRAY instance such that element [1][3] is set to a value; Then
elements [1][1] and [1][2] are set to NULL, the rest are uninitialized */
UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query returns the highest subscript value with a populated element in the shots array.

```
SELECT id, shots.OLAST()
FROM seismic_table;
```

The following is the result of the query.

ID	shots.OLAST()
--	-----
1	NEW arrayVec(1,2)
2	NEW arrayVec(1,3)

## Related Information

For more information, see [ARRAY\\_Constructor\\_Expression](#).

## OPRIOR

Returns the subscript of the element prior to the element specified for *array\_expr*, if that element is currently populated.

OPRIOR returns an unsigned INTEGER value or a new instance of the predefined ARRAY type ArrayVec.

## ANSI Compliance

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

## OPRIOR Syntax

### System Function

```
[TD_SYSFNLIB.] OPRIOR ( array_expr, { index_value | array_bound } )
```

### Method-Style

```
array_expr.OPRIOR ( { index_value | array_bound } )
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

#### *integer\_value*

An unsigned INTEGER value.

#### *array\_bound*

An array instance of the predefined array type *ArrayVec* with a comma-separated list of integer values to define the bounds.

## Usage Notes

OPRIOR takes an array expression as an argument and returns the subscript of the element prior to the element specified by *integer\_value* or *array\_bound*, if that element is currently populated. If you specify an element subscript that is less than or equal to the value of OFIRST for the array, a NULL is returned. Also, if *integer\_value* or *array\_bound* is NULL, an error is returned.

If *array\_expr* is a one-dimensional ARRAY type, OPRIOR returns an unsigned INTEGER value. If *array\_expr* is a multidimensional ARRAY type, OPRIOR returns a new instance of the predefined ARRAY type *ArrayVec*, containing the subscript information. If the array is empty (all elements of the array are in an uninitialized state), then OPRIOR returns NULL.

If *array\_expr* is NULL, an error is returned.

The OPRIOR method with one INTEGER argument is compatible with the Oracle PRIOR method for one-dimensional ARRAY types.

## Examples

### Example: Query a 1-D ARRAY Data Type and Table using OPRIOR

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);
CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
    phonenumbers('1112223333', '6195551234'));
/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
    phonenumbers());
/* Update empty ARRAY instance such that element 3 is set to a value;
   Then elements 1 and 2 are set to NULL, the rest are uninitialized */
UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query returns the subscript of the element prior to element 2 in the phonelist array.

```
SELECT eno, phonelist.OPRIOR(2)
FROM employee_info;
```

The following is the result of the query.

ENO	phonelist.OPRIOR(2)
---	-----
1	1
2	1

The following is the same query using function-style syntax.

```
SELECT eno, OPRIOR(phonelist, 2)
FROM employee_info;
```

## Example: Query a 2-D ARRAY Data Type and Table using OPRIOR

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;
CREATE TABLE seismic_table (
  id INTEGER,
  shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));
/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());
/* Update empty ARRAY instance such that element [1][3] is set to a value; Then
elements [1][1] and [1][2] are set to NULL, the rest are uninitialized */
UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query returns the subscript of the element prior to element [1][2] in the shots array.

```
SELECT id, shots.OPRIOR(NEW arrayVec(1, 2))
FROM seismic_table;
```

The following is the result of the query.

ID	shots.OPRIOR(NEW arrayVec(1, 2))
--	-----
1	NEW arrayVec(1,1)
2	NEW arrayVec(1,1)

## Related Information

For more information, see [ARRAY Constructor Expression](#).

## ONEXT

Returns the subscript of the element subsequent to the element specified for *array\_expr*, if that element is currently populated.

ONEXT returns an unsigned INTEGER value or a new instance of the predefined ARRAY type *ArrayVec*.

### ANSI Compliance

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

## ONEXT Syntax

### System Function

```
[TD_SYSFNLIB.] ONEXT ( array_expr, { index value | array_bound } )
```

### Method-Style

```
array_expr.ONEXT ( { index value | array_bound } )
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

#### *integer\_value*

An unsigned INTEGER value.

#### *array\_bound*

An array instance of the predefined array type *ArrayVec* with a comma-separated list of integer values to define the bounds.



## Usage Notes

ONEXT takes an array expression as an argument and returns the subscript of the element subsequent to the element specified by *integer\_value* or *array\_bound*, if that element is currently populated. If you specify an element subscript that is greater than or equal to the value of OLAST for the array, a NULL is returned. Also, if *integer\_value* or *array\_bound* is NULL, an error is returned.

If *array\_expr* is a one-dimensional ARRAY type, ONEXT returns an unsigned INTEGER value. If *array\_expr* is a multidimensional ARRAY type, ONEXT returns a new instance of the predefined ARRAY type ArrayVec, containing the subscript information. If the array is empty (all elements of the array are in an uninitialized state), then ONEXT returns NULL.

If *array\_expr* is NULL, an error is returned.

The ONEXT method with one INTEGER argument is compatible with the Oracle NEXT method for one-dimensional ARRAY types.

## Examples

### Example: Query a 1-D ARRAY Data Type and Table using ONEXT

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);
CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
    phonenumbers('1112223333', '6195551234'));
/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
    phonenumbers());
/* Update empty ARRAY instance such that element 3 is set to a value;
   Then elements 1 and 2 are set to NULL, the rest are uninitialized */
UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query returns the subscript of the element subsequent to element 2 in the phonelist array.

```
SELECT eno, phonelist.ONEXT(2)
FROM employee_info;
```

The following is the result of the query.

ENO	phonelist.ONEXT(2)
---	-----
1	? (the element following element 2 is unpopulated)
2	3

## Example: Query a 2-D ARRAY Data Type and Table using ONEXT

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;
CREATE TABLE seismic_table (
  id INTEGER,
  shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));
/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());
/* Update empty ARRAY instance such that element [1][3] is set to a value; Then
elements [1][1] and [1][2] are set to NULL, the rest are uninitialized */
UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query returns the subscript of the element subsequent to element [1][2] in the shots array.

```
SELECT id, shots.ONEXT(NEW arrayVec(1, 2))
FROM seismic_table;
```

The following is the result of the query.

ID	shots.ONEXT(NEW arrayVec(1, 2))
--	-----

```

1      ?      (The element following element [1][2] is unpopulated)
2      NEW arrayVec(1,3)

```

The following is the same query using function-style syntax.

```

SELECT id, ONEXT(shots, NEW arrayVec(1, 2))
FROM seismic_table;

```

## Related Information

- [ARRAY Constructor Expression](#).
- [ARRAY Scope Reference](#).

## OEXTEND

Allocates space for one or more new elements in *array\_expr*.

OEXTEND returns a new modified copy of the array argument.

### ANSI Compliance

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

## OEXTEND Syntax

### System Function

```

[TD_SYSFNLIB.] OEXTEND ( array_expr [, num_spaces [, { index_value |
array_bound } ] ] )

```

### Method-Style

```

array_expr.OEXTEND (
  [ num_spaces [, { index_value | array_bound } ] ]
)

```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column

- An ARRAY constructor expression
- A UDF expression
- A UDM expression

***num\_spaces***

An optional unsigned integer value representing the number of NULL elements to append to *array\_expr*.

***index\_value***

An optional unsigned integer value used as an index to an element in *array\_expr*.

***array\_bound***

An array instance of the predefined array type *ArrayVec* with a comma-separated list of integer values to define the bounds.

## Usage Notes

OEXTEND takes an array expression as an argument and allocates space for one or more new elements in the array.

IF you specify...	THEN the method...
OEXTEND()	appends a single NULL element to the ARRAY value. For a multidimensional (n-D) ARRAY, this is done in row-major order.
OEXTEND( <i>num_spaces</i> )	appends <i>n</i> NULL elements to the ARRAY value, where <i>n</i> = <i>num_spaces</i> . For an n-D ARRAY, this is done in row-major order.
OEXTEND( <i>num_spaces</i> , <i>index_value</i> ) or OEXTEND( <i>num_spaces</i> , <i>array_bound</i> )	appends <i>n</i> copies of the element indexed by either <i>index_value</i> or <i>array_bound</i> to the ARRAY value, where <i>n</i> = <i>num_spaces</i> . For an n-D ARRAY, this is done in row-major order.

The OEXTEND method with zero, one, or two arguments of INTEGER type is compatible with the Oracle EXTEND method, for one-dimensional ARRAY types. However, the empty set of parentheses required when OEXTEND is called with no arguments is a deviation from Oracle syntax.

OEXTEND returns an error in the following cases:

- If *array\_expr* is NULL.
- If either *index\_value* or *array\_bound* is NULL.
- If the number of elements appended by OEXTEND overflows the maximum size of the array.

If *num\_spaces* is NULL, no error occurs, but no action is taken by OEXTEND.

## Examples

### Example: Query a 1-D ARRAY Data Type and Table using OEXTEND

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);
CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
    phonenumbers('1112223333', '6195551234'));
/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
    phonenumbers());
/* Update empty ARRAY instance such that element 3 is set to a value;
   Then elements 1 and 2 are set to NULL, the rest are uninitialized */
UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query extends the phonelist array with a single NULL element.

```
SELECT eno, phonelist.OEXTEND()
FROM employee_info;
```

The following is the result of the query.

ENO	phonelist.OEXTEND()
---	-----
1	('1112223333', '6195551234', NULL)
2	(NULL, NULL, '8584850000', NULL)

The following query extends the phonelist array with three NULL elements.

```
SELECT eno, phonelist.OEXTEND(3)
FROM employee_info;
```

The following is the result of the query.

ENO	phonelist.OEXTEND(3)
---	-----
1	('1112223333', '6195551234', NULL, NULL, NULL)
2	(NULL, NULL, '8584850000', NULL, NULL, NULL)

The following query extends the phonelist array with two copies of element 1.

```
SELECT eno, phonelist.OEXTEND(2,1)
FROM employee_info
WHERE eno = 1;
```

The following is the result of the query.

ENO	phonelist.OEXTEND(2,1)
---	-----
1	('1112223333', '6195551234', '1112223333', '1112223333')

The following is the same query using function-style syntax.

```
SELECT eno, OEXTEND(phonelist,2,1)
FROM employee_info
WHERE eno = 1;
```

## Example: Using OEXTEND to Fill the End of a Constructed ARRAY with NULL Elements

The following example shows how you can use the OEXTEND method to fill the end of a constructed ARRAY with NULL elements so that these elements are no longer in an uninitialized state. This can be helpful when the ARRAY may be used as an argument to another system function or operator.

```
CREATE TYPE myarray AS VARRAY(5) OF INTEGER;
CREATE TABLE mytab (id INTEGER, ary myarray);
/* Populate the first 3 elements. The last 2 elements are uninitialized. */
INSERT INTO mytab VALUES (1, NEW myarray(1,2,3));
/* Fill the last 2 elements with NULLs. */
UPDATE mytab
SET ary = ary.OEXTEND(2)
WHERE id = 1;
SELECT ary FROM mytab;
```

The following is the result of the query.

```
(1,2,3,NULL,NULL)
```

## Example: Query a 2-D ARRAY Data Type and Table using OEXTEND

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;
CREATE TABLE seismic_table (
  id INTEGER,
  shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));
/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());
/* Update empty ARRAY instance such that element [1][3] is set to a value; Then
elements [1][1] and [1][2] are set to NULL, the rest are uninitialized */
UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query extends the shots array with a single NULL element.

```
SELECT id, shots.OEXTEND()
FROM seismic_table;
```

The following is the result of the query.

ID	shots.OEXTEND()
--	-----
1	(11,12,NULL)
2	(NULL,NULL,1133,NULL)

The following query extends the shots array with three NULL elements.

```
SELECT id, shots.OEXTEND(3)
FROM seismic_table;
```

The following is the result of the query.

ID	shots.OEXTEND(3)
--	-----
1	(11,12,NULL,NULL,NULL)
2	(NULL,NULL,1133,NULL,NULL,NULL)

The following query extends the shots array with two copies of element [1][1].

```
SELECT id, shots.OEXTEND(2, NEW arrayVec(1,1))
FROM seismic_table
WHERE id = 1;
```

The following is the result of the query.

ID	shots.OEXTEND(2, NEW arrayVec(1,1))
--	-----
1	(11,12,11,11)

## Related Information

- [ARRAY Scope Reference.](#)
- [ARRAY Constructor Expression.](#)

## OTRIM

Removes one or more populated elements from the end of *array\_expr*.

OTRIM returns a new modified copy of the array argument.

### ANSI Compliance

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

## OTRIM Syntax

### System Function

```
[TD_SYSFNLIB.] OTRIM ( array_expr [, integer_value ] )
```

### Method-Style

```
array_expr.OTRIM ( [ integer_value ] )
```



## Syntax Elements

### TD\_SYSFNLIB.

Name of the database where the function is located.

### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

### *integer\_value*

An optional unsigned INTEGER value indicating the number of populated elements to remove from the array. The default value is 1.

## Usage Notes

OTRIM takes an array expression as an argument and removes one or more populated elements from the end of the array value. If you do not specify *integer\_value*, then OTRIM removes a single populated element from the end of the array value; otherwise, OTRIM removes the number of populated elements specified by *integer\_value* from the end of the array value. For a multidimensional ARRAY type, this is done in row-major order. OTRIM returns a new modified copy of the array value as the result.

If *array\_expr* is NULL, an error is returned. If *integer\_value* is NULL, no error occurs, but no action is taken by OTRIM. If the number of elements to be removed by OTRIM is greater than the current value of OCOUNT for the array, then an error is returned.

The OTRIM method is compatible with the Oracle TRIM method for one-dimensional ARRAY types. If *integer\_value* is not specified, the empty set of parentheses required by Teradata syntax is a deviation from Oracle syntax.

## Examples

### Example: Query a 1-D ARRAY Data Type and Table using OTRIM

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);
CREATE TABLE employee_info (eno INTEGER, phonenumber phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
    phonenumbers('1112223333', '6195551234'));
/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
    phonenumbers());
/* Update empty ARRAY instance such that element 3 is set to a value;
   Then elements 1 and 2 are set to NULL, the rest are uninitialized */
UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query removes the last populated element from the phonelist array.

```
SELECT eno, phonelist.OTRIM()
FROM employee_info;
```

The following is the result of the query.

ENO	phonelist.OTRIM()
---	-----
1	('1112223333')
2	(NULL,NULL)

The following query removes the last two populated elements from the phonelist array.

```
SELECT eno, phonelist.OTRIM(2)
FROM employee_info;
```

The following is the result of the query.

ENO	phonelist.OTRIM(2)
---	-----
1	( ) (the only 2 populated elements were removed so we now have an empty array value)
2	(NULL)

**Example: Query a 2-D ARRAY Data Type and Table using OTRIM**

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;
CREATE TABLE seismic_table (
  id INTEGER,
  shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));
/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());
/* Update empty ARRAY instance such that element [1][3] is set to a value; Then
elements [1][1] and [1][2] are set to NULL, the rest are uninitialized */
UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query removes the last populated element from the shots array.

```
SELECT id, shots.OTRIM()
FROM seismic_table
WHERE id = 1;
```

The following is the result of the query.

ID	shots.OTRIM()
--	-----
1	(11)

The following query removes the last two populated elements from the shots array.

```
SELECT id, shots.OTRIM(2)
FROM seismic_table;
```

The following is the result of the query.

ID	shots.OTRIM(2)
--	-----
1	( ) (the only 2 populated elements were removed so we now have an empty array value)
2	(NULL)

The following is the same query using function-style syntax.

```
SELECT id, OTRIM(shots, 2)
FROM seismic_table;
```

## Related Information

For more information, see [ARRAY\\_Constructor\\_Expression](#).

## ODELETE

Removes all elements from *array\_expr*.

ODELETE returns a new modified copy of the array argument.

### ANSI Compliance

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

## ODELETE Syntax

### System Function

```
[TD_SYSFNLIB.] ODELETE ( array_expr )
```

### Method-Style

```
array_expr.ODELETE ( )
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *array\_expr*

An ARRAY expression, which is one of the following:

- A reference to an ARRAY column
- An ARRAY constructor expression
- A UDF expression
- A UDM expression

## Usage Notes

ODELETE takes an array expression as an argument and removes all populated elements from the array value. ODELETE returns a new modified copy of the array value as the result.

If *array\_expr* is NULL, an error is returned.

The ODELETE method is compatible with the Oracle DELETE method for one-dimensional ARRAY types. However, the empty set of parentheses required by Teradata syntax is a deviation from Oracle syntax.

## Examples

### Example: Query a 1-D ARRAY Data Type and Table using ODELETE

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);
CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
    phonenumbers('1112223333', '6195551234'));
/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
    phonenumbers());
/* Update empty ARRAY instance such that element 3 is set to a value;
   Then elements 1 and 2 are set to NULL, the rest are uninitialized */
UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query removes all populated elements from the phonelist array, leaving all of the elements in an uninitialized state.

```
SELECT eno, phonelist.ODELETE()
FROM employee_info;
```

The following is the result of the query.

```
ENO          phonelist.ODELETE()
---          -
```

```
1      ()
2      ()
```

The following is the same query using function-style syntax.

```
SELECT eno, ODELETE(phonelist)
FROM employee_info;
```

## Example: Query a 2-D ARRAY Data Type and Table using ODELETE

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;
CREATE TABLE seismic_table (
    id INTEGER,
    shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));
/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());
/* Update empty ARRAY instance such that element [1][3] is set to a value; Then
elements [1][1] and [1][2] are set to NULL, the rest are uninitialized */
UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query removes all populated elements from the shots array, leaving all elements of the array in an uninitialized state.

```
SELECT id, shots.ODELETE()
FROM seismic_table;
```

The following is the result of the query.

ID	shots.ODELETE()
--	-----
1	()
2	()

## Related Information

For more information, see [ARRAY\\_Constructor\\_Expression](#).

# UDT Data Type

This section describes the user-defined type (UDT) data type.

For information about the VARIANT\_TYPE UDT, see [VARIANT\\_TYPE Data Type](#).

The UDT data type represents a custom data type for modeling the structure and behavior of real-world entities used by applications.

## ANSI Compliance

UDTs are ANSI SQL:2011 compliant.

## Required Privileges

To create a table that has a UDT column, you must have the UDTUSAGE, UDTTYPE, or UDTMETHOD privilege on the SYSUDTLIB database, or have the UDTUSAGE privilege on the specified UDT.

To perform a query on a UDT column, you must have the UDTUSAGE privilege on the specified UDT.

## UDT Data Type Syntax

```
[SYSUDTLIB.] udt_name [ attribute [...] ]
```

### Syntax Elements

#### **SYSUDTLIB.**

The name of the database in which all UDTs are created.

#### ***udt\_name***

The name of a UDT that was created with a CREATE TYPE statement.

#### ***attribute***

Appropriate data type attributes.

A UDT column supports the following attributes:

- NULL
- NOT NULL
- FORMAT
- TITLE
- NAMED
- DEFAULT NULL

For details on using data type attributes with UDTs, see:



- [Default Value Control Phrases](#)
- [Data Type Formats and Format Phrases](#)

A UDT column does not support column storage or column constraint attributes.

## Usage Notes

### Types of UDTs

Vantage supports distinct and structured UDTs.

UDT Type	Description	Example
Distinct	A UDT that is based on a single predefined data type, such as INTEGER or VARCHAR.	A distinct UDT named euro that is based on a DECIMAL(8,2) data type can store monetary data.
Structured	A collection of one or more fields called attributes, where each attribute is defined as a predefined data type or other UDT (nesting is supported).	A structured UDT named circle can consist of x-coordinate, y-coordinate, and radius attributes.

UDTs can further be classified as LOB UDTs or non-LOB UDTs.

IF the UDT is ...	AND ...	THEN the UDT is a ...
distinct	the predefined data type on which it is based is a CLOB or BLOB	LOB UDT.
	the predefined data type on which it is based is not a LOB	non-LOB UDT.
structured	the data type of at least one attribute is CLOB, BLOB, or LOB UDT	LOB UDT.
	none of the attributes has a LOB or LOB UDT data type	non-LOB UDT.

You can create user-defined methods (UDMs) that operate on distinct and structured UDTs. For example, for a distinct UDT named euro, you can define a method that converts the value to a US dollar amount. Similarly, for a structured UDT named circle, you can define a method that computes the area of the circle using the radius attribute.

### External Representation

Every UDT has a corresponding predefined SQL data type that Vantage uses for import and export operations between client applications and the server. The predefined data type that maps to a particular UDT is specified in the transform definition associated with the UDT.

IF the UDT is ...	THEN ...
distinct	Vantage automatically generates a default transform definition for import and export operations that maps the UDT to the predefined data type on which it is based. For example, consider a distinct UDT named euro that is based on a DECIMAL(8,2) data type. The transform definition that Vantage automatically generates allows a client application to load data into a euro column using the external representation of DECIMAL(8,2). A client application can also perform queries on the euro column and receive values in the same format as DECIMAL(8,2).
structured	a transform definition must be created for the UDT using the CREATE TRANSFORM statement. The transform definition specifies a predefined SQL data type that acts as a container to import and export attribute values of the structured UDT. For example, consider a structured UDT named circle that consists of x-coordinate, y-coordinate, and radius attributes. Suppose the data type of each attribute is FLOAT. The transform definition could map a BYTE(24) predefined type to the circle UDT, allowing eight bytes for each FLOAT attribute. The client application can load data into a circle column using the external representation of BYTE(24). Similarly, the application can perform queries on the circle column and receive the attribute values in a BYTE(24) container. The logic for import operations that extracts the values from a predefined SQL data type and sets the corresponding UDT attributes is implemented as a UDF that is specified in the CREATE TRANSFORM statement. Similarly, the logic for export operations that packs the UDT attribute values into the predefined SQL data type is implemented as a UDF or UDM.

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

For more information about external representations for UDTs, see [External Representations for UDTs](#).

## Support for Multiple Transform Groups

You can create multiple transform groups for each UDT by using the CREATE TRANSFORM or REPLACE TRANSFORM statements. CREATE TRANSFORM also allows you to add transform groups for a UDT that already has existing transform groups. REPLACE TRANSFORM drops all existing transform groups for a UDT and creates new transform groups for the UDT. You can also use DROP TRANSFORM to delete transform groups from a UDT.

The maximum number of transform groups allowed for a particular UDT is 16.

### Note:

You cannot use CREATE TRANSFORM or REPLACE TRANSFORM to create new transforms for complex data types (CDTs). You can only create new transforms for structured and distinct user-defined types (UDTs).

When using the CREATE TRANSFORM statement, it is possible to only specify either a from-sql or to-sql function for a transform. However, in order to create a table with a UDT column, the default transform group for the UDT must contain both from-sql and to-sql functions in the transform group.

You can use the TRANSFORM option in the CREATE PROFILE/MODIFY PROFILE or CREATE USER/MODIFY USER statements to specify for a user the particular transform group that will be used for a given data type.

You can use the following macros to find the transform group for a UDT (or CDT), or the transform group settings for a user, profile, or current session.

Macro	Description
SYSUDTLIB.HelpCurrentUserTransforms	Lists the transform group settings of the current logon user.
SYSUDTLIB.HelpCurrentSessionTransforms	Lists the transform group settings of the current session.
SYSUDTLIB.HelpUserTransforms( <i>User</i> )	Lists the transform group settings for a specific user.
SYSUDTLIB.HelpCurrentUDTTransform( <i>UDT</i> )	Lists the transform group settings of the current session for the specified UDT.
SYSUDTLIB.HelpUDTTransform( <i>User</i> , <i>UDT</i> )	Lists the transform group for a UDT for a user.
SYSUDTLIB.HelpProfileTransforms( <i>Profile</i> )	Lists the transform group settings for a specific profile.
SYSUDTLIB.HelpProfileTransform( <i>Profile</i> , <i>UDT</i> )	Lists the transform group for a UDT for a profile.

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

## Restrictions

- You can declare a primary or secondary index on a UDT column when you create an indexed table, join index, or hash index. However, the following UDT types are not supported for any form of primary or secondary index.
  - LOB UDTs
  - VARIANT\_TYPE UDT
- Restrictions that apply to CLOB and BLOB data types also apply to LOB UDTs:
  - A table can have a maximum combination of 32 CLOB, BLOB, or LOB UDT columns.
  - Queue tables cannot have CLOB, BLOB, or LOB UDT columns.
- If you have existing UDTs with non-ASCII characters in the name, you cannot use the ASCII session character set to create new UDTs with LATIN characters in the name. You must use the UTF8 or UTF16 session character sets instead.

## Functions That Operate on UDTs

You can specify UDTs as parameters and return types for UDFs written in C, C++, or Java. This includes scalar and aggregate UDFs, table functions, and table operators.

You can specify UDTs as IN, INOUT, and OUT parameters of stored procedures and external stored procedures written in C, C++, or Java.

FNC functions and Java classes and methods are provided to enable a UDF or external stored procedure to access and set the value of a UDT parameter, or to get information about the UDT parameter.

For information about these functions and methods, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

You can use UDTs with most SQL functions and operators, with the exception of ordered analytical functions, provided that the following is true:

- An implicit cast definition exists that casts the UDT to a predefined type that is accepted by the function or operator.
- The DisableUDTImplCastForSysFuncOp DBS Control field is set to zero.

For more information on using UDTs with SQL functions and operators, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## UDT Data Type Examples

### Example: UDT Data Type

Consider the following statement that creates a distinct UDT named *euro*:

```
CREATE TYPE euro
AS DECIMAL(8,2)
FINAL;
```

The following statement creates a table that defines a *euro* column named *sales*:

```
CREATE TABLE european_sales
(region INTEGER
,sales euro);
```

### Example: Distinct UDT Parameter in a Java UDF

```
CREATE TYPE MONEY AS numeric(10,2) FINAL

REPLACE FUNCTION MyMoney (A1 MONEY)
RETURNS MONEY
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.mymoney';
```

```
public static java.math.BigDecimal mymoney(java.math.BigDecimal a)
                                     throws SQLException
```

Alternatively, you can define the function as follows:

```
REPLACE FUNCTION MyMoney (A1 MONEY)
RETURNS MONEY
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.mymoney(java.math.BigDecimal) returns
java.math.BigDecimal';

public static java.math.BigDecimal mymoney(java.math.BigDecimal a)
                                     throws SQLException
```

### Example: Structured UDT Parameter in a Java UDF

```
CREATE TYPE CIRCLE AS (x double, y double, r double)...

REPLACE FUNCTION MyCircle(A1 CIRCLE)
RETURNS INTEGER
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.mycircle';

public static int mycircle(java.sql.Struct s) throws SQLException
```

Alternatively, you can define the function as follows:

```
REPLACE FUNCTION MyCircle(A1 CIRCLE)
RETURNS INTEGER
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.mycircle(java.sql.Struct) returns
int';

public static int mycircle(java.sql.Struct s) throws SQLException
```

**Example: Distinct UDT Parameter in a Java External Stored Procedure**

```
CREATE TYPE MONEY AS numeric(10,2) FINAL

REPLACE PROCEDURE MyMoney(IN A1 MONEY, OUT A2 MONEY)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.mymoney';

public static void MyMoney(java.lang.BigDecimal A1, java.lang.BigDecimal[] A2)
throws SQLException
```

**Example: Structured UDT Parameter in a Java External Stored Procedure**

```
CREATE TYPE CIRCLE AS (x double, y double, r double)

REPLACE PROCEDURE MyCircle(IN A1 CIRCLE, OUT A2 INTEGER)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.mycircle';

public static void(java.sql.Struct A1, int[] A2) throws SQLException
```

**Related Information**

FOR information on ...	SEE ...
functions and operators that support UDTs	<i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145.
implementing the functionality for UDFs, UDMs, and external stored procedures that operate on UDTs	<i>Teradata Vantage™ - SQL External Routine Programming</i> , B035-1147.
creating UDT definitions	CREATE TYPE in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
creating and dropping transform definitions	<ul style="list-style-type: none"> <li>CREATE TRANSFORM and REPLACE TRANSFORM in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li> <li>DROP TRANSFORM in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li> </ul>

FOR information on ...	SEE ...
specifying nondefault transform group settings for a user	CREATE PROFILE, MODIFY PROFILE, CREATE USER, and MODIFY USER in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
creating cast definitions	CREATE CAST in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
creating tables with UDT columns	CREATE TABLE in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
DisableUDTImplCastForSysFuncOp DBS Control field	<i>Teradata Vantage™ - Database Utilities</i> , B035-1102.

# Parameter Data Types

This section describes data types that can be used only with input or result parameters in a function, method, stored procedure, or external stored procedure.

## TD\_ANYTYPE Data Type

An input or result parameter data type that is used in UDFs, UDMs, and external stored procedures, and that can accept any system-defined data type or user-defined type (UDT). The parameter attributes and return type are determined at execution time.

### ANSI Compliance

TD\_ANYTYPE is a Teradata extension to the ANSI SQL standard.

### Syntax

```
parameter_name TD_ANYTYPE
```

### Syntax Elements

#### *parameter\_name*

The name of an input or result parameter declared in a UDF, UDM, or external stored procedure.

## Usage Notes

### Where TD\_ANYTYPE is Useful

The TD\_ANYTYPE data type is useful in:

- Declaring routines (functions, methods, external stored procedures) that support parameters with differing precisions and character parameters with varying character sets.
- Avoiding precision loss of decimal parameters due to truncation when the arguments do not match with the parameters in the function signature.
- Reducing the number of overloaded routines needed to support routines that have constant parameter counts but differing parameter data types.
- Declaring routines whose return data type is determined at runtime.

### Rules

You can only specify TD\_ANYTYPE as a data type for:

- Input parameters in scalar, aggregate and table functions written in C, C++, or Java.



- Result parameters in scalar and aggregate functions written in C, C++, or Java.
- IN, INOUT, or OUT parameters in external stored procedures written in C, C++, or Java.
- Input and output parameters in UDMs written in C or C++.

## Restrictions

The TD\_ANYTYPE type is not supported as a parameter type for:

- Stored procedures
- SQL UDFs
- UDMs written in Java

TD\_ANYTYPE cannot be used:

- As the return type in table functions
- To represent UDT parameters in Java routines

## Declaring the Return Type of a TD\_ANYTYPE Result Parameter

When invoking a UDF or UDM that is defined with a TD\_ANYTYPE result parameter, you can use the RETURNS *data type* or RETURNS STYLE *column expression* clause to specify the desired return type. The column expression can be any valid table or view column reference, and the return data type is determined based on the type of the column.

For example:

```
SELECT (routine_name (parameter_list) RETURNS data_type);
SELECT (routine_name (parameter_list) RETURNS STYLE TableName.ColumnName);
```

The above queries invoke *routine\_name* using the specified parameters and declare the return type to be *data\_type* or the data type of the column, *TableName.ColumnName*.

---

### Note:

You must enclose the UDF or UDM invocation in parenthesis if you use the RETURNS or RETURNS STYLE clause.

---

When invoking an external stored procedure that is defined with a TD\_ANYTYPE OUT parameter, you can specify the RETURNS *data type* or RETURNS STYLE *column expression* clause along with the OUT parameter in the CALL statement to indicate the desired return type of the OUT parameter.

For example:

```
CALL procedure_name(parameter1, out_parameter RETURNS data_type);
CALL procedure_name(parameter1, out_parameter RETURNS
STYLE TableName.ColumnName);
```

The RETURNS or RETURNS STYLE clause is not mandatory as long as the routine also includes a TD\_ANYTYPE input parameter. If you do not specify a RETURNS or RETURNS STYLE clause, then the data type of the first TD\_ANYTYPE input argument is used to determine the return type of the TD\_ANYTYPE result or OUT parameter. For character types, if the character set is not specified as part of the data type, then the default character set is used.

Note that the RETURNS and RETURNS STYLE clauses are only used to set the return type for a TD\_ANYTYPE OUT parameter in external stored procedures. The data type of a TD\_ANYTYPE INOUT parameter is determined by the data type of the corresponding input argument.

## Example

Consider the following function definition:

```
CREATE FUNCTION ascii (string_expr TD_ANYTYPE)
  RETURNS TD_ANYTYPE
  LANGUAGE C
  NO SQL
  SPECIFIC ascii
  EXTERNAL NAME 'CS!ascii!UDFs/ascii.c'
  PARAMETER STYLE SQL;
```

This function returns the ASCII numeric value of the first character of the input string. It accepts input strings with the following data types:

- CHARACTER
- VARCHAR
- CLOB
- UDT with a CHARACTER, VARCHAR, or CLOB attribute.
- ARRAY with an element type of CHARACTER or VARCHAR.

The function accepts the character set associated with the input string. If no character set is explicitly specified, the default character set is used.

The return type supported by this function is BYTEINT, SMALLINT, or INTEGER. To specify the desired return type for the function, use the RETURNS *data type* or RETURNS STYLE *column expression* clause when invoking the function.

The following is a sample query that invokes this function:

```
SELECT (ascii('hello') RETURNS INTEGER);
```

The output returned by the query is:

```
ascii('hello')
-----
104
```

For more details about this function, including the corresponding C code example for the function, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## Related Information

FOR information on ...	SEE ...
declaring TD_ANYTYPE input and result parameters in UDFs	CREATE FUNCTION in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144, External Form and Table Form.
declaring TD_ANYTYPE input and output parameters in methods	CREATE METHOD in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
declaring TD_ANYTYPE IN, INOUT, or OUT parameters in external stored procedures	CREATE PROCEDURE (External Form) in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
writing routines that use TD_ANYTYPE input and result parameters	<i>Teradata Vantage™ - SQL External Routine Programming</i> , B035-1147.

## VARIANT\_TYPE Data Type

An input parameter data type that can be used to package and pass in a varying number of parameters of varying data types to a UDF as a single UDF input parameter.

Vantage supports dynamic UDTs, which are structured UDTs with a pre-assigned UDT type name of VARIANT\_TYPE, and whose structured type attribute composition is determined at run time.

The VARIANT\_TYPE UDT can be used only as a UDF input parameter data type. When you declare a UDF input parameter to be of VARIANT\_TYPE data type, you can use that parameter to pass in a varying number of parameters of varying types to the UDF.

The additional parameters are packaged and passed to the UDF as a single structured UDT. The number of parameters and the data types of the parameters are determined at runtime. Therefore, you can vary the parameter composition each time you invoke the routine.

To declare an instance of a VARIANT\_TYPE UDT and define the runtime composition of the UDT, use the NEW VARIANT\_TYPE expression. For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145. For information on declaring a UDF input parameter to be a VARIANT\_TYPE data type, see CREATE FUNCTION and REPLACE FUNCTION in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## ANSI Compliance

VARIANT\_TYPE is a Teradata extension to the ANSI SQL standard.

## Syntax

```
parameter_name VARIANT_TYPE
```

## Syntax Elements

*parameter\_name*

The name of an input parameter declared in a UDF.

## Usage Notes

### VARIANT\_TYPE UDT High-Level Features

UDT Classification	Structured Type UDT
UDT Type Name	Variant_Type
Associated Methods (UDMs)	None
Ordering Functionality	<p>Implements full map ordering. The ordering routine will return the value of the first attribute of the VARIANT_TYPE UDT.</p> <p><b>Note:</b> The first attribute of the VARIANT_TYPE UDT cannot be one of the following: LOB, UDT, or LOB-UDT.</p>
Transform Functionality	<p>None</p> <p>You cannot import or export the VARIANT_TYPE UDT. You cannot use this data type in SQL statements that result with the data type being returned to the caller as output. Otherwise, you will receive an SQL error stating that no such UDT is defined.</p>
Cast Functionality	None

### Maximum Number of Supported Parameters

UDFs support a maximum of 128 parameters. In addition, each VARIANT\_TYPE input parameter accommodates up to 128 parameters, and you can declare up to 8 UDF input parameters to be of VARIANT\_TYPE data type. This increases the number of supported UDF input parameters from 128 to  $120 + (8 \times 128) = 1144$  input parameters per UDF.

## Restrictions

- You can specify the VARIANT\_TYPE UDT as a data type of input parameters to UDFs. You cannot use them as the data type for UDF result parameters or for any other purpose.
- You can use the VARIANT\_TYPE data type only with UDFs written in C or C++. You cannot use them with SQL UDFs or UDFs written in Java.
- All restrictions that apply to structured UDTs also apply to the VARIANT\_TYPE UDT.

## Example

This example shows a user-defined aggregate function with an input parameter named *parameter\_1* declared as a VARIANT\_TYPE data type.

```
CREATE TYPE INTEGERUDT AS INTEGER FINAL;
CREATE FUNCTION udf_agch002002dynudt (parameter_1 VARIANT_TYPE)
RETURNS INTEGERUDT CLASS AGGREGATE (4)
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!udf_agch002002dynudt!udf_agch002002dynudt.c'
PARAMETER STYLE SQL;
```

## Related Information

FOR information on ...	SEE ...
declaring VARIANT_TYPE input parameters in UDFs	CREATE FUNCTION and REPLACE FUNCTION in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
declaring an instance of a VARIANT_TYPE UDT and defining the runtime composition of the UDT	NEW VARIANT_TYPE in <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145.
writing UDFs that use VARIANT_TYPE input parameters	<i>Teradata Vantage™ - SQL External Routine Programming</i> , B035-1147.

# Data Type Formats and Format Phrases

This section describes the DATE, TIME and TIMESTAMP formats. It also describes the default formats and the use of format phrases to control how headings and data appear on output, whether to an online user or to a printed report. Format phrases can also be used for converting or casting from one data type to another.

## Data Type Default Formats

Teradata SQL uses a set of default formats for displaying expressions and column data, and for conversions between data types.

For example, Vantage assigns a format to every column of a table at the time the table is created. If the CREATE TABLE statement omits the format for a column, then Vantage assigns a default format for the data type of the column.

You can override the default format by using the FORMAT phrase in conjunction with a CREATE TABLE, ALTER TABLE, SELECT, UPDATE, DELETE, MERGE, or INSERT statement. For more information, see [FORMAT](#).

You can change the default formats that Teradata SQL uses to display numeric, date, and time data types. For details, see [Changing the Default Formats](#).

You can also use the FORMAT phrase with CAST or with Teradata conversion syntax to convert data from one type to another. For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Default Formats

The following table lists the default formats that Teradata SQL uses for each data type. For explanations of the formatting characters, see:

- [FORMAT Phrase and Character Formats](#)
- [FORMAT Phrase and NUMERIC Formats](#)
- [FORMAT Phrase and DateTime Formats](#)

Data Type	Default Format
BIGINT	-(19)9
BYTE[(n)] VARBYTE(n) BLOB[(n)]	<ul style="list-style-type: none"> <li>• If <math>n</math> is <math>\leq 32000</math>, the default format is X(2n).</li> <li>• If <math>n</math> is <math>&gt; 32000</math>, the default format is X(64000).</li> </ul>
BYTEINT	-(3)9
CHARACTER[(n)]	X(n)

Data Type	Default Format
VARCHAR( <i>n</i> )	X( <i>n</i> )
LONG VARCHAR	<ul style="list-style-type: none"> <li>If the server character set is UNICODE, GRAPHIC, or KANJISJIS, the default format is X(32000).</li> <li>If the server character set is LATIN or KANJI1, the default format is X(64000).</li> </ul>
CLOB[( <i>n</i> )]	<ul style="list-style-type: none"> <li>If the server character set is LATIN and <i>n</i> is &lt; 64000, the default format is X(<i>n</i>).</li> <li>If the server character set is LATIN and <i>n</i> is &gt;= 64000, the default format is X(64000).</li> <li>If the server character set is UNICODE and <i>n</i> is &lt; 32000, the default format is X(<i>n</i>).</li> <li>If the server character set is UNICODE and <i>n</i> is &gt;= 32000, the default format is X(32000).</li> </ul>
DATE	In ANSIDATE mode: YYYY-MM-DD. In INTEGERDATE mode: YY/MM/DD.
DECIMAL[( <i>n</i> [, <i>m</i> ])] NUMERIC[( <i>n</i> [, <i>m</i> ])]	--( <i>l</i> ).9( <i>F</i> ), where <i>l</i> = <i>n</i> - <i>m</i> and <i>F</i> = <i>m</i> .
FLOAT REAL DOUBLE PRECISION	-9.999999999999999E-999
INTEGER	-(10)9
INTERVAL DAY	-d( <i>n</i> ), where <i>n</i> is the value of the optional defined precision.
INTERVAL DAY TO HOUR	-d( <i>n</i> ) hh, where <i>n</i> is the value of the optional defined precision.
INTERVAL DAY TO MINUTE	-d( <i>n</i> ) hh:mm, where <i>n</i> is the value of the optional defined precision.
INTERVAL DAY TO SECOND	<p>-d(<i>n</i>) hh:mm:ss where <i>n</i> is value of the optional defined precision and there is no defined fractional precision.</p> <p>-d(<i>n</i>) hh:mm:ss.s(<i>m</i>) for fractional precision, where <i>n</i> is the value of the optional defined precision and <i>m</i> is the value of the optional defined fractional precision.</p>
INTERVAL HOUR	-h( <i>n</i> ), where <i>n</i> is the value of the optional defined precision.
INTERVAL HOUR TO MINUTE	-h( <i>n</i> ):mm, where <i>n</i> is the value of the optional defined precision.
INTERVAL HOUR TO SECOND	<p>-h(<i>n</i>):mm:ss, where <i>n</i> is the value of the optional defined precision and there is no defined fractional precision.</p> <p>-h(<i>n</i>):mm:ss.s(<i>m</i>) for fractional precision, where <i>n</i> is the value of the optional defined precision and <i>m</i> is the value of the optional defined fractional precision.</p>
INTERVAL MINUTE	-m( <i>n</i> ), where <i>n</i> is the value of the optional defined precision.

Data Type	Default Format
INTERVAL MINUTE TO SECOND	-m(n):ss, where <i>n</i> is the value of the optional defined precision and there is no defined fractional precision. -m(n):ss.s(m) for fractional precision, where <i>n</i> is the value of the optional defined precision and <i>m</i> is the value of the optional defined fractional precision.
INTERVAL MONTH	-m(n), where <i>n</i> is the value of the optional defined precision.
INTERVAL SECOND	-s(n), where <i>n</i> is the value of the optional defined precision and there is no defined fractional precision. -s(n).s(m) for fractional precision, where <i>n</i> is the value of the optional defined precision and <i>m</i> is the value of the optional defined fractional precision.
INTERVAL YEAR	-y(n), where <i>n</i> is the value of the optional defined precision.
INTERVAL YEAR TO MONTH	-y(n)-mm, where <i>n</i> is the value of the optional defined precision.
NUMBER	FN9, for both fixed point and floating point NUMBER.
PERIOD(DATE)	In ANSIDATE mode: ('YYYY-MM-DD', 'YYYY-MM-DD'). In INTEGERDATE mode: ('YY/MM/DD', 'YY/MM/DD').
PERIOD(TIME)	('HH:MI:SS.S(F)Z', 'HH:MI:SS.S(F)Z') Time zone information is displayed for PERIOD(TIME[(n)] WITH TIME ZONE), but not for PERIOD(TIME[(n)]).
PERIOD(TIMESTAMP)	('YYYY-MM-DDBHH:MI:SS.S(F)Z', 'YYYY-MM-DDBHH:MI:SS.S(F)Z') Time zone information is displayed for PERIOD(TIMESTAMP[(n)] WITH TIME ZONE), but not for PERIOD(TIMESTAMP[(n)]).
SMALLINT	-(5)9
TIME TIME WITH TIME ZONE	HH:MI:SS.S(F)Z Time zone information is displayed for TIME WITH TIME ZONE, but not for TIME.
TIMESTAMP TIMESTAMP WITH TIME ZONE	YYYY-MM-DDBHH:MI:SS.S(F)Z Time zone information is displayed for TIMESTAMP WITH TIME ZONE, but not for TIMESTAMP.
UDT	Default format of the external type of the UDT, as specified by the transform that defines how to pass the UDT between the application and the server. For more information on transforms, see CREATE TRANSFORM in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.

## Changing the Default Formats

You can change the default formats that Teradata SQL uses for the following data types:



<ul style="list-style-type: none"> <li>• BIGINT</li> <li>• BYTEINT</li> <li>• DATE</li> <li>• DECIMAL/NUMERIC</li> <li>• FLOAT/REAL/DOUBLE PRECISION</li> </ul>	<ul style="list-style-type: none"> <li>• INTEGER</li> <li>• NUMBER</li> <li>• SMALLINT</li> <li>• TIME and TIME WITH TIME ZONE</li> <li>• TIMESTAMP and TIMESTAMP WITH TIME ZONE</li> </ul>
---	---

To change the data type default formats, you must create a specification for data formatting (SDF) file, and then use the `tdlocaledef` utility. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Vantage uses the changes when it assigns formats to columns of new tables where the `CREATE TABLE` statement does not explicitly specify the column format. Columns of tables that were created prior to changing the default formats continue to use the format that Vantage assigned to the column at the time the table was created.

To view the settings of the preceding data type default formats for the current session, use the `HELP SESSION` statement. For more information, see `HELP` and `SHOW` in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## DATE Formats

To simplify your work with dates, Vantage assumes that you want dates in a preset format. Your database administrator sets that format for the system with the `DateForm` setting and the default date format setting in the specification for data formatting (SDF) file.

Those settings determine the following:

- Export data type of DATE values
- Data entry format for DATE values
- Display format of DATE values in macros being executed
- Date format for string-to-DATE comparisons and conversions
- Display format of DATE columns in newly created tables and views

## Changing or Overriding the DATE Format Settings

You can change or override the DATE format settings in any of the following ways:

- Change the `DateForm` at the system level
- Set the `DateForm` at the user or session level
- Change the system default date format in the SDF
- Specify a format in a `FORMAT` phrase

Each of these methods involves consequences that are explained in the following topics and summarized in [Hierarchy of Date Formats](#).

## The DateForm Setting

Before you change the DateForm setting, you need to consider the effect the change has on the following:

- Export data type of DATE values.  
Changing the DateForm setting changes the export data type of DATE values.
- Data-entry format for dates.  
Changing the DateForm setting changes the data entry format of dates for new tables, but not for existing tables. To avoid data entry problems, however, you can enter all dates as ANSI DATE literals.
- Display format of DATE columns.  
Changing the DateForm setting changes the display of DATE values in new tables, but not in existing tables. Therefore, DATE values display differently for new tables than for old tables. Because views are based on the underlying table, this is true for views of tables as well.
- String-to-DATE comparisons and conversions.  
To compare or convert strings to DATE values, the strings must have the same format as the DATE values. Such comparisons can fail after you change the DateForm setting because the DATE format for new tables, views of new tables, and existing macros changes. To avoid format errors in string-to-DATE comparisons, however, you can specify dates as ANSI DATE literals.
- Format of DATE values in macros being executed.  
Changing the DateForm setting can change the format of DATE values in macros, which can cause the macro to fail.

Changing the DateForm setting does not change the DATE format for the following:

- Tables created prior to the format change
- Tables created by users currently logged on
- Macros executed by users currently logged on
- Views based on tables that were created prior to the change

## System Default DATE Format

When the value of DateForm is ANSIDate, the system default DATE format is YYYY-MM-DD. When the value of DateForm is IntegerDate, the system default DATE format is YY/MM/DD.

Your system administrator can change the default format that applies to DATE data types when the DateForm is set to IntegerDate. Changing the default DATE format involves changing the value of the DATE element in a custom Specification for Data Formatting (SDF) file, and using the `tdlocaledef` utility to convert the information into an internal form usable by Vantage.

Setting or changing the default DATE format in the SDF has the same consequences as those for DateForm, except that the default DATE format in the SDF does not change the export data type of DATE values.

To view the value of the system default DATE format, use `HELP SESSION`.

## Using the FORMAT Phrase to Set DATE Formats

You can use the `FORMAT` phrase to set the format for a specific DATE column or DATE value. A `FORMAT` phrase overrides the DATE format of the system, user, and session.

Setting a different DATE format for a column with the `FORMAT` phrase has the same consequences as with any other method:

- Data entry for that column must be in the new format or the ANSI DATE literal format.
- To compare or convert strings to DATE values in the column, the strings must match that format.

A `FORMAT` phrase does not change the export data type of DATE values.

For details about using the `FORMAT` phrase, see [FORMAT Phrase and DateTime Formats](#) or [FORMAT Phrase, DateTime Formats, and Japanese Character Sets](#).

## Valid DATE Formats

The following table describes the valid DATE formats. The components that make up a valid date format are the same components that make up a DATE string validation. See [DATE Data Type](#).

Using a format with a four-digit year, such as `YYYY-MM-DD`, is recommended to avoid the Year 2000 problem. To work with two-digit year formats in strings, see the Century Break feature in *Teradata Vantage™ - Database Utilities*, B035-1102.

	ANSIDate	IntegerDate	User-Defined
Field Mode Display Format	YYYY-MM-DD	YY/MM/DD, or the value of the DATE element in a custom SDF	As defined by format
Allowed Input Separators	Any non-numeric character	Any non-numeric character	Any non-numeric character
Export Data Type	Character	Numeric	Depends on DateForm setting
Description	ANSI-specified date format.	Teradata legacy date format.	Custom date format for the system or a DATE column or value.
Example	1999-11-30	99/11/30	Jan 31, 2000

## Hierarchy of Date Formats

The operations that set the DATE format are best described as a hierarchy, where one setting overrides another.

At the lowest level, the system comes with a DateForm of IntegerDate ('YY/MM/DD'). Your system administrator can override that date format by changing the DateForm to ANSIDate ('YYYY-MM-DD') or creating a custom SDF that changes the default DATE format (any valid date format).

You can override the system-level date format for a user, session, individual column, or individual value.

The following table summarizes the hierarchy of date formats.

For this operation ...	The format is set for ...	And using this session date setting ...	And field mode display format ...	The export data type is ...
SELECT with FORMAT phrase	an individual column	user-defined display format	as defined by FORMAT	CHAR(10) if DateForm is ANSIDate; ELSE four-byte integer
CREATE / ALTER TABLE, with FORMAT phrase				
SET SESSION DateForm	the session	ANSIDate	YYYY-MM-DD	CHAR(10)
CLI SessionOptions parcel DateForm		IntegerDate	YY/MM/DD, or the value of the DATE element in a custom SDF	four-byte integer
CREATE / MODIFY USER DateForm	the user	ANSIDate	YYYY-MM-DD	CHAR(10)
		IntegerDate	YY/MM/DD, or the value of the DATE element in a custom SDF	four-byte integer
System DBS Control utility DateForm	the system	ANSIDate	YYYY-MM-DD	CHAR(10)
		IntegerDate	YY/MM/DD, or the value of the DATE element in a custom SDF	four-byte integer
Default DATE format in custom SDF	the system	IntegerDate	as defined by the DATE format in the SDF	four-byte integer

## TIME and TIMESTAMP Formats

Teradata SQL uses a default set of formats for the output of TIME and TIMESTAMP expressions and column data, and for comparison and conversion of TIME and TIMESTAMP data types.

## Changing or Overriding the TIME and TIMESTAMP Format Settings

You can change or override the TIME and TIMESTAMP format settings for field mode in the following ways:

- Change the system default TIME and TIMESTAMP formats in the specification for data formatting (SDF) file.
- Specify a format in a FORMAT phrase.

The FORMAT phrase sets the format for a specific TIME or TIMESTAMP column or value. A FORMAT phrase overrides the system format.

The TIME and TIMESTAMP format setting pertains to data in report form, as is the case in BTEQ. The format does not control internal storage representation of data or data returned in record or indicator variable mode.

For details about the FORMAT phrase, see [FORMAT Phrase and DateTime Formats](#) or [FORMAT Phrase, DateTime Formats, and Japanese Character Sets](#).

## System Default TIME and TIMESTAMP Format

The following table provides the system default formats.

Data Type	System Default Format
TIME TIME WITH TIME ZONE	HH:MI:SS.S(F)Z Time zone information is displayed for TIME WITH TIME ZONE, but not for TIME.
TIMESTAMP TIMESTAMP WITH TIME ZONE	YYYY-MM-DDBHH:MI:SS.S(F)Z Time zone information is displayed for TIMESTAMP WITH TIME ZONE, but not for TIMESTAMP.

Your system administrator can change the default format for TIME and TIMESTAMP data types. Changing the default TIME and TIMESTAMP formats involves changing the values of the TIME and TIMESTAMP elements in the Specification for Data Formatting (SDF) file, and using the `tdlocaledef` utility to convert the information into an internal form usable by Vantage.

Before changing the default format, consider the effect on the following.

Data-entry format	Changing the default formats changes the data entry format of TIME and TIMESTAMP data for new tables, but not for existing tables. To avoid data entry problems, however, you can enter all times and timestamps as ANSI TIME and TIMESTAMP literals.
Display format	Changing the default format changes the display of TIME and TIMESTAMP values in new tables, but not in existing tables, so values for new tables and old tables display differently. Because views are based on the underlying table, this is true for views of tables as well.

Character-to-TIME and character-to-TIMESTAMP comparisons and conversions	To compare or convert strings to TIME or TIMESTAMP values, the strings must have the same format as the TIME or TIMESTAMP values. Such comparisons can fail after you change the default format, because the format changes for new tables, views of new tables, and existing macros. To avoid format errors in character-to-TIME and character-to-TIMESTAMP comparisons, however, you can specify times and timestamps as ANSI TIME and TIMESTAMP literals.
Format of TIME and TIMESTAMP values in macros being executed	Changing the default format can change the format of TIME and TIMESTAMP values in macros, which can cause the macro to fail.

Changing the default format does not change the TIME or TIMESTAMP format for the following:

- Tables created prior to the format change
- Tables created by users currently logged on
- Macros executed by users currently logged on
- Views based on tables that were created prior to the change

Use `HELP SESSION` to view the values of the system default TIME and TIMESTAMP formats.

## Using the FORMAT Phrase to Set TIME or TIMESTAMP Formats

You can use the `FORMAT` phrase to set the format for a specific TIME or TIMESTAMP column or value. A `FORMAT` phrase overrides the TIME or TIMESTAMP system format.

Setting a different TIME or TIMESTAMP format for a column with the `FORMAT` phrase has the same consequences as with any other method:

- Data entry for that column must be in the new format or the ANSI TIME or TIMESTAMP literal format.
- To compare or convert strings to TIME or TIMESTAMP values in the column, strings must match that format.

A `FORMAT` phrase does *not* change the export data type of TIME or TIMESTAMP values.

For details about using the `FORMAT` phrase, see [FORMAT Phrase and DateTime Formats](#) or [FORMAT Phrase, DateTime Formats, and Japanese Character Sets](#).

## TIME and TIMESTAMP Format Components

For details on the components that make up a TIME and TIMESTAMP format, see [Formatting Characters for Time Information](#).

## Hierarchy of TIME and TIMESTAMP Formats

The operations that set the TIME and TIMESTAMP formats are best described as a hierarchy, where one setting overrides another.

At the lowest level, the system comes with a default TIME and TIMESTAMP format. Your system administrator can override the formats by creating a custom SDF that changes the default TIME and TIMESTAMP formats.

You can override the system-level TIME or TIMESTAMP format for an individual column or individual value.

The following table summarizes the hierarchy of TIME and TIMESTAMP formats.

For this operation ...	The format is set for ...	And field mode display format ...
SELECT with FORMAT phrase	an individual column	as defined by FORMAT.
CREATE / ALTER TABLE, with FORMAT phrase		
Default TIME or TIMESTAMP format in custom SDF	the system	as defined by the TIME and TIMESTAMP format in the SDF.

## Example: TIME Output Format in Field Mode

The following table is a nonexhaustive list of formats that can be used to present an output time in field mode, where the data is 13:20:53.64+03:00.

FORMAT Phrase	Result
FORMAT 'HH:MIBT'	01:20 PM
FORMAT 'HH:MI'	13:20
FORMAT 'HH.MI.SS'	13.20.53
FORMAT 'HH:MI:SSBT'	01:20:53 nachm (nachm is German for PM.)
FORMAT 'HH:MI:SSDS(F)'	13:20:53.64
FORMAT 'HH:MI:SSDS(F)Z'	13:20:53.64+03:00
FORMAT 'HHhMIImSSs'	13h20m53s

## Example: TIMESTAMP Output Format in Field Mode

The following table is a nonexhaustive list of formats for presenting an output timestamp in field mode, where the data is 85/09/12 13:20:53.64+03:00.

FORMAT Phrase	Result
FORMAT 'MM/DD/YYBHH:MIBT'	85/09/12 01:20 PM
FORMAT 'MMMBDD,BYYBHH:MI:SS'	Sep 12, 85 13:20:53

FORMAT Phrase	Result
FORMAT 'E3,BM4BDD,BY4BHH:MI:SSDS(F)'	Thu, September 12, 1985 13:20:53.64
FORMAT 'YYYY-MM-DDBHH:MI:SSDS(F)Z'	1985-09-12 13:20:53.64+03:00

## FORMAT

The FORMAT phrase of a CREATE TABLE, ALTER TABLE, SELECT, UPDATE, DELETE, MERGE, or INSERT statement overrides the default format.

### ANSI Compliance

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

## Syntax

```
FORMAT 'format_string'
```

### Syntax Elements

#### 'format\_string'

A variable length string enclosed in apostrophes containing formatting characters that define the display format for the data type.

## Usage Notes

Formats can be specified for columns that have character, numeric, byte, DateTime, Period, or UDT data types.

FORMAT pertains to data exported in report form, as is the case in BTEQ. FORMAT does *not* control internal storage representation of data or data returned in record or indicator variable mode.

Use the FORMAT phrase in a CREATE TABLE statement or ALTER TABLE statement to define the display format for a column. You can also use it in a retrieval statement to override the default format of a column or to define the display format of an expression. As such, it is both a Data Definition Language phrase and a Data Manipulation Language phrase.

A FORMAT specification can contain a maximum of 30 characters.

A FORMAT phrase can describe up to 18 digit positions (17 if the FORMAT contains an E). IEEE 64-bit floating numbers are accurate to about 15 digits.

The output string that is produced as a result of a FORMAT phrase can have a maximum of 255 characters.



If you specify a FORMAT phrase to define the format for UDT columns, the format must be valid for the external type of the UDT, as specified by the transform that defines how to pass the UDT between the application and the server.

## Relationship to SDF

In addition to defining the default display formats for numeric, date, time, and timestamp data types, the Specification for Data Formatting (SDF) file defines strings that Vantage displays in place of certain formatting characters that appear in a FORMAT phrase.

For example, you can set the value of the *Currency* string in the SDF file to the currency symbol native to your locale. To include that currency symbol when you display numeric monetary information in an SQL SELECT statement, use the L formatting character in the FORMAT phrase.

For more information on how to create an SDF file and convert the contents into an internal form usable by Vantage, see the description of the `tdlocaledef` utility in *Teradata Vantage™ - Database Utilities*, B035-1102.

## Example Display Formats

Examples of character string, numeric, and date formats appear in the following table.

FORMAT	Data	Display Form
X(6)	'HELLO'	'HELLO '
XXXXXX	'HELLO'	'HELLO '
X	'HELLO'	'H'
\$\$9.99	.079	' \$0.08 '
\$\$9.99	1095	*****
ZZ,ZZ9.99	1095	' 1,095.00 '
9.99E99	1095	'1.09E03 '
999V99	123.456	'12346 '
\$(5).9(2)	1	'\$1.00 '

FORMAT	Data	Display Form
G--(8)D9(2)	-12345678.90	' -12 345 678,90 ' where the SDF defines the group separator as '' (blank) and the radix separator as ','.
Z(I)D9(F)	000000.42	' ,42 ' where the SDF defines the radix separator as ','.
YY.DDD	85.224	' 85.224 '
MMMBDD,BYY	Sep 12, 85	' Sep 12, 85 '
YYYY-MM-DD	1996-02-14	' 1996-02-14 '
YYYYBMMMBDD	1985 Sep 12	' 1985 Sep 12 '
KatakanaEBCDIC with default format Note that KatakanaEBCDIC uppercases single-byte data.	mN<ABC>b	MN<ABC>B

## Examples

### Example: Without FORMAT Clause

If the CREATE TABLE statement defined the format for the Salary column as 'ZZ9999.99', the result of the statement:

```
SELECT SUM(Salary) FROM Employee;
```

is:

```
SUM(Salary)
-----
 851100.00
```

## Example: With FORMAT Clause

The result of the following statement:

```
SELECT SUM(Salary) (FORMAT '$$99,999.99')
FROM Employee;
```

is:

```
SUM(Salary)
-----
$851,100.00
```

FORMAT phrases, by themselves, cannot cause conversion of character to numeric data, or numeric to character data. An error message is returned if a FORMAT phrase implies data conversion.

## Example: Using FORMAT to Change the Format of Returned Data

Reconfigure the format of data returned:

```
SELECT empno (FORMAT '99-999'), name, deptno
FROM employee;
```

The system returns:

EmpNo	Name	DeptNo
-----	-----	-----
10-021	Smith T	700
10-007	Aguilar J	600
10-018	Russell S	300
10-011	Chin M	100

## Example: Using FORMAT to Override Default Format

Another example of overriding the column default and formatting the output:

```
SELECT salary (FORMAT '$$$,$$9.99')
FROM employee
WHERE empno = 10019 ;
```

The system returns:

```
Salary
-----
$28,600.00
```

## Example: Using FORMAT as Part of Derived Expression

You can use FORMAT as part of a derived expression. To determine the percent increase if employee 10019 is given a \$1000 raise:

```
SELECT (1000/salary)*100 (FORMAT 'zz9%')
      (TITLE 'Percent Incr')
FROM employee
WHERE empno = 10019 ;
```

```
Percent Incr
-----
                3%
```

## Related Information

For more information on ...	See ...
the display format of character data types	<a href="#">FORMAT Phrase and Character Formats.</a>
the display format of numeric data types	<a href="#">FORMAT Phrase and NUMERIC Formats.</a>
the display format of DateTime data types	<a href="#">FORMAT Phrase and DateTime Formats.</a>
the default data display formats	<a href="#">Data Type Default Formats.</a>
default format examples	the descriptions of each data type in this volume.
using the FORMAT phrase in data conversions	<i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145.
obtaining the format of an expression or column	the FORMAT function in <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145.
using Kanji date and time markers to format DateTime data types	<a href="#">FORMAT Phrase, DateTime Formats, and Japanese Character Sets.</a>

## FORMAT Phrase and Character Formats

### Default Result for Character Data

The following table shows the default if a CREATE TABLE, ALTER TABLE, or SELECT statement does not include the FORMAT phrase.

IF the data type is ...	THEN ...
CHARACTER	return the full length specified for a fixed-length CHARACTER column (that is, pad characters are returned unless TRIM is included in the SELECT statement).
VARCHAR	return the actual character string.
CLOB	<ul style="list-style-type: none"> <li>• If the server character set is LATIN and the length of the CLOB is &lt; 64000, then return the actual character string.</li> <li>• If the server character set is LATIN and the length of the CLOB is ≥ 64000, then return the first 64000 characters.</li> <li>• If the server character set is UNICODE and the length of the CLOB is &lt; 32000, then return the actual character string.</li> <li>• If the server character set is UNICODE and the length of the CLOB is ≥ 32000, then return the first 32000 characters.</li> </ul>

### Formatting Characters

Use the following characters in the FORMAT phrase to control formatting.

Character	Meaning
X X( <i>n</i> )	<p>The letter X formats the display of character data. Each X represents one character. The data is returned in left-to-right order.</p> <p>For example, if a column is defined as FORMAT 'X', only the first (leftmost) character of each field value is displayed.</p> <p>Multiple character positions can be defined either by using the (<i>n</i>) notation, where <i>n</i> defines the length of the string, or by entering the equivalent number of X characters.</p>

Character strings are padded or truncated on the right when they are shorter or longer than the positions defined in the FORMAT phrase.

### How FORMAT Applies to Server Character Sets

The following table shows how FORMAT 'X(*n*)' applies to server character sets.

IF a string expression has this character set ...	THEN FORMAT 'X( <i>n</i> )' returns ...
LATIN GRAPHIC	<i>n</i> logical characters.

IF a string expression has this character set ...	THEN FORMAT 'X(n)' returns ...
UNICODE	
KANJI1 KANJI SJIS	<i>n</i> bytes.

The character string in a FORMAT clause cannot be of type KANJI1. The form FORMAT `_kanji1 'hex_digits'XC` in particular always generates an error.

## Truncation of KANJI1 Character Set Fields [Deprecated]

### NOTICE

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

It is possible to truncate the output of multibyte characters in an expression with KANJI1 server character set with any of the following client character sets:

- KanjiEBCDIC
- KanjiEUC
- KanjiShift-JIS

In ANSI mode, an error occurs if a non-blank character is truncated.

In Teradata mode, the system returns the number of bytes specified and does *not* tell you when truncation occurs. The table below shows the hexadecimal returns. The actual return display depends on which system is used.

This character ...	With this format ...	Returns this value in hexadecimal ...	With this character set ...
B<AA>	'xx'	C20E	KanjiEBCDIC
B AA	'xx'	42A3	KanjiEUC

## Examples

### Example: FORMAT Phrase and Character Formats

Both of the following FORMAT phrases specify a five-position character string:

```
FORMAT 'X(5)'
FORMAT 'XXXXX'
```

## Example: Formatting Characters

Assume your data is the character string 'HELLO'.

IF the format is ...	THEN this is displayed ...
FORMAT 'X(6)'	HELLO with trailing blank
FORMAT 'X'	H

## FORMAT Phrase and NUMERIC Formats

The formatting characters in a FORMAT phrase determine whether the output of numeric data is considered to be monetary or non-monetary. Numeric information is considered to be monetary if the FORMAT phrase contains a currency symbol.

Formatting characters are case insensitive.

The result of a formatted numeric string is right-justified.

## Formatting Characters for Non-Monetary Numeric Information

Numeric information is considered to be non-monetary if the FORMAT phrase does not contain any currency or dual currency symbols. Use the following characters in the FORMAT phrase to control formatting of non-monetary numeric information.

Character	Meaning
G	<p>Invokes the grouping rule defined by <i>GroupingRule</i> in the SDF.</p> <p>The value of <i>GroupSeparator</i> in the SDF is copied to the output string to separate groups of digits to the left of the radix separator, according to the grouping rule defined by <i>GroupingRule</i>. Grouping applies to the integer portion of REAL, FLOAT, or DOUBLE PRECISION data.</p> <p>The G must appear as the first character in a FORMAT phrase.</p> <p>If the FORMAT phrase does not contain the letter G, no grouping is done on the output string.</p> <p>The G cannot appear in a FORMAT phrase that contains any of the following characters:</p> <ul style="list-style-type: none"> <li>• ,</li> <li>• .</li> <li>• /</li> <li>• :</li> <li>• S</li> </ul>

Character	Meaning
D	<p>Radix symbol.</p> <p>The value of <i>RadixSeparator</i> in the current SDF is copied to the output string whenever a D appears in the FORMAT phrase.</p> <p>The D cannot appear in a FORMAT phrase that contains any of the following characters:</p> <ul style="list-style-type: none"> <li>• .</li> <li>• ,</li> <li>• /</li> <li>• :</li> <li>• S</li> <li>• V</li> </ul>
/ : %	<p>Insertion characters.</p> <p>Copied to output string where they appear in the FORMAT phrase.</p> <p>The % insertion character cannot appear in a FORMAT phrase that contains S, and cannot appear between digits in a FORMAT phrase that contains G, D, or E. For example, G9999D99% is valid, but 9(9)D99%E999 is not.</p> <p>The / and : insertion characters cannot appear in a FORMAT phrase that contains any of the following characters:</p> <ul style="list-style-type: none"> <li>• G</li> <li>• D</li> <li>• S</li> </ul>
,	<p>Grouping character.</p> <p>The comma is inserted only if a digit has already appeared.</p> <p>The comma is interpreted as the grouping character regardless of the value of <i>GroupSeparator</i> in the SDF.</p> <p>The comma cannot appear in a FORMAT phrase that contains any of the following characters:</p> <ul style="list-style-type: none"> <li>• G</li> <li>• D</li> <li>• S</li> </ul>
.	<p>Radix character.</p> <p>The period is interpreted as the radix character, regardless of the value of <i>RadixSeparator</i> in the SDF, and is copied to the output string.</p> <p>The period cannot appear in a FORMAT phrase that contains any of the following characters:</p> <ul style="list-style-type: none"> <li>• G</li> <li>• D</li> <li>• V</li> <li>• S</li> </ul>
B	<p>Insertion character.</p> <p>A blank is copied to the output string wherever a B appears in the FORMAT phrase.</p> <p>B cannot appear between digits in a FORMAT phrase that contains G, D, or E. For example, 999B99B9999 is valid, but G9(9)BD99 is not.</p>
+ -	<p>Sign characters.</p>



Character	Meaning
	<p>Can appear at the beginning or end of a format string, but cannot appear between Z or 9 characters. One sign character places the edit character in a fixed position for the output string.</p> <p>If two or more of these characters are present, the sign floats (moves to the position just to the left of the number as determined by the stated structure). Repeated sign characters must appear to the left of a combination of the radix and any 9 formatting characters.</p> <p>If a group of repeated sign characters appears in a FORMAT phrase with a group of repeated Z characters, the group of Z characters must immediately follow the group of sign characters. For example, --ZZZ.</p> <p>One trailing sign character can occur to the right of any digits, and can combine with B. For example, G9(I)B+.</p> <p>The trailing sign character for a mantissa cannot appear to the right of the exponent. For example, 999D999E+999+ is not valid.</p> <p>The + translates to + or - as appropriate; the - translates to - or blank.</p>
V	<p>Implied decimal point position.</p> <p>Internally, the V is recognized as a decimal point to align the numeric value properly for calculation.</p> <p>Because the decimal point is implied, it does not occupy any space in storage and is not included in the output.</p> <p>V cannot appear in a FORMAT phrase that contains the D radix symbol or the . radix character.</p>
Z	<p>Zero-suppressed decimal digit.</p> <p>Translates to blank if the digit is zero and preceding digits are also zero.</p> <p>A FORMAT phrase containing Z characters (including Z(I) and Z(F)), a combination of commas, dots, G, or D, and no other formatting characters means "blank when zero."</p> <p>For example, ZZZZZ, ZZ.Z, GZ(I)DZ(F), GZZZZZZDZZ and Z,ZZZ.ZZ print only blanks if the number is zero.</p> <p>A Z cannot follow a 9.</p> <p>Repeated Z characters must appear to the left of any combination of the radix and any 9 formatting characters.</p> <p>The characters to the right of the radix cannot be a combination of 9 and Z characters; they must be all 9s or all Zs. If they are all Zs, then the characters to the left of the radix must also be all Zs.</p> <p>If a group of repeated Z characters appears in a FORMAT phrase with a group of repeated sign characters, the group of Z characters must immediately follow the group of sign characters. For example, --ZZZ.</p>
9	Decimal digit (no zero suppress).
CHAR( <i>n</i> )	<p>For more than one occurrence of a character, where <i>CHAR</i> can be:</p> <ul style="list-style-type: none"> <li>• - (sign character)</li> <li>• +</li> <li>• Z</li> <li>• 9</li> </ul> <p>and <i>n</i> can be:</p> <ul style="list-style-type: none"> <li>• an integer constant</li> <li>• I</li> <li>• F</li> </ul>

Character	Meaning
	<p>If <i>n</i> is F, <i>CHAR</i> can only be Z or 9.</p> <p>If <i>n</i> is an integer constant, the (<i>n</i>) notation means that the character repeats <i>n</i> number of times. For the meanings of I and F, see the definitions later in this table.</p>
I	<p>The number of characters needed to display the integer portion of numeric and integer data. I can only appear as <i>n</i> in the <i>CHAR(n)</i> character sequence (see the definition of <i>CHAR(n)</i> earlier in this table), where <i>CHAR</i> can be:</p> <ul style="list-style-type: none"> <li>• - (sign character)</li> <li>• +</li> <li>• Z</li> <li>• 9</li> </ul> <p><i>CHAR(I)</i> can only appear once, and is valid for the following types:</p> <ul style="list-style-type: none"> <li>• DECIMAL/NUMERIC</li> <li>• BYTEINT</li> <li>• SMALLINT</li> <li>• INTEGER</li> <li>• BIGINT</li> </ul> <p>The value of I is resolved during the formatting of the numeric data. The value is obtained from the declaration of the data type. For example, I is eight for the DECIMAL(10,2) type.</p> <p>If <i>CHAR(F)</i> also appears in the FORMAT phrase, <i>CHAR(F)</i> must appear to the right of <i>CHAR(I)</i>, and one of the following characters must appear between <i>CHAR(I)</i> and <i>CHAR(F)</i>:</p> <ul style="list-style-type: none"> <li>• D</li> <li>• .</li> <li>• V</li> </ul>
F	<p>The number of characters needed to display the fractional portion of numeric data. F can only appear as <i>n</i> in the <i>CHAR(n)</i> character sequence (see the definition of <i>CHAR(n)</i> earlier in this table), where <i>CHAR</i> can be:</p> <ul style="list-style-type: none"> <li>• Z</li> <li>• 9</li> </ul> <p><i>CHAR(F)</i> is valid for the DECIMAL/NUMERIC data type.</p> <p>The value of F is resolved during the formatting of the monetary numeric data. The value is obtained from the declaration of the data type. For example, F is two for the DECIMAL(10, 2) type.</p> <p>A value of zero for F displays no fractional precision for the data; however, the value of <i>RadixSeparator</i> in the current SDF is copied to the output string if a D appears in the FORMAT phrase.</p> <p><i>CHAR(F)</i> can appear only once. If <i>CHAR(I)</i> also appears in the FORMAT phrase, <i>CHAR(F)</i> must appear to the right of <i>CHAR(I)</i>, and one of the following characters must appear between <i>CHAR(I)</i> and <i>CHAR(F)</i>:</p> <ul style="list-style-type: none"> <li>• D</li> <li>• .</li> <li>• V</li> </ul>
-	<p>Dash character.</p> <p>Used when displaying numbers such as telephone numbers, social security numbers, and account numbers.</p>

Character	Meaning
	<p>A dash appears after the first digit and before the last digit, and is treated as an embedded dash rather than a sign character.</p> <p>A dash cannot follow any of these characters:</p> <ul style="list-style-type: none"> <li>• .</li> <li>• ,</li> <li>• +</li> <li>• G</li> <li>• D</li> <li>• E</li> <li>• S</li> <li>• V</li> </ul>
U+6642 時 U+5206 分 U+79D2 秒	<p>Kanji JI, FUN, and BYOU time markers.</p> <p>Each marker can appear only once.</p> <p>For more information on using Kanji time markers, see <a href="#">FORMAT Phrase, DateTime Formats, and Japanese Character Sets</a>.</p>
S	<p>Signed Zoned Decimal character.</p> <p>Defines signed zoned decimal input as a numeric data type and displays numeric output as signed zone decimal character strings.</p> <p>When converting signed zone decimal input to a numeric data type, the final character is converted according to the rules as follows:</p> <ul style="list-style-type: none"> <li>• Last character = { or 0, then the numeric conversion is n ... 0</li> <li>• Last character = A or 1, then the numeric conversion is n ... 1</li> <li>• Last character = B or 2, then the numeric conversion is n ... 2</li> <li>• Last character = C or 3, then the numeric conversion is n ... 3</li> <li>• Last character = D or 4, then the numeric conversion is n ... 4</li> <li>• Last character = E or 5, then the numeric conversion is n ... 5</li> <li>• Last character = F or 6, then the numeric conversion is n ... 6</li> <li>• Last character = G or 7, then the numeric conversion is n ... 7</li> <li>• Last character = H or 8, then the numeric conversion is n ... 8</li> <li>• Last character = I or 9, then the numeric conversion is n ... 9</li> <li>• Last character = }, then the numeric conversion is -n ... 0</li> <li>• Last character = J, then the numeric conversion is -n ... 1</li> <li>• Last character = K, then the numeric conversion is -n ... 2</li> <li>• Last character = L, then the numeric conversion is -n ... 3</li> <li>• Last character = M, then the numeric conversion is -n ... 4</li> <li>• Last character = N, then the numeric conversion is -n ... 5</li> <li>• Last character = O, then the numeric conversion is -n ... 6</li> <li>• Last character = P, then the numeric conversion is -n ... 7</li> <li>• Last character = Q, then the numeric conversion is -n ... 8</li> <li>• Last character = R, then the numeric conversion is -n ... 9</li> </ul> <p>When displaying numeric output as signed zone decimal character strings, the final character indicates the sign, as follows:</p> <p>If the final data digit is 0, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• { if the result is a positive number</li> </ul>

Character	Meaning
	<ul style="list-style-type: none"> <li>• } if the result is a negative number</li> </ul> <p>If the final data digit is 1, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• A if the result is a positive number</li> <li>• J if the result is a negative number</li> </ul> <p>If the final data digit is 2, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• B if the result is a positive number</li> <li>• K if the result is a negative number</li> </ul> <p>If the final data digit is 3, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• C if the result is a positive number</li> <li>• L if the result is a negative number</li> </ul> <p>If the final data digit is 4, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• D if the result is a positive number</li> <li>• M if the result is a negative number</li> </ul> <p>If the final data digit is 5, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• E if the result is a positive number</li> <li>• N if the result is a negative number</li> </ul> <p>If the final data digit is 6, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• F if the result is a positive number</li> <li>• O if the result is a negative number</li> </ul> <p>If the final data digit is 7, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• G if the result is a positive number</li> <li>• P if the result is a negative number</li> </ul> <p>If the final data digit is 8, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• H if the result is a positive number</li> <li>• Q if the result is a negative number</li> </ul> <p>If the final data digit is 9, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• I if the result is a positive number</li> <li>• R if the result is a negative number</li> </ul> <p>The S must follow the last decimal digit in the FORMAT phrase. It cannot appear in the same phrase with the following characters:</p> <ul style="list-style-type: none"> <li>• %</li> <li>• +</li> <li>• :</li> <li>• /</li> <li>• -</li> <li>• ,</li> <li>• .</li> <li>• D</li> <li>• G</li> <li>• Z</li> <li>• F</li> <li>• E</li> </ul> <p>For examples and details on signed zone decimal conversion, see <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i>, B035-1145.</p>
E	For exponential notation.

Character	Meaning
	Defines the end of the mantissa and the start of the exponent. The exponent consists of one optional + or - sign character followed by one or more 9 formatting characters.
FN9	Specifies a variable-length format that can be up to 64 characters as follows: <ol style="list-style-type: none"> <li>1. If the value is negative, a - sign character is generated, otherwise a space is generated.</li> <li>2. The integral portion of the value is generated (without leading zeroes or blanks).</li> <li>3. If a fractional portion is present, the value of <i>RadixSeparator</i> in the SDF is generated followed by the minimum number of digits required to express the fractional portion.</li> <li>4. If the resulting string is longer than 64 characters, the value is formatted using the FNE format. This format is defined later in this table.</li> </ol>
FNE	Specifies a variable-length exponential format as follows: <ol style="list-style-type: none"> <li>1. The value is rounded, according to the rounding method of the underlying type, to a maximum of 38 digits precision.</li> <li>2. If the value is negative, a - sign character is generated, otherwise a space is generated.</li> <li>3. If a single digit precision is sufficient for the mantissa, that digit is generated. Otherwise the highest order digit is generated followed by the value of <i>RadixSeparator</i> in the SDF. Then the remaining mantissa digits, up to 37 digits, are generated.</li> <li>4. The E character is generated.</li> <li>5. The signed two or, if necessary, three digit exponent is generated.</li> </ol>

## Formatting Characters for Monetary Numeric Information

Numeric information is considered to be monetary if the FORMAT phrase contains a currency symbol. Use the characters in the following table in the FORMAT phrase to control formatting of monetary numeric information.

Character	Meaning
G	<p>Invokes the currency grouping rule defined by <i>CurrencyGroupingRule</i> in the SDF. The value of <i>CurrencyGroupSeparator</i> in the current SDF is copied to the output string to separate groups of digits to the left of the currency radix separator, according to the currency grouping rule defined by <i>CurrencyGroupingRule</i>. Grouping applies to the integer portion of REAL, FLOAT, or DOUBLE PRECISION data.</p> <p>The G must appear as the first character in a FORMAT phrase. A currency character, such as L or C, must also appear in the FORMAT phrase.</p> <p>If the FORMAT phrase does not contain the letter G, no grouping is done on the output string. The G cannot appear in a FORMAT phrase that contains any of the following characters:</p> <ul style="list-style-type: none"> <li>• ,</li> <li>• .</li> <li>• /</li> <li>• :</li> <li>• S</li> </ul>
/	Insertion characters.
:	Copied to output string where they appear in the FORMAT phrase.

Character	Meaning
%	<p>The % insertion character cannot appear in a FORMAT phrase that contains S, and cannot appear between digits in a FORMAT phrase that contains G, D, or E. For example, GL9999D99% is valid, but L9(9)D99%E999 is not.</p> <p>The / and : insertion characters cannot appear in a FORMAT phrase that contains any of the following characters:</p> <ul style="list-style-type: none"> <li>• G</li> <li>• D</li> <li>• S</li> </ul>
B	<p>Insertion character.</p> <p>A blank is copied to the output string wherever a B appears in the FORMAT phrase. B cannot appear between digits in a FORMAT phrase that contains G, D, or E. For example, GNB99D99 is valid, but G9(9)BD99 is not.</p>
+ -	<p>Sign characters.</p> <p>These characters can appear at the beginning or end of a format string, but cannot appear between Z or 9 characters, or between repeated currency characters. One sign character places the edit character in a fixed position for the output string.</p> <p>If two or more of these characters are present, the sign floats (moves to the position just to the left of the number as determined by the stated structure). Repeated sign characters must appear to the left of formatting characters consisting of a combination of the radix and any 9 formatting characters.</p> <p>If a group of repeated sign characters appears in a FORMAT phrase with a group of repeated Z characters or a group of repeated currency characters or both, the groups must be contiguous. For example, +++\$\$\$\$ZZZ.</p> <p>One trailing sign character can occur to the right of any digits, and can combine with B and one currency character or currency sign. For example, G9(I)B+L.</p> <p>The trailing sign character for a mantissa cannot appear to the right of the exponent. For example, 999D999E+999+ is invalid.</p> <p>The + translates to + or - as appropriate; the - translates to - or blank.</p>
\$ £ ¥ ¤ €	<p>Currency signs:</p> <ul style="list-style-type: none"> <li>• \$ means Dollar sign.</li> <li>• £ means Pound sterling.</li> <li>• ¥ means Yen sign.</li> <li>• ¤ means general currency sign.</li> <li>• € means Euro sign.</li> </ul> <p>A currency sign cannot appear between Z or 9 formatting characters, or between repeated sign characters.</p> <p>One currency sign places the edit character in a fixed position for the output string.</p> <p>If a result is formatted using a single currency sign with Zs for zero-suppressed decimal digits (for example, £ZZ9.99), blanks can occur between the currency sign and the leftmost nonzero digit of the number.</p> <p>If the same currency sign appears more than once, the sign floats to the right, leaving no blanks between it and the leftmost digit.</p> <p>A currency sign cannot appear in the same phrase with a currency character, such as L.</p> <p>If + or - is present, the currency character cannot precede it.</p> <p>If a group of repeated currency signs appears in a FORMAT phrase with a group of repeated sign characters or a group of repeated Z characters or both, the groups must be contiguous. For example, +++\$\$\$\$ZZZ.</p>

Character	Meaning
	<p>One currency sign can occur to the right of any digits, and can combine with B and one trailing sign character. For example, G9(I)B+\$.</p> <p>A currency sign cannot appear to the right of an exponent. For example, 999D999E+999B+\$ is invalid.</p>
L C N O U A	<p>Currency characters.</p> <p>The value of the corresponding currency string in the current SDF is copied to the output string whenever the character appears in the FORMAT phrase.</p> <ul style="list-style-type: none"> <li>• L in a FORMAT phrase is interpreted as the currency symbol and the value of the <i>Currency</i> string in the SDF is copied to the output string.</li> <li>• C in a FORMAT phrase is interpreted as the ISO currency symbol and the value of the <i>ISOCurrency</i> string in the SDF is copied to the output string.</li> <li>• N in a FORMAT phrase is interpreted as the full currency name, such as Yen or Kroner, and the value of the <i>CurrencyName</i> string in the SDF is copied to the output string.</li> <li>• O in a FORMAT phrase is interpreted as the dual currency symbol and the value of the <i>DualCurrency</i> string in the SDF is copied to the output string.</li> <li>• U in a FORMAT phrase is interpreted as the dual ISO currency symbol and the value of the <i>DualISOCurrency</i> string in the SDF is copied to the output string.</li> <li>• A in a FORMAT phrase is interpreted as the full dual currency name, such as Euro, and the value of the <i>DualCurrencyName</i> string in the SDF is copied to the output string.</li> </ul> <p>A currency character cannot appear between Z or 9 formatting characters, or between repeated sign characters.</p> <p>If the same currency character appears more than once, the value that is copied to the output string floats to the right, leaving no blanks between it and the leftmost digit. Repeated characters must be contiguous, and must appear to the left formatting characters consisting of a combination of the radix and any 9 formatting characters.</p> <p>If a group of repeated currency characters appears in a FORMAT phrase with a group of repeated sign characters or a group of repeated Z characters or both, the groups must be contiguous. For example, +++LLLZZZ.</p> <p>A currency character cannot appear in the same phrase with any of the following characters:</p> <ul style="list-style-type: none"> <li>• other currency characters</li> <li>• a currency sign, such as \$ or £</li> <li>• ,</li> <li>• .</li> </ul> <p>One currency character can occur to the right of any digits, and can combine with B and one trailing sign character. For example, G9(I)B+L.</p> <p>A currency character cannot appear to the right of an exponent. For example, 999D999E+999B+L is invalid.</p>
V	<p>Implied decimal point position.</p> <p>Internally, the V is recognized as a decimal point to align the numeric value properly for calculation.</p> <p>Because the decimal point is implied, it does not occupy any space in storage and is not included in the output.</p> <p>V cannot appear in a FORMAT phrase that contains the 'D' radix symbol or the '.' radix character.</p>
Z	<p>Zero-suppressed decimal digit.</p> <p>Translates to blank if the digit is zero and preceding digits are also zero.</p> <p>A Z cannot follow a 9.</p>

Character	Meaning
	<p>Repeated Z characters must appear to the left of any combination of the radix and any 9 formatting characters.</p> <p>The characters to the right of the radix cannot be a combination of 9 and Z characters; they must be all 9s or all Zs. If they are all Zs, then the characters to the left of the radix must also be all Zs.</p> <p>If a group of repeated Z characters appears in a FORMAT phrase with a group of repeated sign characters, the group of Z characters must immediately follow the group of sign characters. For example, --ZZZ.</p>
9	Decimal digit (no zero suppress).
E	<p>For exponential notation.</p> <p>Defines the end of the mantissa and the start of the exponent.</p> <p>The exponent consists of one optional + or - sign character followed by one or more 9 formatting characters.</p>
CHAR( <i>n</i> )	<p>For more than one occurrence of a character, where <i>CHAR</i> can be one of the following:</p> <ul style="list-style-type: none"> <li>• - (sign character)</li> <li>• +</li> <li>• Z</li> <li>• 9</li> <li>• \$</li> <li>• ¤</li> <li>• ¢</li> <li>• ¥</li> <li>• £</li> </ul> <p>and <i>n</i> can be:</p> <ul style="list-style-type: none"> <li>• an integer constant</li> <li>• I</li> <li>• F</li> </ul> <p>If <i>n</i> is F, <i>CHAR</i> can only be Z or 9.</p> <p>If <i>n</i> is an integer constant, the (<i>n</i>) notation means that the character repeats <i>n</i> number of times. For the meanings of I and F, see the definitions later in this table.</p>
,	<p>Currency grouping character.</p> <p>The comma is inserted only if a digit has already appeared.</p> <p>The comma is interpreted as the currency grouping character regardless of the value of the <i>CurrencyGroupSeparator</i> in the SDF.</p> <p>The comma cannot appear in a FORMAT phrase that contains any of the following characters:</p> <ul style="list-style-type: none"> <li>• G</li> <li>• D</li> <li>• L</li> <li>• C</li> <li>• O</li> <li>• U</li> <li>• N</li> <li>• A</li> </ul>



Character	Meaning
	<ul style="list-style-type: none"> <li>• S</li> </ul>
.	<p>Currency radix character.</p> <p>The period is interpreted as the currency radix character, regardless of the value of the <i>CurrencyRadixSeparator</i> in the SDF, and is copied to the output string.</p> <p>The period cannot appear in a FORMAT phrase that contains any of the following characters:</p> <ul style="list-style-type: none"> <li>• G</li> <li>• D</li> <li>• L</li> <li>• V</li> <li>• C</li> <li>• O</li> <li>• U</li> <li>• N</li> <li>• A</li> <li>• S</li> </ul>
D	<p>Radix symbol.</p> <p>The value of <i>CurrencyRadixSeparator</i> in the current SDF is copied to the output string whenever a D appears in the FORMAT phrase.</p> <p>A currency symbol, such as a dollar sign or yen sign, must also appear in the FORMAT phrase.</p> <p>The D cannot appear in a FORMAT phrase that contains any of the following characters:</p> <ul style="list-style-type: none"> <li>• ,</li> <li>• .</li> <li>• /</li> <li>• :</li> <li>• S</li> <li>• V</li> </ul>
I	<p>The number of characters needed to display the integer portion of numeric and integer data.</p> <p>I can only appear as <i>n</i> in the <i>CHAR(n)</i> character sequence (see the definition of <i>CHAR(n)</i> earlier in this table), where <i>CHAR</i> can be:</p> <ul style="list-style-type: none"> <li>• - (sign character)</li> <li>• +</li> <li>• Z</li> <li>• 9</li> <li>• \$</li> <li>• ¤</li> <li>• ¢</li> <li>• ¥</li> <li>• £</li> </ul> <p><i>CHAR(I)</i> can only appear once, and is valid for the following types:</p> <ul style="list-style-type: none"> <li>• DECIMAL/NUMERIC</li> <li>• BYTEINT</li> <li>• SMALLINT</li> <li>• INTEGER</li> </ul>

Character	Meaning
	<ul style="list-style-type: none"> <li>• BIGINT</li> </ul> <p>The value of I is resolved during the formatting of the monetary numeric data. The value is obtained from the declaration of the data type. For example, I is eight for the DECIMAL(10, 2) type.</p> <p>If <i>CHAR(F)</i> also appears in the FORMAT phrase, <i>CHAR(F)</i> must appear to the right of <i>CHAR(I)</i>, and one of the following characters must appear between <i>CHAR(I)</i> and <i>CHAR(F)</i>:</p> <ul style="list-style-type: none"> <li>• D</li> <li>• .</li> <li>• V</li> </ul>
F	<p>The number of characters needed to display the fractional portion of numeric data. F can only appear as <i>n</i> in the <i>CHAR(n)</i> character sequence (see the definition of <i>CHAR(n)</i> earlier in this table), where <i>CHAR</i> can be:</p> <ul style="list-style-type: none"> <li>• Z</li> <li>• 9</li> </ul> <p><i>CHAR(F)</i> is valid for the DECIMAL/NUMERIC data type.</p> <p>The value of F is resolved during the formatting of the monetary numeric data. The value is obtained from the declaration of the data type. For example, F is two for the DECIMAL(10, 2) type.</p> <p>A value of zero for F displays no fractional precision for the data; however, the value of <i>CurrencyRadixSeparator</i> in the current SDF is copied to the output string if a D appears in the FORMAT phrase.</p> <p><i>CHAR(F)</i> can appear only once. If <i>CHAR(I)</i> also appears in the FORMAT phrase, <i>CHAR(F)</i> must appear to the right of <i>CHAR(I)</i>, and one of the following characters must appear between <i>CHAR(I)</i> and <i>CHAR(F)</i>:</p> <ul style="list-style-type: none"> <li>• D</li> <li>• .</li> <li>• V</li> </ul>
-	<p>Dash character.</p> <p>Used when storing numbers such as telephone numbers, social security numbers, and account numbers.</p> <p>A dash appears after the first digit and before the last digit, and is treated as an embedded dash rather than a sign character. A dash cannot follow any of these characters:</p> <ul style="list-style-type: none"> <li>• .</li> <li>• ,</li> <li>• +</li> <li>• G</li> <li>• N</li> <li>• A</li> <li>• C</li> <li>• L</li> <li>• O</li> <li>• U</li> <li>• D</li> <li>• V</li> <li>• S</li> </ul>

Character	Meaning
	<ul style="list-style-type: none"> <li>• E</li> <li>• \$</li> <li>• ¤</li> <li>• £</li> <li>• ¥</li> <li>• ¤</li> </ul>
S	<p>Signed Zoned Decimal character.</p> <p>Defines signed zoned decimal input as a numeric data type and displays numeric output as signed zone decimal character strings.</p> <p>When converting signed zone decimal input to a numeric data type, the final character is converted as follows:</p> <ul style="list-style-type: none"> <li>• Last character = { or 0, then the numeric conversion is n ... 0</li> <li>• Last character = A or 1, then the numeric conversion is n ... 1</li> <li>• Last character = B or 2, then the numeric conversion is n ... 2</li> <li>• Last character = C or 3, then the numeric conversion is n ... 3</li> <li>• Last character = D or 4, then the numeric conversion is n ... 4</li> <li>• Last character = E or 5, then the numeric conversion is n ... 5</li> <li>• Last character = F or 6, then the numeric conversion is n ... 6</li> <li>• Last character = G or 7, then the numeric conversion is n ... 7</li> <li>• Last character = H or 8, then the numeric conversion is n ... 8</li> <li>• Last character = I or 9, then the numeric conversion is n ... 9</li> <li>• Last character = }, then the numeric conversion is -n ... 0</li> <li>• Last character = J, then the numeric conversion is -n ... 1</li> <li>• Last character = K, then the numeric conversion is -n ... 2</li> <li>• Last character = L, then the numeric conversion is -n ... 3</li> <li>• Last character = M, then the numeric conversion is -n ... 4</li> <li>• Last character = N, then the numeric conversion is -n ... 5</li> <li>• Last character = O, then the numeric conversion is -n ... 6</li> <li>• Last character = P, then the numeric conversion is -n ... 7</li> <li>• Last character = Q, then the numeric conversion is -n ... 8</li> <li>• Last character = R, then the numeric conversion is -n ... 9</li> </ul> <p>When displaying numeric output as signed zone decimal character strings, the final character indicates the sign, as follows:</p> <p>If the final data digit is 0, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• { if the result is a positive number</li> <li>• } if the result is a negative number</li> </ul> <p>If the final data digit is 1, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• A if the result is a positive number</li> <li>• J if the result is a negative number</li> </ul> <p>If the final data digit is 2, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• B if the result is a positive number</li> <li>• K if the result is a negative number</li> </ul> <p>If the final data digit is 3, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• C if the result is a positive number</li> </ul>

Character	Meaning
	<ul style="list-style-type: none"> <li>• L if the result is a negative number</li> </ul> <p>If the final data digit is 4, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• D if the result is a positive number</li> <li>• M if the result is a negative number</li> </ul> <p>If the final data digit is 5, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• E if the result is a positive number</li> <li>• N if the result is a negative number</li> </ul> <p>If the final data digit is 6, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• F if the result is a positive number</li> <li>• O if the result is a negative number</li> </ul> <p>If the final data digit is 7, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• G if the result is a positive number</li> <li>• P if the result is a negative number</li> </ul> <p>If the final data digit is 8, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• H if the result is a positive number</li> <li>• Q if the result is a negative number</li> </ul> <p>If the final data digit is 9, then the final result digit is displayed as:</p> <ul style="list-style-type: none"> <li>• I if the result is a positive number</li> <li>• R if the result is a negative number</li> </ul> <p>The S must follow the last decimal digit in the FORMAT phrase. It cannot appear in the same phrase with the following characters.</p> <ul style="list-style-type: none"> <li>• %</li> <li>• +</li> <li>• :</li> <li>• /</li> <li>• -</li> <li>• ,</li> <li>• .</li> <li>• Z</li> <li>• E</li> <li>• D</li> <li>• G</li> <li>• F</li> <li>• N</li> <li>• A</li> <li>• C</li> <li>• L</li> <li>• U</li> <li>• O</li> <li>• \$</li> <li>• ¤</li> <li>• £</li> <li>• ¥</li> <li>• ¤</li> </ul>

Character	Meaning
	For examples and details on signed zone decimal conversion, see <i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145.

## Results of a FORMAT Phrase Defining Too Few Positions

A FORMAT phrase that defines fewer positions than are required by numeric values causes the data to be returned as follows:

- Asterisks appear when the integer portion cannot be accommodated.
- When only the integer portion can be accommodated, any digits to the right of the least significant digit are either truncated (for an integer value) or rounded (for a floating, number, or decimal value).

Rounding is based on “Round to the Nearest” mode, as illustrated by the following process.

1. Let B represent the actual result.
2. Let A and C represent the nearest bracketing values that can be represented, such that  $A < B < C$ .
3. The determination as to whether A or C is the represented result is made as follows
  - a. When possible, the result is the value nearest to B.
  - b. If A and C are equidistant (for example, the fractional part is exactly .5), the result is the even number.

## Examples: Rounding

The following SELECT statement returns 48 and 48.

```
SELECT 47.5(FORMAT 'zzzz'), 48.5(FORMAT('zzzz')) ;
```

A FORMAT clause can cause the number to be rounded; for example, consider the following:

```
SELECT 1.3451 (FORMAT 'zz.z');
```

The FORMAT clause operates on 1.3451 and returns 1.3.

```
SELECT 13451 / 10000.000
```

1.345 is returned as expected.

```
SELECT 13451 / 10000.000 (FORMAT 'zz.z');
```

The arithmetic expression is evaluated first (yielding 1.345), then the FORMAT clause is applied, returning 1.3.

```
SELECT 13451 / 10000.00 (FORMAT 'zz.z');
```

Note that two ‘roundings’ occur; first to 1.35 when evaluating the expression, then a second rounding to 1.4 as a result of the FORMAT statement.

To avoid double ‘roundings’, extend the precision of the arithmetic expression two decimal places beyond that of the FORMAT clause.

## Examples: Display Results

The display results of various FORMAT phrases for numeric data appear in the following table. For FORMAT phrases that contain G, D, L, and N formatting characters, assume that the related entries in the SDF are:

```
RadixSeparator {"",""}
GroupSeparator {"."}
GroupingRule {"3"}
Currency {"$"}
CurrencyName {"US Dollars"}
```

FORMAT Phrase	Data	Result
'\$\$9.99'	.069	\$0.07
'\$\$9.99'	1095	*****
'ZZ,ZZ9.99'	1095	1,095.00
'9.99E99'	1095	1.10E03
'999V99'	128.457	128.46
'\$(5).9(2)'	1	\$1.00
'999-9999'	8278777	827-8777
'ZZ,ZZ9.99-'	1095	1,095.00
'ZZ,ZZ9.99-'	-1095	1,095.00-
'99999S'	-1095	0109N
'99999S'	1095	0109E
'--(8)D9(2)'	0034567890	34567890,00
'G--(8)D9(2)'	-12345678.90	-12.345.678,90
'Z(I)D9(F)'	000000.42	,42
'G-(10)9'	1234567890	1.234.567.890

FORMAT Phrase	Data	Result
'-9D9999999999999999E-999'	1.74524064372835 e-2	1,74524064372835 e-002
'GLLZ(I)D9(F)'	9988.77	\$9.988,77
'-Z(I)BN'	998877.66	998878 US Dollars

If the FORMAT phrase does not include a sign character or a signed zoned decimal character, then the sign for a negative value is discarded and the output is displayed as a positive number.

## FORMAT Phrase and DateTime Formats

The date and time formatting characters in a FORMAT phrase determine the output of DATE, TIME, and TIMESTAMP information.

IF the data type is ...	THEN use formatting characters for ...
DATE PERIOD(DATE)	date information. For Period types, the specified format is associated with both the beginning and ending bounds of the period.
TIME TIME WITH TIME ZONE PERIOD(TIME)	time information. For Period types, the specified format is associated with both the beginning and ending bounds of the period.
TIMESTAMP TIMESTAMP WITH TIME ZONE PERIOD(TIMESTAMP)	date and time information. Date formatting characters must be grouped separately from time formatting characters. For Period types, the specified format is associated with both the beginning and ending bounds of the period.

Formatting characters are case-insensitive.

## Formatting Characters for Date Information

Use the following characters in the FORMAT phrase to control formatting of date information in DATE, PERIOD(DATE), PERIOD(TIMESTAMP), and TIMESTAMP types:

Characters	Meaning
MMMM M4	Represent the month as a full month name, such as November. Valid names are specified by <i>LongMonths</i> in the current SDF. M4 is equivalent to MMMM, and is preferable to allow for a shorter, unambiguous format string. You cannot specify M4 in a format that also has M3 or MM.
MMM M3	Represent the month as an abbreviated month name, such as 'Apr' for April. Valid names are specified by <i>ShortMonths</i> in the current SDF.

Characters	Meaning
	M3 is equivalent to MMM, and is preferable to allow for a shorter, unambiguous format string. You cannot specify MMM in a format that also has MM.
MM	Represent the month as two numeric digits.
DDD D3	Represent the date as the sequential day in the year, using three numeric digits, such as '032' as February 1. D3 is equivalent to DDD, and allows for a shorter format string. You cannot specify DDD or D3 in a format that also has DD.
DD	Represent the day of the month as two numeric digits.
YYYY Y4	Represent the year as four numeric digits. Y4 is equivalent to YYYY, and allows for a shorter format string. You cannot specify YYYY or Y4 in a format that also has YY.
YY	Represent the year as two numeric digits.
EEEE E4	Represent the day of the week using the full name, such as Thursday. Valid names are specified by <i>LongDays</i> in the current SDF. E4 is equivalent to EEEEE, and allows for a shorter format string.
EEE E3	Represent the day of the week as an abbreviated name, such as 'Mon' for Monday. Valid abbreviations are specified by <i>ShortDays</i> in the current SDF. E3 is equivalent to EEE, and allows for a shorter format string.
/	Slash separator. Copied to output string where it appears in the FORMAT phrase. This is the default separator for Teradata dates.
B b	Blank representation separator. Use this instead of a space to represent a blank.
,	Comma separator. Copied to output string where it appears in the FORMAT phrase.
'	Apostrophe separator. Copied to output string where it appears in the FORMAT phrase.
:	Colon separator. Copied to output string where it appears in the FORMAT phrase.
.	Period separator. Copied to output string where it appears in the FORMAT phrase.
-	Dash separator. Copied to output string where it appears in the FORMAT phrase. This is the default separator for ANSI dates.
9	Decimal digit. This formatting character can only be used with separators less than 0x009F.



Characters	Meaning
	<p>The 9(<i>n</i>) notation can be used for more than one occurrence of this character, where <i>n</i> is an integer constant and means that the '9' repeats <i>n</i> number of times.</p> <p>This formatting character is for DATE and PERIOD(DATE) types only and cannot appear as a date formatting character for PERIOD(TIMESTAMP) and TIMESTAMP types.</p>
Z	<p>Zero-suppressed decimal digit.</p> <p>This formatting character can only be used with separators less than 0x009F.</p> <p>The Z(<i>n</i>) notation can be used for more than one occurrence of this character, where <i>n</i> is an integer constant and means that the 'Z' repeats <i>n</i> number of times.</p> <p>This formatting character is for DATE and PERIOD(DATE) types only and cannot appear as a date formatting character for PERIOD(TIMESTAMP) and TIMESTAMP types.</p>

## Formatting Characters for Time Information

Use the following characters in the FORMAT phrase to control formatting of time information in PERIOD(TIME), PERIOD(TIMESTAMP), TIME and TIMESTAMP types:

Characters	Meaning
HH	Represent the hour as two numeric digits.
MI	Represent the minute as two numeric digits.
SS	Represent the second as two numeric digits.
S( <i>n</i> ) S(F)	<p>Number of fractional seconds.</p> <p>Replace <i>n</i> with a number between 0 and 6, or use F for the number of characters needed to display the fractional seconds precision.</p> <p>The value of F is resolved during the formatting of the TIME or TIMESTAMP data. The value is obtained from the fractional seconds precision in the declaration of the data type. For example, F is two for the TIME(2) type.</p> <p>A value of zero for F displays no radix symbol and no fractional precision for the data.</p> <p>The S(F) formatting characters must follow a D formatting character or a . separator character.</p> <p>A value of <i>n</i> that is less than the PERIOD(TIME), PERIOD(TIMESTAMP), TIME or TIMESTAMP fractional second precision produces an error.</p>
D	<p>Radix symbol.</p> <p>The value of <i>RadixSeparator</i> in the current SDF is copied to the output string whenever a D appears in the FORMAT phrase.</p> <p>Separator characters, such as . or :, can also appear in the FORMAT phrase, but only if they do not match the value of <i>RadixSeparator</i>.</p>
T	<p>Represent time in 12-hour format instead of 24-hour format.</p> <p>The appropriate time of day, as specified by <i>AMPM</i> in the current SDF is copied to the output string where a T appears in the FORMAT phrase.</p>
Z	Time zone.

Characters	Meaning
	<p>The Z controls the placement of the time zone in the output of PERIOD(TIME), PERIOD(TIMESTAMP), TIME and TIMESTAMP data, and can only appear at the beginning or end of the time formatting characters.</p> <p>For example, the following statement uses a FORMAT phrase that includes a Z before the time formatting characters:</p> <pre>SELECT CURRENT_TIMESTAMP   (FORMAT 'YYYY-MM-DDBZBHH:MI:SS.S(6)');</pre> <p>If the PERIOD(TIME), PERIOD(TIMESTAMP), TIME or TIMESTAMP data contains time zone data, the time zone is copied to the output string. The time zone format is +HH:MI or -HH:MI, depending on the time zone hour displacement.</p>
:	<p>Colon separator.</p> <p>Copied to output string where it appears in the FORMAT phrase. This is the default separator for ANSI time.</p> <p>This character cannot appear in the FORMAT phrase if the value of <i>RadixSeparator</i> in the current SDF is a colon.</p>
.	<p>Period separator.</p> <p>This can also be used to indicate the fractional seconds.</p> <p>Copied to output string where it appears in the FORMAT phrase.</p> <p>This character cannot appear in the FORMAT phrase if the value of <i>RadixSeparator</i> in the current SDF is a period.</p>
-	<p>Dash separator.</p> <p>Copied to output string where it appears in the FORMAT phrase.</p>
h	<p>Hour separator.</p> <p>A lowercase h character is copied to the output string.</p> <p>The h formatting character must follow the HH formatting characters.</p> <p>This character cannot appear in the FORMAT phrase if the value of <i>RadixSeparator</i> in the current SDF is a lowercase h character.</p>
m	<p>Minute separator.</p> <p>A lowercase m character is copied to the output string.</p> <p>The m formatting character must follow the MI formatting characters.</p> <p>This character cannot appear in the FORMAT phrase if the value of <i>RadixSeparator</i> in the current SDF is a lowercase m character.</p>
s	<p>Second separator.</p> <p>A lowercase s character is copied to the output string.</p> <p>The s formatting character must follow SS or SSDS(F) formatting characters.</p> <p>This character cannot appear in the FORMAT phrase if the value of <i>RadixSeparator</i> in the current SDF is a lowercase s character.</p>
B b	<p>Blank representation separator.</p> <p>Use this instead of a space to represent a blank.</p> <p>This character cannot appear in the FORMAT phrase if the value of <i>RadixSeparator</i> in the current SDF is a blank.</p>

## Kanji Date and Time Markers

For information on using Kanji date and time markers, see [FORMAT Phrase, DateTime Formats, and Japanese Character Sets](#).

## Examples: Date Formats

The following table is a nonexhaustive list of formats that can be used to present an output date, where the data is 85/09/12:

FORMAT Phrase	Result
FORMAT 'YY/MM/DD'	85/09/12
FORMAT 'DD-MM-YY'	12-09-85
FORMAT 'YYYY/MM/DD'	1985/09/12
FORMAT 'YYYY-MM-DD'	1985-09-12
FORMAT 'YYYY.DDD'	1985.225
FORMAT 'YBDDD'	85 225
FORMAT 'DDBMMMBYYYY'	12 Sep 1985
FORMAT 'MMMBDD,BYYYY'	Sep 12, 1985
FORMAT 'YYYYBMMMBDD'	1985 Sep 12
FORMAT 'MMM'	Sep
FORMAT 'EEE,BM4BDD,BYYYY'	Thu, September 12, 1985
FORMAT 'E4,BMMMBDD,BYYYY'	Thursday, September 12, 1985
FORMAT 'E4BDDBM4BYYYY'	Jeudi 12 Septembre 1985 (Jeudi is French for Thursday and Septembre is French for September.)
FORMAT '999999'	850912

## Examples: Time Formats

The following table is a nonexhaustive list of formats that can be used to present an output time, where the data is 13:20:53.64+03:00:

FORMAT Phrase	Result
FORMAT 'HH:MIBT'	01:20 PM
FORMAT 'HH:MI'	13:20

FORMAT Phrase	Result
FORMAT 'HH.MI.SS'	13.20.53
FORMAT 'HH:MI:SSBT'	01:20:53 Nachm (Nachm is German for PM.)
FORMAT 'HH:MI:SSDS(F)'	13:20:53.64
FORMAT 'HH:MI:SSDS(F)Z'	13:20:53.64+03:00
FORMAT 'HHhMIhSSs'	13h20m53s

## Examples: Timestamp Formats

The following table is a nonexhaustive list of formats that can be used to present an output timestamp, where the data is 85/09/12 13:20:53.64+03:00:

FORMAT Phrase	Result
FORMAT 'MM/DD/YBHH:MIBT'	09/12/85 01:20 PM
FORMAT 'MMMBDD,BYYBHH:MI:SS'	Sep 12, 85 13:20:53
FORMAT 'E3,BM4BDD,BY4BHH:MI:SSDS(F)'	Thu, September 12, 1985 13:20:53.64
FORMAT 'YYYY-MM-DOBHH:MI:SSDS(F)Z'	1985-09-12 13:20:53.64+03:00

## Format Consistency

If a field is declared as type DATE and a FORMAT phrase is specified for it, the date must be entered in the specified format and it must be enclosed in apostrophes.

For example, data supplied to, or used in, a conditional WHERE or HAVING clause for the following column must be defined as 'OCT 24, 1985'.

```
DOB DATE FORMAT 'MMMBDD,bYYYY'
```

This format consistency restriction applies only to the date values themselves, and *not* to the date separators. The input date separators need not match the FORMAT phrase date separators. In addition, you can enter a 4-digit year even if the FORMAT phrase specifies the year as 'YY'.

## DATE Comparisons

If a field is to be inserted into a DATE column, the format must match either the ANSI DATE literal format or the format of the column. If a field is to be compared with a DATE value, the format must match one of the following:

- ANSI DATE literal format
- Format of the DATE value
- DATE format determined by DateForm and default DATE data type format in the SDF. See [DATE Formats](#).

For example, the comparison works if the data is CHAR(8) in the form YY/MM/DD and a DATE column format is YY/MM/DD. The comparison fails, however, if the column format is YYYY-MM-DD.

To perform comparisons that do not meet these qualifications, convert the values as described in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Examples: Querying Using Date Formats

Use a statement like the following to display a date in uppercase:

```
SELECT DATE (FORMAT 'MMMbdd,bYYYY') (CHAR(12), UC);
```

Using 1985-09-12 for data, this statement returns:

```
SEP 12, 1985
```

The following query shows how to specify a date if, for example, the p\_date column was specified as FORMAT 'DDBMMMBYYYY':

```
SELECT *
FROM sales
WHERE p_date = '30 Mar 1994';
```

If the p\_date column was specified as FORMAT 'YYYY-MM-DD', the query would be as follows:

```
SELECT *
FROM sales
WHERE p_date = '1994-03-30';
```

## Examples: Using Time Formats

Create a table using the FORMAT phrase to specify the output format:

```
CREATE TABLE t1
(f1 TIME(3) WITH TIME ZONE FORMAT 'HH:MI:SS.S(F)'
,f2 TIMESTAMP(4) FORMAT 'YYYY-MM-DDBHH:MI:SS.S(F)Z');
```

Populate the table using TIME and TIMESTAMP literals.

```
INSERT t1 (TIME '10:44:25.123-08:00',
TIMESTAMP '2000-09-20 10:44:25.1234');
```

Query the data:

```
SELECT f1, f2
FROM t1;
```

The query returns:

f1	f2
-----	
10:44:25.123	2000-09-20 10:44:25.1234

Column f1 was defined as a TIME WITH TIME ZONE data type, but the FORMAT did not include the Z formatting character, so the time zone does not appear in the output. Column f2 was defined as a TIMESTAMP data type, and the FORMAT includes the Z formatting character. However the timestamp data is not associated with any time zone, so no time zone information appears in the output.

To display the time zone information for column f1, use the FORMAT phrase in the SELECT:

```
SELECT CAST (f1 AS TIME(3) WITH TIME ZONE FORMAT 'HH:MI:SS.S(F)Z')
FROM t1;
```

The query returns:

f1
-----
10:44:25.123-8:00

For more information on TIME and TIMESTAMP literals, see [Data Literals](#).

For more information on using the FORMAT phrase in data type conversions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Record Mode Import Anomaly

When data is imported in Record Mode into a NULLABLE DATE field, and the source data is a binary integer of value zero, then the field is set to NULL (not zero).

## FORMAT Phrase, DateTime Formats, and Japanese Character Sets

The FORMAT phrase can present the date and time in Japanese format.

## Separators

FORMAT allows symbols like SLASH and COMMA as separators between day, month, and year. Non-Kanji characters in the FORMAT clause are halfwidth.

For example, applying (FORMAT 'YY/MM/DD') to 920310 yields 92/03/10.

## Kanji Date Markers

The FORMAT phrase can contain the following Kanji date markers to separate day, month, and year information in DATE and TIMESTAMP data types.

Use this ideograph ...	After these formatting characters ...	To describe ...
U+5E74 (年)	'YYYY' 'YY'	Year
U+6708 (月)	'MM'	Month
U+65E5 (日)	'DD'	Day

For example, apply (FORMAT 'YY 年 MM 月 DD 日') to 920310 to yield 92 年 03 月 10 日.

Because Kanji characters have no canonical form when represented by the KANJI1 character data type, the hexadecimal KANJI1 literal cannot be used for the FORMAT clause.

The Kanji year, month, and day designators can appear in the FORMAT clause in any order, for example, (FORMAT 'MM 月 YY 年 DD 日')

Date formatting for DATE and TIMESTAMP data types also supports the last five Japanese Imperial Eras, as described in the following table.

Era	Kanji Ideographs	Start Date
Meiji	明治	1867/1/9
Taisho	大正	1912/7/30
Showa	昭和	1926/12/25
Heisei	平成	1989/1/8
Reiwa	令和	2019/5/1

To enable the Japanese Imperial Era formatting, the year designator '和暦 YY 年' must be used in the format clause (U+548C (和) U+66A6 (暦) indicates the WA and REKI ideographs). For example, (FORMAT '和暦 YY 年 MM 月 DD 日'), (FORMAT '和暦 YY 年 DD 日 MM 月'), and (FORMAT '和暦 YY 年 DD 日') are valid, but (FORMAT '和暦 MM 月 DD 日') is incorrect and an error is generated.

Applying (FORMAT '和暦 YY 年 MM 月 DD 日') to 920310 yields 平成 04 年 03 月 10 日, where 平成 are the two ideographs that together indicate the Heisei era, and 年, 月, and 日 are the NEN, GATSU, and NICHI ideographs respectively (04 stands for the fourth year of the Heisei era).

## Kanji Time Markers

Kanji characters are valid in the FORMAT clause as markers for time information in TIME and TIMESTAMP data types. The following table lists the appropriate markers, their Kanji characters, and their description.

Marker	Kanji Character	Description
JI	U+6642 (時)	Hours
FUN	U+5206 (分)	Minutes
BYOU	U+79D2 (秒)	Seconds

For example, applying (FORMAT '99 時 99 分 99 秒') to 010203 yields 01 時 02 分 03 秒.

You can use Kanji characters with fractional seconds in TIME and TIMESTAMP formatting. For example:

```
CREATE TABLE t3
(tstart TIME(2) FORMAT 'HH 時 MI 分 SS.S(2)秒'
,tend TIME FORMAT 'HH 時 MI 分 SSDS(F 秒');
```

Time data must include the same Kanji time markers that appear in the FORMAT phrase. For example, the following CREATE TABLE statement uses the FORMAT phrase to define the display format for a TIMESTAMP column.

```
CREATE TABLE t2
(tstart TIMESTAMP FORMAT 'YY 年 MM 月 DD 日 HH 時 MI 分 SS 秒');
```

The following INSERT statement is valid, because the character string data includes the Kanji time markers specified in the FORMAT phrase.

```
INSERT t2 ('92 年 03 月 10 日 18 時 06 分 23 秒');
```

## Using FORMAT for Input

FORMAT can also be used for input.

For example, the following statement creates a table table\_2 with two date type columns:

```
CREATE Table_2
(f1 DATE FORMAT 'YY 年 MM 月 DD 日',
f2 DATE FORMAT '和暦 YY 年 MM 月 DD 日');
```



The following SQL statement is valid:

```
INSERT INTO Table_2 ('92 年 03 月 10 日', '平成 04 年 03 月 10 日');
```

Each Imperial Era (except the current one) has an end, which is considered in the validation of input. Using Japanese Imperial Era formatting for dates previous to the Meiji era (before 1867/1/9) is considered a run-time error.

Normally on input, Vantage assigns default values for the year (current year), month (current month), and day (first day), if not specified.

## Rules Applied for the First Year of an Era

For the first year of an era, the default month and day is affected by the starting date of the era. For other years, the standard default of the first month and first day are applied.

The following rules are applied for the first year of an era:

- If the day and month have not been indicated, the day is set to the first day of that era (for example, day 25 for Showa era) and the month is set to the first month of that era (for example, month 12 for Showa era).
- If the day is indicated but the month is not, the month is set to the first month of that era.

Note that this can produce an error.

For example, the date '平成 01 年 06 日' indicates the first year of the Heisei era and day 6. The month is defaulted to the first month of the Heisei era (January).

- If the month is indicated but the day is not:
  - If it is the first month of the era (for example, month 12 for the Showa era), then day is set to the first day of that era.
  - If it is not the first month of the era, the day is set to 1 (the beginning of the month).

## Naming Columns and Expressions

Each column in a SELECT result has a name that is derived from the list of specified column names (expressions) that generated the data. The name is typically the column name from which the data came.

For example, the columns in the result

```
SELECT EmpNo, Name
FROM Employee;
```

are labeled EmpNo and Name by default.

In some cases, it is necessary to associate a column name that is different from the default column name with a result. Vantage provides two ways to name a column.

This phrase for naming ...	Is ...
NAMED	a Teradata extension to the ANSI SQL:2011 standard.
AS	ANSI SQL:2011 compliant.

## Naming Columns

Vantage allows for naming derived columns and renaming existing columns. For every expression that is given a name, that name is entered in a NAMED list that associates the expression with the name. This is true for both forms of naming columns.

The AS and NAMED phrases are documented individually on subsequent pages.

## Rules for Columns Associated With an Explicit Table Name

If a column is associated with an explicit table name, the table is searched for the column name. Based on the result of the search, one of the following occurs:

IF this ...	IS ...	THEN ...
a matching column	found in the table	the requested information is returned.
	<i>not</i> found in the table	the NAMED list is searched for the column name. "Named" names are not valid when used to fully qualify a table name.
a column name	not found in the explicit table name or in the NAMED list	a nonexistent column error is reported.

## Rules for Columns Not Associated With an Explicit Table Name

If a column is not associated with an explicit table name, tables named in the SQL statement are searched, and one of the following occurs.

IF this ...	IS ...	THEN ...
a matching column	found in only one table	the requested information is returned.
	found in two or more tables	an ambiguous column name error is reported.
	<i>not</i> found	the NAMED list is searched for the column name. Named names are not valid when used to fully qualify a table name.
a column name	not found in the SQL statement or in the NAMED list	a nonexistent column error is reported.

## Qualifying Column Names and Named Objects

When you qualify a column with a table name, the qualifying column cannot be a named object. The following examples demonstrate why this does not work.

Begin with table t, defined as:

```
CREATE TABLE T (a INT, b INT);
```

Using this base table, note how the SQL resolver parses and resolves the following statement, creating a view based on the base table:

```
CREATE VIEW V AS
SELECT T.a (NAMED X), X (NAMED Y);
```

Name order and resolution are extremely important because of the way the resolver treats this statement.

Step	IF Column x is ...	THEN ...	ELSE ...
1	in table t	proceed	
2	not in table t	find column x in view v	display an appropriate error message

The following example explains why named objects can be tricky to use in statements such as this:

```
CREATE VIEW V AS
SELECT a*5+3 (NAMED X), x*2 (NAMED Y)
FROM T;
```

The Resolver analyzes this statement as follows:

1. Look for X as a column in table T.
2. If X is not found, then try to locate column X in view V (the named object).

Usage of named objects is important, as shown in the following example:

```
CREATE VIEW V AS
SELECT a (NAMED b), b (NAMED y)
FROM T;
```

Note that the phrase b (NAMED Y) resolves to T.b.

The following example does not work because the table named T does not have a column named X. X is a named object, not a column name. The erroneous example is:

```
CREATE VIEW v AS
SELECT t.a (NAMED X), t.x (NAMED Y);
```

Because table T does not have a column named X, the statement fails. X is a named object. The rule is that when you qualify a column with a table name, the qualifying column cannot be a named object.

## AS

The AS phrase assigns a name to an expression.

### ANSI Compliance

AS is ANSI SQL:2011 compliant.

## Syntax

```
value_expression [ AS ] name
```

### Syntax Elements

#### *value\_expression*

The expression to which a temporary name is to be assigned.

#### AS

An optional keyword indicating that the variable that follows is the temporary new name for *value\_expression*.

#### *name*

The new temporary name for *value\_expression*.

## Usage Notes

The temporary name can be referenced elsewhere in the request.

Column headings (that is, TITLE) default to the newly assigned name. See [TITLE](#).

## Examples

### Example: Using AS to Assign a Temporary Name to an Expression

This syntax allows the ORDER BY clause to include column names rather than positionally defining the sort columns. The AS keyword is not included when column\_2 is renamed to second\_value.

```
SELECT COLUMN1 +100 AS first_value, column_2 second_value
FROM table_1
WHERE second_value > 100
ORDER BY first_value;
```

## Example: Creating a Temporary Named Column

This syntax creates a temporary named column and uses it in a WHERE clause.

```
SELECT Name,((Salary + (YrsExp * 200))/12) AS Projection
FROM Employee
WHERE DeptNo = 600
AND Projection < 2500;
```

## NAMED

The NAMED phrase assigns a name to an expression.

### ANSI Compliance

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

For ANSI compliance, use [AS](#) instead of NAMED.

## Syntax

```
( expression ) ( NAMED name )
```

### Syntax Elements

#### *expression*

The expression to which a temporary name is to be assigned.

#### NAMED

A keyword indicating that the variable that follows is the temporary new name for *expression*.

#### *name*

The new temporary name for *expression*.

## Usage Notes

The temporary name can be referenced elsewhere in the request. But it should not be the same as another column name of any table used in a query.

Column headings (that is, TITLE) default to the newly assigned name. See [TITLE](#).

## Differences Between NAMED and AS

The keyword NAMED is required in Teradata syntax. The keyword AS is optional.

The NAMED clause is enclosed in parentheses immediately following the renamed column.

AS is the ANSI SQL:2011 syntax. Teradata syntax using the NAMED clause is supported for backward compatibility. Use AS in place of NAMED for ANSI compliance. See [AS](#).

## Example: Using a NAMED Phrase

In the following SELECT statement, a NAMED phrase associates an arithmetic expression with the name “Projection” so that the calculated column can be more easily referenced in the WHERE clause and the heading (TITLE) in the report is more meaningful.

```
SELECT Name, ((Salary + (YrsExp * 200))/12) (NAMED Projection)
FROM Employee
WHERE DeptNo = 600 AND Projection < 2500;
```

The ANSI syntax for this example is the same as used for Example 2 for the [AS](#).

## TITLE

The TITLE phrase of a CREATE TABLE, ALTER TABLE, or SELECT statement gives a name to a column heading.

## ANSI Compliance

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

## Syntax

```
TITLE char_string
```

## Syntax Elements

### *char\_string*

A character string literal that defines a column name heading.

## Usage Notes

The maximum length for the column name heading is 256 Unicode characters.

A TITLE phrase submitted in BTEQ can be up to three lines.

A double slash character (//) defines a line break. You can insert blanks to center lines.

You can use the TITLE phrase in a CREATE TABLE or ALTER TABLE statement to specify a standard heading.

## Determining the Title for a Column or Expression

You can use the TITLE function to get the title for a specified expression or column. For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## TITLE Phrase Rules

Teradata SQL uses the following rules to generate a title for expression x:

IF x is ...	THEN ...
a column reference with an explicit TITLE phrase	<p>that title value is returned.</p> <p>Here is an example:</p> <p>The explicit title for Project.ProjId, as defined in the CREATE TABLE statement for the Project table, is Project//Id.</p> <p>The following statement returns the indicated result.</p> <pre> SELECT Proj_Id FROM Project WHERE Description='O/E Batch System'; Project       Id ----- OE2-0003 </pre>
a column reference with no explicit TITLE	<p>the column name is returned.</p> <p>Here is an example:</p> <p>The Salary column has no explicit TITLE.</p> <p>The following statement returns the indicated result.</p>

IF x is ...	THEN ...
	<pre> SELECT Salary FROM Employee WHERE EmpNo = 10002; Salary ----- 35,000.00 </pre>
a constant	<p>TITLE is a character representation of that constant. The character representation of the constant could be the identical characters or a normalized form of the constant.</p> <p>Here is an example:</p> <p>The following statement returns 12 as the column heading for the value 12.</p> <pre> SELECT Name, 12 FROM Employee WHERE EmpNo = 10003; Name      12 ----- -- Leidner   12 </pre>
of the form “operator y”, where operator is a unary (+ or -) or aggregate operator	<p>TITLE is ‘operator’ followed by ‘(y)’</p> <p>Here is an example:</p> <p>The following statement returns the aggregate operator name (SUM) as part of the column title.</p> <pre> SELECT SUM(Salary) FROM Employee WHERE DeptNo = 700; Sum(Salary) ----- 113,000.00 </pre>
of the form “y operator z”	<p>y followed by “operator” followed by z.</p> <p>Here is an example:</p> <p>The following statement returns the headings of the column data for which the computation is performed and the operator.</p> <pre> SELECT Salary_Loan - 1000 FROM Employee_Loan WHERE EmpNo = 10004; Salary_Loan - 1000 ----- 41000.00 </pre>
an attribute function of the form “function (y)”	<p>TITLE is function, followed by (y).</p> <p>Here is an example:</p> <p>The following statement returns the indicated result.</p>



IF x is ...	THEN ...
	<pre> SELECT FORMAT (Employee.EmpNo); Format(EmpNo) ----- ZZZZ9 </pre>
of the form “y (data_description)”, where data_description does <i>not</i> contain a TITLE phrase or a NAMED phrase	<p>TITLE is (y). Here is an example: The following statement returns the indicated result.</p> <pre> SELECT YrsExp (BYTEINT) FROM Employee WHERE EmpNo = 10016; YrsExp -----       20 </pre>
of the form “y (data_description)”, where data_description does not contain a TITLE phrase, but does contain a NAMED phrase	<p>TITLE is the name that is specified in the NAMED phrase. Here is an example: The following statement returns the indicated result.</p> <pre> SELECT YrsExp (BYTEINT, NAMED YearsOfExperience) FROM Employee WHERE EmpNo = 10016; YearsOfExperience -----               20 </pre>
of the form “y (data_description)”, where data_description contains a TITLE phrase	<p>TITLE is the title that is specified by the phrase. Here is an example: The following statement returns the indicated result.</p> <pre> SELECT Salary (INTEGER, TITLE 'Pay') FROM Employee WHERE EmpNo = 10018; Pay ----- 65000 </pre>

The number of dashes used to define column width is an attribute of BTEQ.

You can modify this display by including a FORMAT phrase in the SELECT statement.

## Examples

### Example: Using the TITLE Phrase

The following example shows a simple use of the TITLE phrase.

```
CREATE TABLE Charges,
FALLBACK
(EmpNo SMALLINT FORMAT '9(5)' TITLE 'Employee//Id' BETWEEN 10001 AND 32001
NOT NULL,
Proj_Id CHAR(8) TITLE 'Project// Id' NOT NULL,
WkEnd DATE FORMAT 'YYYY/MM/DD' TITLE 'Week//Ending',
Hours DECIMAL(4,1) FORMAT 'ZZ9.9' BETWEEN 0.5 AND 999.5)
PRIMARY INDEX (EmpNo, Proj_Id)
INDEX (Proj_Id);
```

### Example: Using the TITLE Phrase in a SELECT Statement

When a title is specified at column creation time, note that the defined column name, not the title name, must be specified in statements that access the column. For example, to retrieve the total hours worked on each project by user Peterson, the project column must be referenced:

```
SELECT Proj_Id, SUM(Hours)
FROM CHARGES
WHERE EmpNo = 10001
GROUP BY Proj_Id ORDER BY Proj_Id ;
```

The returned report, however, uses the following title headings:

Project Id	Sum(Hours)
-----	-----
PAY-0001	9.5
PAY-0002	34.5

A TITLE phrase also can be given in the retrieval statement to override a default title.

For example, in the following SELECT statement, a TITLE phrase provides a more descriptive heading for the DOB column.

```
SELECT Name, DOB (TITLE 'Birthdate')
FROM Employee;
```

This statement returns:

Name	Birthdate
-----	-----
Smith T	51/10/31
Newman P	56/08/29
Omura H	54/04/24
.	.
.	.

The multiline TITLE definition in this statement:

```
SELECT Name, DOB (TITLE 'Date//Of//Birth')
FROM Employee;
```

returns:

Name	Date Of Birth
-----	-----
Smith T	51/10/31
Newman P	56/08/29
Omura H	54/04/24
.	.
.	.

# Data Type Conversions

## Data Type Conversions

The following sections describe the SQL CAST function and the rules for converting data from one type to another, both explicitly and implicitly.

A data type conversion modifies the data definition (data type, data attributes, or both) of an expression and can be either implicit or explicit. Explicit conversions can be made using the CAST function or Teradata conversion syntax.

## Forms of Data Type Conversions

Vantage supports the following forms of data conversion:

- Implicit
- Explicit using the CAST function
- Explicit using Teradata conversion syntax

## Implicit Type Conversions

Vantage permits the assignment and comparison of some types without requiring the types to be explicitly converted. SQL Engine also performs implicit type conversions in the following cases:

- On some argument types passed to macros, stored procedures, and SQL functions such as SQRT.
- On the expression that defines a time zone displacement in an AT clause.

## Examples

### Example: Implicit Type Conversion During Assignment

Consider the following tables:

```
CREATE TABLE T1
  (Fname VARCHAR(25)
  ,Fid    INTEGER
  ,Yrs    CHARACTER(2));
CREATE TABLE T2
  (Wname VARCHAR(25))
```

```
,Wid    INTEGER
,Age    SMALLINT);
```

In the following statement, Vantage implicitly converts the character string in T1.Yrs to a numeric value:

```
UPDATE T2 SET Age = T1.Yrs + 5;
```

This is not evident in the syntax of the source statement, but becomes evident when the dictionary information for tables T1 and T2 is accessed.

## Example: Implicit Type Conversion During Comparison

Consider the table T1 in [Example: Implicit Type Conversion During Assignment](#).

In the following statement, Vantage implicitly converts both operands of the comparison operation to FLOAT values before performing the comparison:

```
SELECT Fname, Fid
FROM T1
WHERE T1.Yrs < 55;
```

## Example: Implicit Type Conversion in Parameter Passing Operations

Consider the SQRT system function that computes the square root of an argument.

In the following statement, Vantage implicitly converts the character argument to a FLOAT type:

```
SELECT SQRT('13147688');
```

## Supported Data Types

SQL Engine performs implicit conversion on the following types:

From Byte to:

- Byte

Byte types include BYTE, VARBYTE, and BLOB.

- UDT

The UDT must have an implicit cast that casts the predefined type to a UDT. To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

From Numeric to:

- Numeric
- DATE
- Character
- UDT

The UDT must have an implicit cast that casts the predefined type to a UDT. To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

From DATE to:

- Numeric
- DATE
- Character
- UDT

The UDT must have an implicit cast that casts the predefined type to a UDT. To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

From Character to:

- Numeric
- DATE
- Character

Character types include CHAR, VARCHAR, and CLOB.

- Period
- TIME
- TIMESTAMP
- UDT

The UDT must have an implicit cast that casts the predefined type to a UDT. To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

From TIME to:

- UDT

The UDT must have an implicit cast that casts the predefined type to a UDT. To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

From TIMESTAMP to:

- UDT

The UDT must have an implicit cast that casts the predefined type to a UDT. To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

From INTERVAL to:

- UDT

The UDT must have an implicit cast that casts the predefined type to a UDT. To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

From UDT to:

- Predefined data types that are the target of implicit casts defined for the UDT

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

- Other UDTs that are the target of implicit casts defined for the UDT

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Implicit Conversion of DateTime Types

SQL Engine performs implicit conversion on DateTime data types in the following cases:

- When passing data using dynamic parameter markers, or the question mark (?) placeholder.
- With INSERT, INSERT ... SELECT, and UPDATE statements.
- With MERGE INTO statements.
- When handling default values for the CREATE TABLE and ALTER TABLE statements. For more information, see [DEFAULT Phrase](#).
- During stored procedure execution, including the execution of the following statements: DECLARE, SELECT ... INTO, and SET. See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148.

Implicit conversion is dependent on client-side support. For information about the client products which support implicit conversion of DateTime types, see the Teradata Tools and Utilities user documentation.

The following conversions are supported.

FROM...	TO...	For further details, see...
DATE	TIMESTAMP	<a href="#">Implicit DATE-to-TIMESTAMP Conversion</a> .

FROM...	TO...	For further details, see...
TIME	TIMESTAMP	<a href="#">Implicit TIME-to-TIMESTAMP Conversion.</a>
TIMESTAMP	DATE	<a href="#">Implicit TIMESTAMP-to-DATE Conversion.</a>
TIMESTAMP	TIME	<a href="#">Implicit TIMESTAMP-to-TIME Conversion.</a>
INTERVAL	INTERVAL	<a href="#">Implicit INTERVAL-to-INTERVAL Conversion.</a>

SQL Engine performs implicit conversion on DateTime data types during assignment in the following cases:

FROM...	TO...	For further details, see...
DATE	TIMESTAMP	<a href="#">Implicit DATE-to-TIMESTAMP Conversion.</a>
TIME	TIMESTAMP	<a href="#">Implicit TIME-to-TIMESTAMP Conversion.</a>
TIMESTAMP	DATE	<a href="#">Implicit TIMESTAMP-to-DATE Conversion.</a>
TIMESTAMP	TIME	<a href="#">Implicit TIMESTAMP-to-TIME Conversion.</a>
Interval	Exact Numeric	The INTERVAL type must have only one field, e.g., INTERVAL YEAR. <a href="#">Implicit INTERVAL-to-Numeric Conversion.</a>
Exact Numeric	Interval	The INTERVAL type must have only one field, e.g., INTERVAL YEAR. <a href="#">Implicit Numeric-to-INTERVAL Conversion.</a>

**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. See *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211.

SQL Engine performs implicit conversion on DateTime data types in single table predicates and join predicates in the following cases:

FROM...	TO...	For further details, see...
TIMESTAMP	DATE	<a href="#">Implicit TIMESTAMP-to-DATE Conversion.</a>
Interval	Exact Numeric	The INTERVAL type must have only one field, e.g., INTERVAL YEAR. <a href="#">Implicit INTERVAL-to-Numeric Conversion.</a>
Exact Numeric	Interval	<a href="#">Implicit Numeric-to-INTERVAL Conversion.</a>

For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

The following are not supported:



- Implicit conversion from TIME to TIMESTAMP and from TIMESTAMP to TIME are not supported in comparisons.
- Implicit conversion of DateTime types in set operations.

## Implicit Conversion Rules

Teradata SQL performs implicit type conversions on expressions before any operation is performed.

## Truncation During Conversion

In some cases, implicit conversion can result in truncation of values without an error.

Recommendation: As a best practice, use an explicit CAST instead of relying on implicit conversions when possible.

### Example: Converting to CHAR Using Teradata Conversion Syntax

Consider the following table definition:

```
CREATE TABLE Test1 (c1 INT, c2 VARCHAR(1));
```

The following two INSERT statements complete without any errors.

```
INSERT INTO Test1 VALUES (1, '1');
INSERT INTO Test1 VALUES (2, 2);
```

The following query returns two rows.

```
SELECT * FROM Test1;
  c1      c2
-----
   1      1
   2      <<<< Note that the value inserted in c2 is a blank
```

In the second INSERT statement, the number 2 was implicitly converted to CHAR using Teradata conversion syntax (that is, not using CAST). The process is as follows:

1. Convert the numeric value to a character string using the default or specified FORMAT for the numeric value. Leading and trailing pad characters are not trimmed.
2. Extend to the right with pad characters if required, or truncate from the right if required, to conform to the target length specification.

If non-pad characters are truncated, no string truncation error is reported.

The conversion right-justifies the number, but takes the first byte of the result which is a single blank character. For more information about numeric to character conversions, see [Numeric-to-Character Conversion](#).

## Restrictions

SQL Engine does not perform implicit conversion on input arguments to UDFs, UDMs, or external stored procedures (external routines). Arguments do not necessarily have to be exact matches to the parameter types, but they must be compatible. For example, you can pass a SMALLINT argument to an external routine that expects an INTEGER argument because SMALLINT and INTEGER are compatible. To pass a DATE type argument to an external routine that expects an INTEGER argument, you must explicitly cast the DATE type to an INTEGER type. For more information, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Some SQL functions and operators require arguments that are exact matches to the parameter types. For details, refer to the documentation for the specific function or operator.

## Related Information

- [Byte-to-Byte Conversion](#)
- [Numeric-to-Numeric Conversion](#)
- [Numeric-to-DATE Conversion](#)
- [Numeric-to-Character Conversion](#)
- [Numeric-to-UDT Conversion](#)
- [DATE-to-Numeric Conversion](#)
- [DATE-to-DATE Conversion](#)
- [DATE-to-Character Conversion](#)
- [DATE-to-UDT Conversion](#)
- [Character-to-Numeric Conversion](#)
- [Character-to-DATE Conversion](#)
- [Character-to-Character Conversion](#)
- [Character-to-Period Conversion](#)
- [Character-to-TIME Conversion](#)
- [Character-to-TIMESTAMP Conversion](#)
- [Character-to-UDT Conversion](#)
- [TIME-to-UDT Conversion](#)
- [TIMESTAMP-to-UDT Conversion](#)
- [INTERVAL-to-UDT Conversion](#)
- [UDT-to-Character Conversion](#)
- [UDT-to-DATE Conversion](#)
- [UDT-to-INTERVAL Conversion](#)
- [UDT-to-Numeric Conversion](#)
- [UDT-to-TIME Conversion](#)
- [UDT-to-TIMESTAMP Conversion](#)

- [UDT-to-UDT Conversion](#)
- For details on implicit type conversion of operands for comparison operations, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## CAST in Explicit Data Type Conversions

Converts an expression of a given data type to a different data type or the same data type with different attributes.

Teradata SQL supports two different syntaxes for CAST functionality, only one of which is ANSI SQL:2011 compliant.

### ANSI Compliance

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

### 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.

## Usage Notes

The ANSI SQL:2011 compliant form can be used to convert data types in either ANSI-compliant SQL statements or Teradata SQL statements.

The Teradata extended syntax is more general. It allows a type declaration or data attributes or both.

Avoid using the extended form of CAST for any application intended to be ANSI-compliant and portable.

CAST functions identically in both ANSI and Teradata modes.

When converting DateTime data types, you can use the AT clause to specify the time zone used for the CAST. You can specify the source time zone, a specific time zone displacement, or the current session

time zone. For more information, see the section on converting the specific data type, for example, [TIMESTAMP-to-DATE Conversion](#).

CAST does not convert the following data type pairs:

- Numeric to character, if the server character set is GRAPHIC.
- Character expressions having different server character sets.

To make such a conversion, use the TRANSLATE function. For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

- Byte (BYTE, VARBYTE, and BLOB) to any data type other than UDT or byte, and data types other than byte or UDT to byte.
- CLOB to any data type other than UDT or character, and data types other than character or UDT to CLOB.

For information on casting to and from geospatial types, see *Teradata Vantage™ - Geospatial Data Types*, B035-1181.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement.

## Character Truncation Rules

The following rules apply to character strings.

IF the string is cast in this mode ...	THEN it is truncated of ...
ANSI	trailing pad character spaces to achieve the desired length. Truncation of other characters, or part of a multibyte character, returns an error.
Teradata	trailing characters to achieve the desired length. Truncation on Kanji1 character data types containing multibyte characters might result in truncating one byte of the multibyte character.

## Server Character Set Rules

When *data\_definition\_list* specifies a data type of CHARACTER (CHAR) or CHARACTER VARYING (VARCHAR) and does not specify a CHARACTER SET clause to indicate which server character set to use, then the resulting server character set is as follows.

IF the data type of <i>expression</i> is ...	THEN the server character set of the resulting characters is ...
non-character	the user default server character set.
character	the server character set of <i>expression</i> .

## Numeric Overflow, Field Mode, and CAST

Numeric overflows are handled differently depending on whether you are running ANSI or Teradata mode, and whether you are running in Field Mode or not.

Field Mode is not ANSI SQL:2011 compatible. In Field Mode, conversion to a numeric or decimal data type that results in a numeric overflow is returned as asterisks ('\*\*\*') rather than an error message.

Record and Indicator Modes do not behave in this manner and return an error message.

## Examples

The following examples illustrate how to perform data type conversions using CAST.

### Example: Performing Data Type Conversions

Using ANSI CAST syntax:

```
SELECT ID_Col, Name_Col
FROM T1
WHERE Int_Col = CAST(SUBSTRING(Char_Col FROM 3 FOR 3) AS INTEGER);
```

### Example: Performing Data Type Conversions Using ANSI CAST Syntax

Using ANSI CAST syntax:

```
SELECT CAST(SUBSTRING(Char_Col FROM 1 FOR 2) AS INTEGER),
       CAST(SUBSTRING(Char_Col FROM 3 FOR 3) AS INTEGER)
FROM T1;
```

### Example: Performing Data Type Conversions Using Teradata Extensions to the ANSI CAST Syntax

Using Teradata extensions to the ANSI CAST syntax:

```
CREATE TABLE t2 (f1 TIME(0) FORMAT 'HHhMI'm');

INSERT t2 (CAST('15h33m' AS TIME(0) FORMAT 'HHhMI'm));

SELECT f1 FROM t2;
```

The result from the SELECT statement is:

```
f1
-----
15h33m
```

## Related Information

For further rules that apply to the conversion between specific data types, for example, numeric-to numeric or character-to-numeric, see the appropriate succeeding topic.

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

## Data Conversions in Field Mode

### Field Mode: User Response Data

In Field Mode, a report format used in BTEQ, all data is returned in character form. The alignment and spacing of columns is controlled by data formats and title information. Each row returned is essentially a character string ready for display.

In Field Mode, it is unnecessary to explicitly convert numeric data to character format.

### Conversions to Numeric Types

When in Field Mode, a numeric overflow returned for character to numeric data type conversion is not treated as an error. If the result exceeds the number of digits normally reserved for the numeric data type, the result appears as a set of asterisks in the report.

For example, the character to SMALLINT conversion in the following statement results in numeric overflow because the number of digits normally reserved for a SMALLINT is five:

```
SELECT '100000' (SMALLINT);
```

The result is:

```
'100000'
-----
*****
```

Additionally, when in Field Mode, asterisks appear in the report for conversions to numeric types involving results that do not fit the specified output format.

For example, the DATE to INTEGER conversion in the following statement results in a value that does not fit the format specified by the FORMAT phrase:

```
SELECT CAST (CURRENT_DATE as integer format '9999');
```

The result is:

```
Date
----
****
```

The same query executed in Record or Indicator Variable Mode reports an error.

## Byte-to-Byte Conversion

You can convert a byte expression to a different byte type with either the CAST statement or Teradata conversion syntax.

### Byte-to-Byte Conversion with CAST

#### ANSI Compliance

CAST is ANSI SQL:2011 compliant, provided the syntax does not specify data attributes.

#### Syntax

```
CAST (byte_expression AS
{
  { byte_data_type | UDT_data_type } [ data_attribute [...] ] |
  data_attribute [...]
}
)
```

#### Syntax Elements

##### *byte\_expression*

An expression in byte format to be cast to a different data definition.

##### *byte\_data\_type*

The new byte type to which *byte\_expression* is to be converted.

##### *UDT\_data\_type*

A UDT that has a cast definition that casts the byte type to the UDT.

To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*.

#### ***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## **Teradata Byte-to-Byte Conversion Syntax**

### **ANSI Compliance**

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

### **Syntax**

```
byte_expression (
  {
    byte_data_type [, data_attribute [,...]] |
    data_attribute [,...]] [, byte_data_type [, data_attribute [,...]] ] ]
  }
```

### **Syntax Elements**

#### ***byte\_expression***

An expression in byte format to be cast to a different data definition.

#### ***byte\_data\_type***

The new byte type to which *byte\_expression* is to be converted.

#### ***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE



## Usage Notes

### Conversions Where Source and Target Types Differ in Length

If the length specified by *byte\_data\_type* is less than the length of *byte\_expression*, bytes beyond the specified length are truncated. No error is reported.

If *byte\_data\_type* is fixed-length and the length is greater than that of *byte\_expression*, bytes of value binary zero are appended as required.

### Supported Source and Target Data Types

Vantage supports byte data type conversions according to the following table.

Source Data Type	Target Data Type	Allowable Conversions
BYTE	<ul style="list-style-type: none"> <li>• BYTE</li> <li>• VARBYTE</li> <li>• BLOB</li> </ul>	<ul style="list-style-type: none"> <li>• Implicit</li> <li>• Explicit using CAST and Teradata conversion syntax</li> </ul>
VARBYTE		
BLOB		
BYTE	UDT	<ul style="list-style-type: none"> <li>• Implicit</li> <li>• Explicit using CAST</li> </ul>
VARBYTE		
BLOB		
UDT	<ul style="list-style-type: none"> <li>• BYTE</li> <li>• VARBYTE</li> <li>• BLOB</li> </ul>	<ul style="list-style-type: none"> <li>• Implicit</li> <li>• Explicit using CAST and Teradata conversion syntax</li> </ul> <p>Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</p>

### Rules for Implicit Byte-to-UDT Conversions

SQL Engine performs implicit Byte-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit Byte-to-UDT data type conversion requires a cast definition that specifies the following:

- the AS ASSIGNMENT clause
- a BYTE, VARBYTE, or BLOB source data type

The source data type of the cast definition does not have to be an exact match to the source of the implicit type conversion.

If multiple implicit cast definitions exist for converting different byte types to the UDT, SQL Engine uses the implicit cast definition for the byte type with the highest precedence. The following list shows the precedence of byte types in order from lowest to highest precedence:

- BYTE
- VARBYTE
- BLOB

## Using HASHBUCKET to Convert a BYTE Type to an INTEGER Type

You can use the HASHBUCKET function to convert a BYTE(1) or BYTE(2) type to an INTEGER type.

## Examples

### Example: Explicit Conversion of BLOB to VARBYTE

Consider the following table definition:

```
CREATE TABLE large_images
(id INTEGER
,image BLOB);
```

The following statement casts the BLOB column to a VARBYTE type, and uses the result as an argument to the POSITION function:

```
SELECT POSITION('FFF1'xb IN (CAST(image AS VARBYTE(64000))))
FROM large_images
WHERE id = 5;
```

### Example: Implicit Conversion of VARBYTE to BLOB

Consider the following table definitions:

```
CREATE TABLE small_images
(id INTEGER
,image1 VARBYTE(30000)
,image2 VARBYTE(30000));

CREATE TABLE large_images
(id INTEGER
,image BLOB);
```

SQL Engine performs a VARBYTE to BLOB implicit conversion for the following INSERT statement:

```
INSERT large_images
SELECT id, image1 || image2
FROM small_images;
```

## Character-to-Character Conversion

You can shorten or expand output character strings with either the CAST statement or Teradata conversion syntax.

## Character-to-Character Conversion with CAST

### ANSI Compliance

CAST is ANSI SQL:2011 compliant, provided the syntax does not specify any data attributes.

### Syntax

```
CAST ( character_expression AS
{
  character_data_type [ data_attribute [...] ] |
  data_attribute [...]
}
)
```

### Syntax Elements

#### *character\_expression*

A character expression to be converted.

***character\_data\_type***

The data type to which the expression is to be converted.

***data\_attribute***

One of the following data attributes:

- CHARACTER SET
- FORMAT
- NAMED
- TITLE

## Teradata Character-to-Character Conversion

### ANSI Compliance

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

### Syntax

```
character_expression (
  {
    character_data_type [, data_attribute [,...]] |
    data_attribute [,...]] [, character_data_type [, data_attribute [,...]] ] ]
  }
)
```

### Syntax Elements

***character\_expression***

A character expression to be converted.

***character\_data\_type***

The data type to which the expression is to be converted.

***data\_attribute***

One of the following data attributes:

- CHARACTER SET
- FORMAT
- NAMED

- TITLE

## Usage Notes

### General Usage Notes

If the source string (CHAR, VARCHAR, or CLOB) is longer than the target data type (CHAR, VARCHAR, or CLOB), excess characters are truncated.

Session Mode for INSERT or UPDATE	Non-pad Characters Truncated to Store Character Values in a Table
ANSI	Error is reported.
Teradata	Error is not reported.

Pad characters are trimmed or appended, according to the following rules:

Source String Data Type	Length	Target Data Type	Pad Characters
CHAR	Longer than the target	CLOB or VARCHAR	Any trailing pad characters are trimmed.
CHAR, VARCHAR, or CLOB	Shorter than the target	CHAR	Trailing pad characters are appended to the target.
CHAR	All pad characters	CLOB or VARCHAR	Field is truncated to zero length.

### Example: Character-to-Character Conversions

Following are examples of character-to-character conversions.

Character String	String Length	Character Description	Conversion Result	Converted Length
'HELLO'	5	CHAR(3)	'HEL', if session is in Teradata mode	3
			Error, if session is in ANSI mode	
'HELLO'	5	CHAR(7)	'HELLO '	7
'HELLO'	5	VARCHAR(7)	'HELLO'	5
'HELLO '	7	VARCHAR(6)	'HELLO '	6

Character String	String Length	Character Description	Conversion Result	Converted Length
'HELLO '	7	VARCHAR(3)	'HEL', if session is in Teradata mode	3
			Error, if session is in ANSI mode	

## CAST Syntax Usage Notes

The server character set of *character\_expression* must have the same server character set as the target data type.

If CAST is used to convert data to a character string and non-pad characters would be truncated, an error is reported.

## Teradata Conversion Syntax Usage Notes

The server character set of *character\_expression* can be changed to a different server character set specified as *data\_attribute*, where *data\_attribute* is the CHARACTER SET phrase.

## Implicit Character-to-Character Conversion

CLOB types can only be converted to or from CHAR or VARCHAR types. For example, implicit conversion is performed on CLOB data that is inserted into a CHAR or VARCHAR column. See [Implicit Type Conversions](#).

Comparisons of strings, fixed- or variable-length, require operands of equal length. The following table shows that the shorter string is converted by being padded on the right.

Expression	Conversion	Result
'x'='x '	'xΔ'='x '	TRUE
'x'='xx'	'xΔ'='xx'	FALSE

where Δ is a pad character.

If a character is not in the repertoire of the target character set, an error is reported.

## Related Information

For information on using the TRANSLATE function to perform a translation, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Implicit Character-to-Character Translation

Implicit string translation occurs when two character strings are incompatible within a given operation. For example,

```
SELECT *
FROM string_table
WHERE clatin < csjis;
```

where clatin represents a character column defined as CHARACTER SET LATIN and csjis represents a character column defined as CHARACTER SET KANJISJIS.

If an implicit translation of character string '*string*' to a UNICODE character string is required, it is equivalent to executing the TRANSLATE (*string* USING *source\_repertoire\_name*\_TO\_Unicode ) function, where *source-repertoire-name* is the server character set of *string*.

More specifically, if as in the above example, *string* is of KANJISJIS type, then the translation is equivalent to executing the TRANSLATE (*string* USING KanjiSJIS\_TO\_Unicode) function.

## ANSI Compliance

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

## Character Literals

The following rules apply to implicit character-to-character translation involving character literals.

IF one operand is a ...	AND the other operand is a ...	THEN ...
literal	literal	both operands are translated to UNICODE.
	non-literal	the literal is translated to the type of the non-literal. If that fails, both are translated to UNICODE.
	literal expression	the literal is translated to the type of the literal expression. If that fails, both are translated to UNICODE.
literal expression	literal expression	both operands are translated to UNICODE.
	non-literal	the literal expression is translated to the type of the non-literal. If that fails, both are translated to UNICODE.
non-literal	non-literal	both operands are translated to UNICODE.

## KANJISJIS Server Character Set

Implicit character-to-character translation always converts a character string argument that has the KANJISJIS server character set to UNICODE.

## SQL Rules for Implicit Translation for Expression and Function Arguments

The following are the rules for implicit translation between types of expressions and function arguments.

For string functions that produce a character result, the results are summarized by the following table.

FOR this function ...	The result is ...
TRIM	converted back to the type of the main string argument (last argument).
(concatenation)	not translated and remains with the character data type of the arguments after any implicit translation.

Note that the other string functions either do not involve conversion or the type of the result is based on the function and not the server character set of the argument.

For example, in the following TRIM function, <unicode-literal> is first translated to Latin, and then the trim operation is performed.

```
...
TRIM(<unicode-literal> FROM <latin-value>)
```

The result is Latin.

## Character-to-DATE Conversion

You can convert a character string to a DATE value with either the CAST statement or Teradata conversion syntax.

## Character-to-DATE Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative formatting for the date data.



**Syntax**

```
CAST (
  character_expression AS DATE
  [ data_attribute [...] ]
)
```

**Syntax Elements*****character\_expression***

A character expression to be converted.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

**Teradata Character-to-DATE Conversion Syntax****ANSI Compliance**

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

**Syntax**

```
character_expression ( [ data_attribute [,...] ] DATE [, data_attribute [,...] ] )
```

**Syntax Elements*****character\_expression***

A character expression to be converted.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Usage Notes

The character expression is trimmed of leading and trailing pad characters and handled as if it was a string literal in the declaration of a DATE literal.

Character-to-DATE conversion is supported for CHAR and VARCHAR types only. The source character type cannot be CLOB.

If the string can be converted to a valid DATE, then it is. Otherwise, an error is returned.

## Character String Format

If the dateform of the current session is INTEGERDATE, the date representation in the character string must match the DATE output format according to the rules in the following table.

IF the statement ...	THEN ...
specifies a FORMAT phrase for the DATE	the character string must match that DATE format.
does not specify a FORMAT phrase	<p>If the DATE column definition:</p> <ul style="list-style-type: none"> <li>Specifies a FORMAT phrase, the character string must match the DATE format.</li> <li>Does not specify a FORMAT phrase, the character string must match 'YY/MM/DD', or the current setting of the default date format in the specification for data formatting (SDF) file.</li> </ul>

For an example, see [Example: IntegerDate Dateform Mode](#).

If the dateform of the current session is ANSI DATE, the date representation in the character string must match the DATE output format according to the rules in the following table.

IF the statement ...	THEN ...
specifies a FORMAT phrase for the DATE	the character string must match that DATE format.
does not specify a FORMAT phrase	<p>If in field mode, then if the DATE column definition:</p> <ul style="list-style-type: none"> <li>Specifies a FORMAT phrase, the character string must match that DATA format.</li> <li>Does not specify a FORMAT phrase, the character string must match the ANSI format ('YYYY-MM-DD')</li> </ul> <p>If in record or indicator mode, the character string must match the ANSI format ('YYYY-MM-DD').</p>

# Forcing a FORMAT on CAST for Converting Character to DATE

You can use a FORMAT phrase to convert a character string that does not match the format of the target DATE data type. A character string in a conversion that does not specify a FORMAT phrase uses the output format for the DATE data type.

For example, suppose the session dateform is INTEGERDATE and the default DATE format of the system is set to 'yyyymmdd' through the tdlocaledef utility. The following statement fails, because the character string contains separators, which does not match the default DATE format:

```
SELECT CAST ('2005-01-01' AS DATE);
```

To override the default DATE format, and convert a character string that contains separators, specify a FORMAT phrase for the DATE target type:

```
SELECT CAST ('2005-01-01' AS DATE FORMAT 'YYYY-MM-DD');
```

In character-to-DATE conversions, the FORMAT phrase must not consist solely of the following formatting characters.

<ul style="list-style-type: none"> <li>• EEEE</li> <li>• E4</li> </ul>	<ul style="list-style-type: none"> <li>• EEE</li> <li>• E3</li> </ul>
--	---

# Character Strings That Omit Day, Month, or Year

If the character string and the format for a character-to-DATE conversion omits the day, month, or year, the system uses default values for the target DATE value.

IF the character string omits the ...	THEN the system uses the ...
day	value of 1 (the first day of the month).
month	value of 1 (the month of January).
year	current year (at the current session time zone).

Consider the following table:

```
CREATE TABLE date_log
(id INTEGER
, start_date DATE
, end_date DATE
, log_date DATE);
```

The following INSERT statement converts three character strings to DATE values. The first character string omits the day, the second character string omits the month, and the third character string omits the year. Assume the current year is 1992.

```
INSERT date_log
  (1001
   ,CAST ('January 1992' AS DATE FORMAT 'MMMMBYYYY')
   ,CAST ('1992-01' AS DATE FORMAT 'YYYY-DD')
   ,CAST ('01/01' AS DATE FORMAT 'MM/DD'));
```

The result of the INSERT statement is as follows:

```
SELECT * FROM date_log;
      id  start_date  end_date  log_date
-----
      1001    92/01/01  92/01/01  92/01/01
```

## Implicit Character-to-DATE Conversion

If the string does not represent a valid date, an error is reported.

In record or indicator mode, when the DateForm mode of the session is set to ANSIDate, the string must use the ANSI DATE format.

## Examples

### Example: IntegerDate Dateform Mode

For example, suppose the session dateform is INTEGERDATE, and the default DATE format of the system is set to 'yyyymmdd' through the tdlocaledef utility.

Consider the following table, where the start\_date column uses the default DATE format and the end\_date column uses the format 'YYYY/MM/DD':

```
CREATE TABLE date_log
  (id INTEGER
   ,start_date DATE
   ,end_date DATE FORMAT 'YYYY/MM/DD');
```

The following INSERT statement works because the character strings match the formats of the corresponding DATE columns and SQL Engine can successfully perform implicit character-to-DATE conversion:

```
INSERT INTO date_log (1099, '20030122', '2003/01/23');
```

To perform character-to-DATE conversion on character strings that do not match the formats of the corresponding DATE columns, you must use a FORMAT phrase:

```
INSERT INTO date_log
(1047
,CAST ('Jan 12, 2003' AS DATE FORMAT 'MMMBDD,BYYYY')
,CAST ('Jan 13, 2003' AS DATE FORMAT 'MMMBDD,BYYYY'));
```

## Example: ANSIDate Dateform Mode

Suppose the session dateform is ANSIDATE. The default DATE format of the system is 'YYYY-MM-DD'.

Consider the following table, where the start\_date column uses the default DATE format and the end\_date column uses the format 'YYYY/MM/DD':

```
CREATE TABLE date_log
(id INTEGER
,start_date DATE
,end_date DATE FORMAT 'YYYY/MM/DD');
```

The following INSERT statement works because the character strings match the formats of the corresponding DATE columns and SQL Engine can successfully perform implicit character-to-DATE conversion:

```
INSERT INTO date_log (1099, '2003-01-22', '2003/01/23');
```

To perform character-to-DATE conversion on character strings that do not match the formats of the corresponding DATE columns, you must use a FORMAT phrase:

```
INSERT INTO date_log
(1047
,CAST ('Jan 12, 2003' AS DATE FORMAT 'MMMBDD,BYYYY')
,CAST ('Jan 13, 2003' AS DATE FORMAT 'MMMBDD,BYYYY'));
```

## Example: Implicit Character-to-DATE Conversion

Assume that the DateForm mode of the session is set to ANSIDate.

The following CREATE TABLE statement specifies a FORMAT phrase for the DATE data type column:

```
CREATE SET TABLE datetab (f1 DATE FORMAT 'MMM-DD-YYYY');
```

In field mode, the following INSERT statement successfully performs the character to DATE implicit conversion because the format of the string conforms to the format of the DATE column in the datetab table:

```
INSERT INTO datetab ('JAN-10-1999');
```

In record or indicator mode, when the DateForm mode of the session is set to ANSIDate, the following INSERT statement successfully performs the character to DATE implicit conversion because the format of the string is in the ANSI DATE format:

```
INSERT INTO datetab ('2002-05-10');
```

## Related Information

- For an example, see [Example: ANSIDate Dateform Mode](#).
- For more information on default formats and the FORMAT phrase, see [Data Type Formats and Format Phrases](#).

## Character-to-INTERVAL Conversion

You can convert a character string to an INTERVAL value with either the CAST statement or Teradata conversion syntax.

### Character-to-INTERVAL Conversion with CAST

#### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI SQL, Teradata supports the specification of data attributes.

#### Syntax

```
CAST (
  character_expression AS interval_data_type
  [ data_attribute [...] ]
)
```

## Syntax Elements

### *character\_expression*

A character expression to be converted.

### *interval\_data\_type*

An INTERVAL data type to which the expression is to be converted.

### *data\_attribute*

One of the following data attributes:

- NAMED
- TITLE

## Teradata Character-to-INTERVAL Conversion Syntax

### ANSI Compliance

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

### Syntax

```
character_expression (
  [ data_attribute, [...] ]
  interval_data_type
  [, data_attribute [, ...]
)
```

## Syntax Elements

### *character\_expression*

A character expression to be cast to an INTERVAL value.

### *data\_attribute*

One of the following optional data attributes:

- NAMED
- TITLE

### *interval\_data\_type*

An INTERVAL data type to which *character\_expression* is to be converted.

## Usage Notes

The character value is trimmed of leading and trailing pad characters and handled as if it was a string literal in the declaration of an INTERVAL string literal.

Character-to-INTERVAL conversion is supported for CHAR and VARCHAR types only. The source character type cannot be CLOB.

If the contents of the character string can be converted to a valid INTERVAL, then they are; otherwise, an error is returned.

You cannot convert a character data type of GRAPHIC to an INTERVAL string literal.

## Examples

### Example: Querying with CAST

The following query returns '-265-11'.

```
SELECT CAST('-265-11' AS INTERVAL YEAR(4) TO MONTH);
```

### Example: Converting to an INTERVAL Value

If the source character string contains values not normalized in the INTERVAL form, but which nevertheless can be converted to a proper INTERVAL, the conversion is made.

For example, the following query returns '-267-06'

```
SELECT CAST('265-30' AS INTERVAL YEAR(4) TO MONTH);
```

## Character-to-Numeric Conversion

You can convert a character string to an numeric value with either the CAST statement or Teradata conversion syntax.

### Character-to-Numeric Conversion with CAST

#### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative formatting for the numeric data.



**Syntax**

```
CAST (
  character_expression AS numeric_data_type
  [ data_attribute [...] ]
)
```

**Syntax Elements*****character\_expression***

A character expression to be converted.

***numeric\_data\_type***

The numeric type to which *character\_expression* is to be converted.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

**Teradata Character-to-Numeric Conversion Syntax****ANSI Compliance**

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

**Syntax**

```
character_expression (
  [ data_attribute, [...] ]
  numeric_data_type
  [, data_attribute [, ...] ]
)
```

**Syntax Elements*****character\_expression***

A character expression to be converted.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

***numeric\_data\_type***

The data type to which the expression is to be converted.

## Usage Notes

Before processing begins, the numeric description is scanned for a FORMAT phrase, which is used to determine the radix separator, group separator, currency sign or string, signzone (S), or implied decimal point (V) formatting.

Conversion is performed positionally, character by character, from left to right, until the end of the number.

Only all-numeric character strings can be converted from character to numeric formats. For example, you can convert the character strings 'US Dollars 123456' or '123456' to the integer value 123456, but you cannot convert the string 'EX1AM2PL3E' to a numeric value.

The following list shows the steps for converting character type data to numeric. Note that you cannot convert a character\_expression of GRAPHIC character type to numeric.

Conversion is performed stage by stage, without returning to a previous stage; however, stages can be skipped.

1. Leading pad characters are ignored. Trailing pad characters are ignored, except for signed zoned decimal input.

Embedded spaces are only allowed according to the following rules:

- If the current SDF file defines the group separator as a space, then the character string can include spaces to separate groups of digits to the left of the radix separator, according to the grouping rule defined by GroupingRule or CurrencyGroupingRule.
  - If the current SDF file defines the radix separator as a space, then the character string can include one space as the radix character.
  - If the FORMAT phrase contains a currency formatting character, such as N, and the matching currency string in the SDF file, such as CurrencyName, contains a space, the character string can include spaces as part of that currency string.
2. The sign (+ or -) is saved as part of the number. A mantissa sign may appear before the first digit in the string, or after the last digit in the string. An exponent sign may appear with a preceding mantissa sign.
  3. The currency sign is ignored if it matches the FORMAT. A currency string is ignored if it matches the FORMAT. Only one currency is allowed in the string.

4. Digits are saved as the integral, fractional, or exponent part of the number, depending on whether the radix or the letter E has been parsed.
5. Separators are ignored, unless they match the radix specified in the FORMAT.  
If a separator matches the radix specified in the FORMAT, the location is saved as the beginning of the fractional part of the number. V marks the fractional component for implied decimals.  
The allowance of currency and separators is a non-ANSI Teradata extension of character to numeric conversion.
6. Embedded dashes (between digits) are allowed, unless the number is signed or includes a radix, currency, or exponent.
7. The letter E is saved as the beginning of the exponent part of the number. One space is allowed following an E.
8. The exponent sign (+ or -) is saved.
9. The exponent digits are saved. A sign character cannot appear after any exponent digit.

## Supported Character Types

The character expression to be converted must be CHAR or VARCHAR. CLOBs cannot be explicitly converted to numeric types.

## Numeric Overflow

In Field Mode, numeric overflow in character to numeric conversion is not treated as an error. If the result exceeds the number of digits normally reserved for the data type, asterisks are displayed.

In Record and Indicator Variable Modes, numeric overflow is reported as an error. This behavior applies to both the CAST and Teradata conversion syntax.

## FORMAT Phrase Controls Parsing of the Data

A FORMAT phrase, by itself, cannot convert a character type value to a numeric type value. The phrase controls partially how the resultant value is parsed.

Some examples of character to numeric conversion appear in the following table. For FORMAT phrases that contain G, D, C, and N formatting characters, assume that the related entries in the specification for data formatting file (SDF) are:

```
RadixSeparator {"."}
GroupSeparator {","}
GroupingRule   {"3"}
Currency       {"$"}

```

```
ISOCurrency    {"USD"}
CurrencyName  {"US Dollars"}
```

Character String	Converted To	Resultant Numeric Value	Field Mode Display Result
'\$20,000.00'	DECIMAL(10,2)	20000.00	20000.00
'\$\$\$50'	DECIMAL(10,2)	error Only one currency is allowed in the character string.	error
'\$.50'	DECIMAL(8,2)	.50	.50
'.345'	DECIMAL(8,3)	.345	.345
'-1.234E-02'	FLOAT	-.01234	-.01234
'-1E-.2'	FLOAT	error The radix must precede the exponent part of the number.	error
'00000000-.93'	DECIMAL(12,4)	error Embedded dashes cannot appear in a string containing a radix.	error
'- 55'	INTEGER	-55	-55
'E67'	FLOAT	0.0	0.000000000000000E 000
'9876'	DECIMAL(4,2) FORMAT '99V99'	98.76	9876
'-123'	INTEGER	-123	-123
'9876'	DECIMAL(4,2) FORMAT '9(2)V9(2)'	98.76	9876
'1-2-3'	INTEGER	123	123
'123-'	INTEGER	-123	-123
'123- '	INTEGER	-123	-123
'-1.234E 02'	FLOAT	-123.4	-1.234000000000000E 002
'111,222,333'	INTEGER FORMAT 'G9(I)'	111222333	0,111,222,333
'2.49US Dollars'	DECIMAL(10,2) FORMAT 'GZ(I)D9(F)BN'	2.49	2.49 US Dollars

Character String	Converted To	Resultant Numeric Value	Field Mode Display Result
'25000USD'	INTEGER FORMAT '9(I)C'	25000	0000025000USD

A conversion that does not specify a FORMAT phrase uses the corresponding data type default format as defined in the SDF.

## Implicit Character-to-Numeric Conversion

Implicit character to numeric conversion produces a valid result only if the character string represents a numeric value.

If a CHAR or VARCHAR character string is present in an expression that requires a numeric operand, it is read as a formatted numeric and is converted to a FLOAT value, using the default format for FLOAT.

To override the implicit format, use a FORMAT phrase.

Or, to change the default format for FLOAT, you can change the setting of the *REAL* element in the specification for data formatting (SDF) file. For information on default data type formats, the SDF file, and the FORMAT phrase, see [Data Type Formats and Format Phrases](#).

To use a CLOB type in an expression that requires a numeric operand, you must first explicitly convert the CLOB to CHAR or VARCHAR.

An empty character string (zero length) or a character string consisting only of pad characters is interpreted as having a numeric value of zero.

If the default format for FLOAT is -9.99E-99, then:

THIS expression ...	IS converted to ...	AND the result is ...
1.1*'\$20.00'	1.10E 00*2.00E1	2.20E 01
'2'+2'	2.00E 00+2.00E 00	4.00E 00
'A' + 2	-----	error

If a column or parameter of numeric data type is specified with a string value, the string is again assumed to be a formatted numeric. For example, the following INSERT statement specifies the Salary as a numeric string:

```
INSERT INTO Employee (EmpNo, Name, Salary)
VALUES (10022, 'Clements D', '$38,000.00');
```

The conversion to numeric type removes editing symbols. When selected, the salary data contains only the special characters allowed by the FORMAT phrase for Salary in the CREATE TABLE statement. If the FORMAT phrase is 'G-(9)D9(2)', then the output looks like this:

```

Salary
-----
38,000.00

```

If the FORMAT phrase is 'G-L(9)D9(2)', then the output looks like this:

```

Salary
-----
$38,000.00

```

## Example: Implicit Conversion of Character to Numeric

The INSERT statement in the following example implicitly converts the character data type to the target numeric data type:

```

CREATE TABLE t1
(f1 DECIMAL(10,2) FORMAT 'G-U(9)D9(2)');

INSERT t1 ('USD12,345,678.90');

```

If a column definition in a CREATE TABLE statement does not specify a FORMAT phrase for the data type, the column uses the corresponding data type default format as defined in the specification for data formatting (SDF) file.

## Related Information

For more information on default data type formats, the SDF file, and the meaning of formatting characters in a FORMAT phrase, see [Data Type Formats and Format Phrases](#).

## Character-to-Period Conversion

Converts a character string to a Period value.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

### Syntax

```

CAST (
  character_expression AS period_data_type
  [ data_attribute [...] ]
)

```

## Syntax Elements

### *character\_expression*

A character expression to be converted.

### *period\_data\_type*

The data type to which the expression is to be converted.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Usage Notes

A character value expression can be cast as PERIOD(DATE), PERIOD(TIME), or PERIOD(TIMESTAMP) using the CAST function or implicit casting. A character input value can also be implicitly cast as a Period type.

After any leading and trailing pad characters in the source character value are trimmed, the resulting character string must conform to the format of the target type. Conversion of the beginning and ending portions of the character value expression to corresponding DateTime values follow the existing rules of CHARACTER/VARCHAR to DateTime data type conversions.

The existing rules include conversion of the source value with a TIME or TIMESTAMP format to UTC based on the specified time zone in the source or, if not specified, the current session time zone. The exception to conversion to UTC for Period data types is when the ending portion of the source character is a TIMESTAMP value without a time zone and the value is equal to the maximum value that is used to represent UNTIL\_CHANGED; in this case, the value is not changed to UTC.

If the target type has a TIME or TIMESTAMP element type and the beginning or ending bound portions of the character value expression contains leap seconds, the seconds portion gets adjusted to 59.999999 with the precision truncated to the target precision.

If target type has a TIME or TIMESTAMP element type and the target precision is lower than either precision specified in the source character string, an error is reported. If the target precision is higher than a precision specified for a bound in the source character string, trailing zeros are added to the fractional seconds of the corresponding bound of the Period value resulting from the cast.

The target elements are set to the corresponding resulting values.

If the result beginning bound is not less than the result ending bound in their UTC forms, an error is reported.

If an ANSI DateTime format is used to interpret the character data during conversion, then enclosing the beginning and ending values inside apostrophes is optional. For details, see [Character Strings that Use ANSI DateTime Format](#).

## Character Strings that Use ANSI DateTime Format

Here is a list of valid character string representations when the implicit or explicit character-to-period conversion uses the ANSI DateTime format to interpret the beginning and ending bound elements.

- `'(' beginning_element_value ' ', Δ ' ending_element_value ' ' )'`
- `'(beginning_element_value , Δ ending_element_value )'`
- `'("beginning_element_value " , Δ ending_element_value )'`
- `'(beginning_element_value , Δ "ending_element_value " )'`

where formats of *beginning\_element\_value* and *ending\_element\_value* depend on the target data type.

Target Data Type	Format
PERIOD(DATE)	YYYY-MM-DD
PERIOD(TIME[(n)])	HH:MI:SS.S(F)
PERIOD(TIMESTAMP[(n)])	YYYY-MM-DDBHH:MI:SS.S(F)

## Implicit Character-to-Period Conversion

A CHARACTER or VARCHAR value is implicitly cast as a Period data type for an assignment, update, insert, merge, or parameter passing operation when the target site has a Period data type and for a comparison operation if the other operand has a Period data type. If any other non-Period value is directly assigned to a Period target site, an error is reported. In the same manner, if any other non-Period value is directly compared to a Period value, an error is reported.

### Note:

In some cases, a value may be explicitly cast as a Period data type in order to avoid this error.

During implicit conversion from CHARACTER or VARCHAR to Period data type, the ANSI DateTime format string is used to interpret the beginning and ending element values in the character string, if the response mode is other than the Field mode or if the character string data is parameterized. If the response mode is Field mode and if the character string data is not parameterized, then the target period format is used to interpret the beginning and ending element values in the character string. The following table describes this in detail.

Mode	Parameterized Data Present	Format for Implicit Cast Interpretation
Field	No	Target format



Mode	Parameterized Data Present	Format for Implicit Cast Interpretation
Field	Yes	ANSI format
Non-field	Yes	ANSI format
Non-field	No	ANSI format

When the ANSI DateTime format string is used to interpret the beginning and ending element values in the character string, enclosing the beginning and ending values inside the apostrophes is optional. This relaxation applies even during an explicit cast.

## Example: Casting Concatenated Character Literals

In the following example, two concatenated character literals are cast as PERIOD(TIMESTAMP(2)). The output is adjusted according to the current session time zone during display. Assume the current session time zone displacement is INTERVAL '-08:00' HOUR TO MINUTE and the format derived from SDF is 'YYYY-MM-DDBHH:MI:SS.S(2)Z'.

```
SELECT CAST('('2005-02-02 12:12:12.34+08:00'', ' ||
          ''2006-02-03 12:12:12.34+08:00'' )'
          AS PERIOD(TIMESTAMP(2)));
```

The following PERIOD(TIMESTAMP(2)) value is returned:

```
('2005-02-01 20:12:12.34', '2006-02-02 20:12:12.34')
```

## Related Information

- For the meanings of the format characters, see the description of the FORMAT phrase in [Data Type Formats and Format Phrases](#).
- For details, see [Character Strings that Use ANSI DateTime Format](#).

## Character-to-TIME Conversion

You can convert a character string to an TIME or TIME WITH TIME ZONE value with either the CAST statement or Teradata conversion syntax.

## Character-to-TIME Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative output formatting for the time data.

---

**Note:**

TIME (without time zone) is not ANSI SQL:2011 compliant. Vantage internally converts a TIME value to UTC based on the current session time zone or on a specified time zone.

---

## Syntax

```
CAST (
  character_expression AS TIME
  [ ( fractional_seconds_precision ) ]
  [ WITH TIME ZONE ]
  [ data_attribute [...] ]
)
```

## Syntax Elements

### *character\_expression*

A character expression to be converted.

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Teradata Character-to-TIME Conversion Syntax

### ANSI Compliance

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

Syntax

```
character_expression (  
  [ data_attribute [,...] ] TIME  
  [ ( fractional_seconds_precision ) ]  
  [, { data_attribute | WITH TIME ZONE } [,...] ]  
)
```

Syntax Elements

*character\_expression*

A character expression to be converted.

*data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

*fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

Usage Notes

The character value is trimmed of leading and trailing pad characters and handled as if it were a string literal in the declaration of a TIME string literal.

If the contents of the string can be converted to a valid TIME, the conversion is made; otherwise, an error is returned to the application.

Character-to-TIME conversion is supported for CHAR and VARCHAR types only. You cannot convert a character data type of CLOB or GRAPHIC to TIME.

You can use a FORMAT phrase to specify an explicit format for the TIME target data type. A conversion that does not specify a FORMAT phrase uses the default format for the TIME data type.

IF the character string is converted to ...	THEN the default format ...
TIME	does not use the time zone formatting character and does not display a time zone.

IF the character string is converted to ...	THEN the default format ...
TIME WITH TIME ZONE	uses the time zone formatting character to display the time zone.

## Conversions That Include Time Zone

The following rules apply to character-to-TIME conversions that include time zone information:

- If the target data type does not specify a time zone, for example, TIME(0), the source character string may contain a time zone of the format +hh:mi or -hh:mi, but only if it appears immediately before or immediately after the time.

For example, the following conversion is successful:

```
SELECT CAST ( '-02:0011:23:44'
AS TIME(0) );
```

The following conversion is not successful because of the blank separator character between the time zone and the time:

```
SELECT CAST ( '+02:00 11:23:44.56'
AS TIME(2) );
```

- If the source character string contains a time zone, and the target data type does not specify a time zone, for example, TIME(0), the conversion uses the time zone in the character string to convert the character string to Universal Coordinated Time (UTC). This is done regardless of whether the FORMAT phrase contains the time zone formatting character.

```
SELECT CAST ( '10:15:12+12:30'
AS TIME(0));
```

- If the source character string does not contain a time zone, and the target data type specifies a time zone and a target FORMAT phrase that includes time zone formatting characters, the output includes the session time zone.

```
SELECT CAST ( '10:15:12'
AS TIME(0) WITH TIME ZONE FORMAT 'HH:MI:SSBZ');
```

- If both the source character string and the target data type do not specify a time zone, the source character string is internally converted to UTC based on the current session time zone.

## Conversions That Include Fractional Seconds

The following rules apply to conversions that include fractional seconds:

- The fractional seconds precision in the source character string must be less than or equal to the fractional seconds precision specified by the target type.

```
SELECT CAST('12:30:25.44' AS TIME(3));
```

If no fractional seconds appear in the source character string, then the fractional seconds precision is always less than or equal to the target data type fractional seconds precision, because the valid range for the precision is zero to six, where the default is six.

```
SELECT CAST('12:30:25' AS TIME(3));
```

- If the target data type is defined by a FORMAT phrase, the fractional seconds precision formatting characters must be greater than or equal to the precision specified by the data type.

```
SELECT CAST('12h:15.12s:30m'
AS TIME(4) FORMAT 'HHh:SSDS(4)s:MIIm');
```

A FORMAT phrase must specify a fractional seconds precision of six if the target data type does not specify a fractional seconds precision, because the default precision is six.

```
SELECT CAST ('12:30:25' AS TIME FORMAT 'HH:MI:SSDS(6)');
```

## Character Strings That Omit Hour, Minute, or Second

If the character string in a character-to-TIME conversion omits the hour, minute, or second, the system uses default values for the target TIME value.

IF the character string omits the ...	THEN the system uses the ...
hour	value of 0.
minute	
second	

Consider the following table:

```
CREATE TABLE time_log
(id INTEGER
, start_time TIME
, end_time TIME
, log_time TIME);
```

The following INSERT statement converts three character strings to TIME values. The first character string omits the hour, the second character string omits the minute, and the third character string omits the second.

```
INSERT time_log
(1001
,CAST ('01:02.030405' AS TIME FORMAT 'MI:SS.S(6)')
,CAST ('01:02.030405' AS TIME FORMAT 'HH:SS.S(6)')
,CAST ('01:02' AS TIME FORMAT 'HH:MI'));

```

The result of the INSERT statement is as follows:

```
SELECT * FROM time_log;
      id      start_time      end_time      log_time
-----
      1001  00:01:02.030405  01:00:02.030405  01:02:00.000000

```

## FORMAT Phrase Restrictions

In character-to-TIME conversions, the FORMAT phrase must not consist solely of the following formatting characters:

- Z
- T

## Implicit Character-to-TIME Conversion

In field mode, the string must conform to the format of the target TIME type.

In record or indicator mode, the string must use the ANSI TIME format.

## Examples

### Example: Fractional Seconds

This query returns the value '12:23:39.999900' (with the fractional seconds extended to 6 places as requested by CASTing to a TIME(6) type).

```
SELECT CAST(' 12:23:39.9999 '
AS TIME(6));

```

### Example: Truncation of Non-pad Character Data

This query returns an error because the requested conversion requires truncation of non-pad character data.

```
SELECT CAST(' 12:23:39.9999 '
AS TIME(3));
```

### Example: Non Valid MINUTE Value

This query returns an error because the MINUTE value of 63 is not valid.

```
SELECT CAST(' 12:63:39.9999 '
AS TIME(6));
```

### Example: FORMAT Phrase

This query returns the value '15h33m'.

```
SELECT CAST('15h33m'
AS TIME(0) FORMAT 'HHhMI m');
```

### Example: Implicit Conversion of Character to TIME

The following CREATE TABLE statement specifies a FORMAT phrase for the TIME data type column:

```
CREATE SET TABLE timetab (f1 TIME(0) FORMAT 'TBHHhMI mSSs');
```

In field mode, the following INSERT statement successfully performs the character to TIME implicit conversion because the format of the string conforms to the format of the TIME column in the timetab table:

```
INSERT INTO timetab ('AM 10h20m30s');
```

In record or indicator mode, the following INSERT statement successfully performs the character to TIME implicit conversion because the format of the string is in the ANSI TIME format:

```
INSERT timetab ('11:23:34');
```

## Related Information

For more information on default formats and the FORMAT phrase, see [Data Type Formats and Format Phrases](#).

## Character-to-TIMESTAMP Conversion

You can convert a character string to a TIMESTAMP or TIMESTAMP WITH TIME ZONE value with either the CAST statement or Teradata conversion syntax.

## Character-to-TIMESTAMP Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative formatting for the time data.

---

#### Note:

TIMESTAMP (without time zone) is not ANSI SQL:2011 compliant. Vantage internally converts a TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

---

### Syntax

```
CAST (
  character_expression AS TIMESTAMP
  [ ( fractional_seconds_precision ) ]
  [ WITH TIME ZONE ]
  [ data_attribute [...] ]
)
```

### Syntax Elements

#### *character\_expression*

A character expression to be converted.

#### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.



***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Teradata Character-to-TIMESTAMP Conversion Syntax

### ANSI Compliance

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

### Syntax

```
character_expression (
  [ data_attribute [,...] ] TIMESTAMP
  [ ( fractional_seconds_precision ) ]
  [, { data_attribute | WITH TIME ZONE } [,...] ]
)
```

### Syntax Elements

***character\_expression***

A character expression to be converted.

***fractional\_seconds\_precision***

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Usage Notes

The source expression is trimmed of leading and trailing pad characters and then handled as if it were a string literal in the declaration of a TIMESTAMP string literal.

Character-to-TIMESTAMP conversion is supported for CHAR and VARCHAR types only. You cannot convert a character data type of CLOB or GRAPHIC to TIMESTAMP.

If the contents of the string can be converted to a valid TIMESTAMP value, then the conversion is performed; otherwise an error is returned.

You can use a FORMAT phrase to specify an explicit format for the TIMESTAMP target data type. A conversion that does not specify a FORMAT phrase uses the default format for the TIMESTAMP data type.

IF the character string is converted to ...	THEN the default format ...
TIMESTAMP	does not use the time zone formatting character and does not display a time zone.
TIMESTAMP WITH TIME ZONE	uses the time zone formatting character to display the time zone.

## Conversions That Include Time Zone

The following rules apply to character-to-TIMESTAMP conversions that include time zone information:

- If the target data type does not specify a time zone, for example, TIMESTAMP(0), the source character string may contain a time zone of the format +hh:mi or -hh:mi, but only if it appears immediately before or immediately after the time.

For example, the following conversion is successful:

```
SELECT CAST ( '2008-09-19 11:23:44-02:00'
AS TIMESTAMP(0) FORMAT 'Y4-MM-DDBHH:MI:SSBZ' );
```

The following conversion is not successful because of the blank separator character between the time zone and the time:

```
SELECT CAST ( '2008-01-19 +02:00 11:23:44'
AS TIMESTAMP(0) FORMAT 'Y4-MM-DDBZBHH:MI:SS' );
```

- If the source character string contains a time zone, and the target data type does not specify a time zone, the conversion uses the time zone in the character string to convert the character string to Universal Coordinated Time (UTC). This is done whether or not the FORMAT phrase contains the time zone formatting character.

```
SELECT CAST ( '2002-02-20 10:15:12+12:30' AS TIMESTAMP(0) );
```

- If the target FORMAT phrase includes time zone formatting characters, and the source character string does not contain a time zone, the output includes the session time zone. This is done whether or not the target data type specifies a time zone.

```
SELECT CAST ( '2002-02-20 10:15:12'
AS TIMESTAMP(0) WITH TIME ZONE FORMAT 'Y4-MM-DDBHH:MI:SSBZ' );
```

- If both the source character string and the target data type do not specify a time zone, the source character string is internally converted to UTC based on the current session time zone.

## Conversions That Include Fractional Seconds

The following rules apply to conversions that include fractional seconds:

- The fractional seconds precision in the source character string must be less than or equal to the fractional seconds precision specified by the target type.

```
SELECT CAST( '2002-01-01 12:30:25.44' AS TIMESTAMP(3));
```

If no fractional seconds appear in the source character string, then the fractional seconds precision is always less than or equal to the target data type fractional seconds precision, because the valid range for the precision is zero to six, where the default is six.

```
SELECT CAST( '2002-01-01 12:30:25' AS TIMESTAMP(3));
```

- If the target data type is defined by a FORMAT phrase, the fractional seconds precision formatting characters must be greater than or equal to the precision specified by the data type.

```
SELECT CAST( '12-02-07 12:30:25' AS TIMESTAMP(3)
FORMAT 'DD-MM-YYBHH:MI:SSDS(3)');
```

A FORMAT phrase must specify a fractional seconds precision of six if the target data type does not specify a fractional seconds precision, because the default precision is six.

```
SELECT CAST( '12-02-07 12h:15.12s:30m'
AS TIMESTAMP FORMAT 'DD-MM-YYBHHh:SSDS(6)s:MIm');
```

## Character Strings That Omit Day, Month, Year, Hour, Minute, or Second

If the character string in a character-to-TIMESTAMP conversion omits the day, month, year, hour, minute, or second, the system uses default values for the target TIMESTAMP value.

IF the character string omits the ...	THEN the system uses the ...
day	value of 1 (the first day of the month).
month	value of 1 (the month of January).
year	current year.

IF the character string omits the ...	THEN the system uses the ...
hour	value of 0.
minute	
second	

Consider the following table:

```
CREATE TABLE timestamp_log
(id INTEGER, start_ts TIMESTAMP, end_ts TIMESTAMP);
```

The following INSERT statement converts two character strings to TIMESTAMP values. Both strings omit the hour, minute, and second. Additionally, the first character string omits the day and the second character string omits the month.

```
INSERT timestamp_log
(1001
,CAST ('January 2006' AS TIMESTAMP FORMAT 'MMMMBYYYY')
,CAST ('2006-01' AS TIMESTAMP FORMAT 'YYYY-DD'));
```

The result of the INSERT statement is as follows:

```
SELECT * FROM timestamp_log;
      id      start_ts      end_ts
-----
1001  2006-01-01 00:00:00.000000  2006-01-01 00:00:00.000000
```

Here is an INSERT statement where both character strings omit the year. Additionally, the first character string omits the hour and the second character string omits the minute. Assume the current year is 2003.

```
INSERT timestamp_log
(1002
,CAST ('January 23 04:05' AS TIMESTAMP FORMAT 'MMMBDDDBMI:SS')
,CAST ('01-23 04:05' AS TIMESTAMP FORMAT 'MM-DDBHH:SS'));
```

The result of the INSERT statement is as follows:

```
SELECT * FROM timestamp_log WHERE id = 1002;
      id      start_ts      end_ts
-----
1001  2003-01-23 00:04:05.000000  2003-01-23 04:00:05.000000
```

## Restrictions on FORMAT Phrase

In character-to-TIMESTAMP conversions, the FORMAT phrase must not consist solely of the following formatting characters:

- EEEE
- E4
- EEE
- E3
- T
- Z

## Implicit Character-to-TIMESTAMP Conversion

In field mode, the string must conform to the format of the target TIMESTAMP type.

In record or indicator mode, the string must use the ANSI TIMESTAMP format.

## Example: Querying with CAST

The following query returns '2007-12-31 23:59:59.999999-08:00'.

```
SELECT CAST('2007-12-31 23:59:59.999999'  
AS TIMESTAMP(6) WITH TIME ZONE);
```

Notice that the source character string did not need to have explicit Time Zone fields for this conversion to work properly.

## Related Information

For more information on default formats and the FORMAT phrase, see [Data Type Formats and Format Phrases](#).

## Character-to-UDT Conversion

Converts a character data string to a UDT.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

## Syntax

```
CAST ( character_expression AS UDT_data_type )
```

## Syntax Elements

### *character\_expression*

A character expression to be converted.

### *UDT\_data\_type*

The data type to which the expression is to be converted.

## Usage Notes

Explicit character-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement.

## Implicit Character-to-UDT Conversion

SQL Engine performs implicit Character-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see [Usage Notes](#)) exists that specifies the AS ASSIGNMENT clause.

The source character type of the cast definition does not have to be an exact match to the source character type of the implicit conversion. SQL Engine can use an implicit cast definition that specifies a CHAR, VARCHAR, or CLOB source type.

If multiple implicit cast definitions exist for converting different character types to the UDT, SQL Engine uses the implicit cast definition for the character type with the highest precedence. The following list shows the precedence of character types in order from lowest to highest precedence:

- CHAR
- VARCHAR
- CLOB

For non-CLOB character types, if no Character-to-UDT implicit cast definitions exist, SQL Engine looks for other cast definitions that can substitute.

IF the following combination of implicit cast definitions exists ...				THEN SQL Engine ...
Numeric-to-UDT	DATE-to-UDT	TIME-to-UDT	TIMESTAMP-to-UDT	
X				uses the numeric-to-UDT implicit cast definition. If multiple numeric-to-UDT implicit cast definitions exist, then SQL Engine returns an SQL error.
	X			uses the DATE-to-UDT implicit cast definition.
		X		uses the TIME-to-UDT implicit cast definition.
			X	uses the TIMESTAMP-to-UDT implicit cast definition.
X	X			reports an error.
X		X		
X			X	
	X	X		
	X		X	
		X	X	
X	X	X		
X	X		X	
X		X	X	
	X	X	X	
X	X	X	X	

Substitutions are valid because Vantage can implicitly cast the non-CLOB character type to the substitute data type, and then use the implicit cast definition to cast from the substitute data type to the UDT.

## Related Information

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

# Character Data Type Assignment Rules

## Server Character Sets

LATIN, UNICODE, KANJISJIS, KANJI1, and GRAPHIC server character sets are generally mutually assignable.

Consider an assignment of an expression to a character string column. The assignment may be the result of the SQL UPDATE or INSERT statement, or it may be the result of a Load utility assignment.

The expression is converted to the server character set of the character column.

## Exceptions to GRAPHIC Data

The following exceptions apply to GRAPHIC data:

- When you import GRAPHIC data and assign it to a character column, that column must be defined as GRAPHIC.
- When you import character data that is not GRAPHIC, you cannot assign it to a column defined as GRAPHIC.

For more information, see the documentation on the USING row descriptor in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

- You cannot assign non-GRAPHIC data to a GRAPHIC column from BTEQ or load utilities.

For more information, see the documentation on the USING row descriptor in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

- You cannot assign or export GRAPHIC data from a single byte character set like ASCII or EBCDIC.

## DATE-to-Character Conversion

You can convert a DATE value to a character string with either the CAST statement or Teradata conversion syntax.

## DATE-to-Character Conversion with CAST

### ANSI Compliance

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

### Syntax

```
CAST (
  date_expression AS character_data_type
  [ CHARACTER SET server_character_set ]
  [ data_attribute [...] ]
)
```



## Syntax Elements

### *date\_expression*

A date expression to be converted.

### *character\_data\_type*

The data type to which the expression is to be converted.

### *server\_character\_set*

The server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default server character set is used.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Teradata DATE-to-Character Conversion Syntax

### ANSI Compliance

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

### Syntax

```
date_expression (
  [ data_attribute, [...] ] character_data_type
  [, { data_attribute | CHARACTER SET server_character_set } [,... ] ]
)
```

## Syntax Elements

### *date\_expression*

A date expression to be converted.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

***character\_data\_type***

The data type to which the expression is to be converted.

***server\_character\_set***

The server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default server character set is used.

## Usage Notes

When converting DATE to CHAR(*n*) or VARCHAR(*n*), then *n* must be equal to or greater than the length of the DATE value as represented by a character string literal.

IF the target data type is ...	AND <i>n</i> is ...	THEN ...
CHAR( <i>n</i> )	greater than the length of the DATE value as represented by a character string literal	trailing pad characters are added to pad the representation.
	too small	a string truncation error is returned.
VARCHAR( <i>n</i> )	greater than the length of the DATE value as represented by a character string literal	no blank padding is added to the character representation.
	too small	a string truncation error is returned.

## Restrictions

DATE types cannot be implicitly or explicitly converted to character types if the server character set is GRAPHIC.

DATE to CLOB conversion is not supported.

## Forcing a FORMAT on CAST for Converting DATE to Character

The default format for DATE to character conversion uses the format in effect for the DATE value.

To override the default format, you can convert a DATE value to a string using a FORMAT phrase. The resulting format, however, is the same as the DATE value. If you want a different format for the string value, you need to also use CAST as described here.

You must use nested CAST operations in order to convert values from DATE to CHAR and force an explicit FORMAT on the result regardless of the format associated with the DATE value. This is because of the rules for matching FORMAT phrases to data types.

## Examples

### Example: Converting a DATE Value to a Character String

The dateform mode of the session is INTEGERDATE and column F1 in the table INTDAT is a DATE value with the explicit format 'YYYY,MMM,DD'.

```
SELECT F1 FROM INTDAT ;
```

The result (without a type change) is the following report:

```
F1
-----
1900,Dec,31
```

Assume that you want to convert this to a value of CHAR(12), and an explicit output format of 'MMMBDD,BYYYY'. Use nested CAST phrases and a FORMAT to obtain the desired result: a report in character format.

```
SELECT
  CAST( (CAST (F1 AS FORMAT 'MMMBDD,BYYYY')) AS CHAR(12))
FROM INTDAT;
```

The result after the nested CASTs is the following report.

```
F1
-----
Dec 31, 1900
```

The inner CAST establishes the display format for the DATE value and the outer CAST indicates the data type of the desired result.

## Example: Creating a Script to Convert Date Values

Suppose you need to create a script to convert date values to the ANSI DATE format, regardless of the source of the DATE value or the DATEFORM mode of the session.

You can use nested CASTs and a FORMAT to do this as demonstrated by the example that follows.

```
SELECT
  CAST( (CAST (F1 AS FORMAT 'YYYY-MM-DD')) AS CHAR(10))
FROM INTDAT;
```

The result after the nested CASTs is the following report.

```
F1
-----
1900-12-31
```

## DATE-to-DATE Conversion

You can convert the format or title of a DATE value with either the CAST statement or Teradata conversion syntax.

## DATE-to-DATE Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative output formatting of date data. This is a Teradata extension to CAST.

A DATE-to-DATE conversion involving a DATE type with a dateform of INTEGERDATE is a Teradata extension to the ANSI SQL:2011 standard. This is a Teradata extension to CAST

### Syntax

```
CAST ( date_expression AS
  { DATE [ data_attribute [...] ] |
    [ data_attribute [...] ]
  }
)
```

## Syntax Elements

### *date\_expression*

A date expression to be converted.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Teradata DATE-to-DATE Conversion Syntax

### ANSI Compliance

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

### Syntax

```
date_expression (
  { DATE [, data_attribute [,...]] } |

  [ data_attribute [,...]] [, DATE [, data_attribute [,...]] ]
}
```

## Syntax Elements

### *date\_expression*

A date expression to be converted.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Example: Finding Employee Birthdates

Consider a table named `employee` that was created with a session dateform mode of `INTEGERDATE` where `dob` is a `DATE` column with a format of `M3BDDBY4`. To list employees who were born between January 30, 1938, and March 30, 1943, you could specify the date information as follows:

```
SELECT name, dob
FROM employee
WHERE dob BETWEEN 'Jan 30 1938' AND 'Mar 30 1943'
ORDER BY dob;
```

The result returns the date of birth information as specified for the `Employee` table:

Name	DOB
-----	-----
Inglis C	Mar 07 1938
Peterson J	Mar 27 1942

To change the date format to an alternate form, change the `SELECT` to:

```
SELECT name, dob (FORMAT 'yy-mm-dd')
FROM employee
WHERE dob BETWEEN 'Jan 30 1938' AND 'Mar 30 1943'
ORDER BY dob ;
```

The format specification changes the display to the following:

Name	DOB
-----	-----
Inglis C	38-03-07
Peterson J	42-03-27

## DATE-to-Numeric Conversion

You can convert a `DATE` value to the following numeric types with either the `CAST` statement or Teradata conversion syntax:

- `BYTEINT`
- `SMALLINT`
- `INTEGER`
- `BIGINT`
- `DECIMAL (n, m)`

- NUMBER
- FLOAT

## DATE-to-Numeric Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of numeric data attribute phrases.

### Syntax

```
CAST (
    date_expression AS numeric_data_type
    [ data_attribute [...] ]
)
```

### Syntax Elements

#### *date\_expression*

A date expression to be converted.

#### *numeric\_data\_type*

The data type to which the expression is to be converted.

#### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Teradata DATE-to-Numeric Conversion Syntax

### ANSI Compliance

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

### Syntax

```
date_expression (
    [ data_attribute, [...] ]
```

```

    numeric_data_type
    [, data_attribute [, ...]
)

```

## Syntax Elements

### *date\_expression*

A date expression to be converted.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

### *numeric\_data\_type*

The data type to which the expression is to be converted.

## Usage Notes

When a date is converted to a numeric, the value returned is the integer value for the internal stored date, which is encoded using the following formula:

```
(year - 1900) * 10000 + (month * 100) + day
```

Allowable date values range from AD January 1, 0001 to AD December 31, 9999.

For example, December 31, 1985 would be stored as the integer 851231; July 4, 1776 stored as -1239296; and March 30, 2041 stored as 1410330.

Conversion of DATE to DECIMAL(*n,m*) where the number of digits (*n*) is too small generates a numeric overflow error. Conversion of DATE to BYTEINT or SMALLINT generates a numeric overflow error if the value returned is outside the range of values that the data type can represent.

No error is generated on conversion of DATE to INTEGER or FLOAT.

## FORMAT Phrase

A FORMAT phrase in DATE to numeric conversion may only contain the 9 or Z formatting character. For example:



```
SELECT CAST (DATE '2007-12-31' AS INTEGER FORMAT '9999999');
```

## Implicit DATE-to-Numeric Conversion

SQL Engine performs implicit DATE-to-numeric type conversion when you assign a DATE type to a numeric type, compare a DATE type and numeric type, or pass a DATE type to a system function that takes a numeric type.

## Example: DATE-to-Numeric Conversion

The following example converts DATE data in the dob column of the employee table to a numeric format.

Note that the best practice is to define date data as a DATE type; do not define date data as a numeric type.

To change the display from date format to integer format, change the statement to:

```
SELECT name, dob (INTEGER)
FROM employee
WHERE dob BETWEEN 380307 AND 420825
ORDER BY dob ;
```

or

```
SELECT name, CAST (dob AS INTEGER)
FROM employee
WHERE dob BETWEEN 380307 AND 420825
ORDER BY dob ;
```

and the display becomes:

Name	DOB
-----	-----
Inglis C	380307
Peterson J	420327

## DATE-to-Period Conversion

Casts a DATE value to a PERIOD(DATE) or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) value.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

## Syntax

```
CAST (
  date_expression AS period_data_type
  [ data_attribute [...] ]
)
```

## Syntax Elements

### *date\_expression*

A date expression to be converted.

### *period\_data\_type*

The data type to which the expression is to be converted.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Usage Notes

A DATE value can be cast as PERIOD(DATE) or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function. If an attempt is made to cast a DATE value as PERIOD(TIME[(n)] [WITH TIME ZONE]), an error is reported.

If the target type is PERIOD(DATE), the result beginning element is set to the source value. The result ending element is set to the result beginning bound plus one granule of the target type (that is, INTERVAL '1' DAY). If the result ending bound exceeds the maximum DATE value (that is, the source value is equal to the maximum DATE value), or the result ending bound equal to maximum DATE value (that is, the resulting ending bound value equal to value of UNTIL\_CHANGED) an error is reported.

If the target type is PERIOD(TIMESTAMP[(n)]), the result beginning element is set to the UTC value obtained using the current session time zone and a timestamp value formed from the source DATE value and a time portion of zero. The result ending element is set to the result beginning bound plus one granule of the target type (note that this cannot cause an error).

If the target type is PERIOD(TIMESTAMP[(n)] WITH TIME ZONE), the time portion of the result beginning element is set to the UTC value obtained using the current session time zone and a timestamp value formed from the source DATE value and a time portion of zero. The time zone of the result beginning

element is set to the current session time zone displacement. The result ending element is set to the result beginning bound plus one granule of the target type (note that this cannot cause an error).

---

**Note:**

The result has the same value for the beginning bound and last value.

---

## Examples

### Example: Casting a DATE literal as PERIOD(DATE)

In the following example, a DATE literal is cast as PERIOD(DATE). The result beginning bound is obtained from the source. The result ending element is set to the result beginning bound plus INTERVAL '1' DAY.

```
SELECT CAST(DATE '2005-02-03' AS PERIOD(DATE));
```

The following PERIOD(DATE) value is returned:

```
('2005-02-03', '2005-02-04')
```

### Example: Casting a DATE literal as PERIOD(TIMESTAMP(4))

In the following example, a DATE literal is cast as PERIOD(TIMESTAMP(4)). The result beginning bound is formed from the DATE literal and a time portion of zero. The result ending element is set to the result beginning bound plus INTERVAL '0.0001' SECOND.

```
SELECT CAST(DATE '2005-02-03' AS PERIOD(TIMESTAMP(4)));
```

The following PERIOD(TIMESTAMP(4)) value is returned:

```
('2005-02-03 00:00:00.0000', '2005-02-03 00:00:00.0001')
```

## DATE-to-TIMESTAMP Conversion

You can convert a DATE value to a TIMESTAMP or TIMESTAMP WITH TIME ZONE value with either the CAST statement or Teradata conversion syntax.

## DATE-to-TIMESTAMP Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting of timestamp data.

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when converting from DATE to TIMESTAMP using CAST. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

---

#### Note:

TIMESTAMP (without time zone) is not ANSI SQL:2011 compliant. Vantage internally converts a TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

---

### Syntax

```
CAST (
  date_expression AS TIMESTAMP
  [ ( fractional_seconds_precision ) ]
  [ WITH TIME ZONE ]
  [ AT { LOCAL | [ TIME ZONE ] { expression | time_zone_string } } ]
  [ data_attribute [...] ]
)
```

### Syntax Elements

#### *date\_expression*

A date expression to be converted.

#### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

#### AT LOCAL

Use the time zone displacement based on the current session time zone.

**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.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Teradata DATE-to-TIMESTAMP Conversion Syntax

### ANSI Compliance

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

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when converting from DATE to TIMESTAMP using Teradata conversion syntax. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

---

**Note:**

TIMESTAMP (without time zone) is not ANSI SQL:2011 compliant. Vantage internally converts a TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

---

### Syntax

```
date_expression (
  [ data_attribute [,...] ] TIMESTAMP
  [ ( fractional_seconds_precision ) ]
  [, WITH TIME ZONE ]
  [ AT { LOCAL | [ TIME ZONE ] { expression | time_zone_string } } ]
  [, data_attribute [,...] ]
)
```

## Syntax Elements

### *date\_expression*

A date expression to be converted.

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

### AT LOCAL

Use the time zone displacement based on the current session time zone.

### 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.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Usage Notes

The following table shows the result of the CAST function or Teradata conversion based on the various options specified. If the target precision is higher than zero, trailing zeros are added in the result to adjust the precision.

IF you specify...	THEN...
AT LOCAL	<p>a local timestamp value is formed from the source <i>date_expression</i> with the time portion set to '00:00:00'. Then, the result is formed from this local timestamp value adjusted to UTC by subtracting the time zone displacement based on the current session time zone.</p> <p>This is the same as not specifying the AT clause.</p>

IF you specify...	THEN...
AT expression or AT TIME ZONE <i>expression</i>	a local timestamp value is formed from the source <i>date_expression</i> with the time portion set to '00:00:00'. Then, the result is formed from this local timestamp value adjusted to UTC by subtracting the time zone displacement defined by <i>expression</i> .
AT <i>time_zone_string</i> or AT TIME ZONE <i>time_zone_string</i>	a local timestamp value is formed from the source <i>date_expression</i> with the time portion set to '00:00:00'. The time zone displacement is determined based on <i>time_zone_string</i> and the local timestamp value. Then, the result is formed from the local timestamp value adjusted to UTC by subtracting the time zone displacement.

## Implicit DATE-to-TIMESTAMP Conversion

SQL Engine performs implicit conversion from DATE to TIMESTAMP types in some cases. See [Implicit Conversion of DateTime Types](#).

The following conversions are supported:

From source type...	To target type...
DATE ANSI Date dateform mode or IntegerDate dateform mode	TIMESTAMP TIMESTAMP WITH TIME ZONE

The TIMESTAMP value is always converted to DATE in case of comparison. See [TIMESTAMP-to-DATE Conversion](#).

## Examples

### Example: Converting a DATE Value to a TIMESTAMP Value

In this example, the result of the CAST is the timestamp formed from the source expression value '2008-05-14' and the default time '00:00:00' adjusted to UTC by the current session time zone displacement, INTERVAL '01:00' HOUR TO MINUTE. Thus, the value of the CAST is '2008-05-13 23:00:00' at UTC.

The result value of the CAST at UTC is adjusted to the current session time zone displacement, INTERVAL '01:00' HOUR TO MINUTE, so the result of the SELECT statements is: TIMESTAMP '2008-05-14 00:00:00'.

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;
SELECT CAST(DATE '2008-05-14' AS TIMESTAMP(0));
SELECT CAST(DATE '2008-05-14' AS TIMESTAMP(0) AT LOCAL);
```

## Example: Converting a DATE Value to a TIMESTAMP WITH TIME ZONE value

In this example, the result of the CAST is the timestamp formed from the source expression value '2008-05-14' and the default time '00:00:00' adjusted to UTC by the current session time zone displacement, INTERVAL '06:00' HOUR TO MINUTE. Thus, the value of the CAST is '2008-05-13 18:00:00' at UTC with the current session time zone displacement INTERVAL '06:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to its time zone displacement, INTERVAL '06:00' HOUR TO MINUTE, so the result of the SELECT statements is: TIMESTAMP '2008-05-14 00:00:00+06:00'.

```
SET TIME ZONE INTERVAL '06:00' HOUR TO MINUTE;
SELECT CAST(DATE '2008-05-14' AS TIMESTAMP(0) WITH TIME ZONE);
SELECT CAST(DATE '2008-05-14' AS TIMESTAMP(0) WITH TIME ZONE
  AT LOCAL);
```

## Example: CAST for a TIMESTAMP

In the following SELECT statement, the result of the CAST is the timestamp formed from the date '2008-05-14' and the default time '00:00:00' adjusted to UTC by the specified time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE. Thus, the value of the CAST is '2008-05-14 08:00:00' at UTC.

The result value of the CAST at UTC is adjusted to the current session time zone displacement, INTERVAL '05:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-14 13:00:00'.

```
SET TIME ZONE INTERVAL '05:00' HOUR TO MINUTE;
SELECT CAST(DATE '2008-05-14' AS TIMESTAMP(0) AT -8);
```

Consider the following SELECT statement:

```
SELECT CAST(DATE '2008-05-14' AS TIMESTAMP(0) WITH TIME ZONE AT -8);
```

In this case, the result of the CAST is the timestamp formed from the source expression value '2008-05-14' and the default time '00:00:00' adjusted to UTC by the specified time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE. Thus, the value of the CAST is '2008-05-14 08:00:00' at UTC with the specified time zone displacement INTERVAL '-08:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to its time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-14 00:00:00-08:00'. The current session time zone has no effect.



## Example: Converting the DATE Value to a TIMESTAMP Value Based On a Time Zone String

In this example, the current timestamp is:

```

Current TimeStamp(6)
-----
2010-03-09 19:23:27.620000+00:00

```

The following statement converts the DATE value '2010-03-09' to a TIMESTAMP value, where the time zone displacement is based on the time zone string, 'America Pacific'.

```
SELECT CAST(DATE '2010-03-09' AS TIMESTAMP(0) AT 'America Pacific');
```

The result of the query is:

```

2010-03-09
-----
2010-03-09 08:00:00

```

## DATE-to-UDT Conversion

Converts DATE data to UDT data.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

### Syntax

```
CAST ( date_expression AS UDT_data_type )
```

### Syntax Elements

#### *date\_expression*

A date expression to be converted.

#### *UDT\_data\_type*

The data type to which the expression is to be converted.

## Usage Notes

Explicit DATE-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement.

## Implicit DATE-to-UDT Conversion

Performing an implicit data type conversion requires that an appropriate cast definition (see [Usage Notes](#)) exists that specifies the AS ASSIGNMENT clause.

SQL Engine performs implicit DATE-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

If no DATE-to-UDT implicit cast definition exists, Vantage looks for other cast definitions that can substitute.

IF the following combination of implicit cast definitions exists ...		THEN SQL Engine ...
Numeric-to-UDT	Character-to-UDT a non-CLOB character type	
X		uses the Numeric-to-UDT implicit cast definition. If multiple Numeric-to-UDT implicit cast definitions exist, then SQL Engine returns an SQL error.
	X	uses the Character-to-UDT implicit cast definition. If multiple Character-to-UDT implicit cast definitions exist, then SQL Engine returns an SQL error.
X	X	reports an error.

Substitutions are valid because Vantage can implicitly cast a DATE type to the substitute data type, and then use the implicit cast definition to cast from the substitute data type to the UDT.

## Related Information

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

## INTERVAL-to-Character Conversion

You can convert an INTERVAL value to its canonical character string representation with either the CAST statement or Teradata conversion syntax.

INTERVAL-to-Character conversion is supported for CHAR and VARCHAR types only. The target type cannot be CLOB.

## INTERVAL-to-Character Conversion with CAST

### ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of character data attribute phrases.

### Syntax

```
CAST (
  interval_expression AS character_data_type
  [ CHARACTER SET server_character_set ]
  [ data_attribute [...] ]
)
```

### Syntax Elements

#### *interval\_expression*

An INTERVAL expression to be converted.

#### *character\_data\_type*

The data type to which the expression is to be converted.

#### *server\_character\_set*

The server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default server character set is used.

#### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Usage Notes

### INTERVAL-to-Fixed CHARACTER Conversion

When the target data type is CHAR( $n$ ), then  $n$  must be equal to or greater than the length of the canonical form of the value as represented by a character string literal.

If  $n$  is greater than that length, trailing pad characters are added to pad the canonical representation.

If  $n$  is too small, then a string truncation error is returned.

### INTERVAL-to-VARCHAR Conversion

When the target data type is VARCHAR( $n$ ), then  $n$  must be equal to or greater than the length of the canonical form of the value as represented by a varying character string literal.

If  $n$  is too small, then a string truncation error is returned.

## Teradata INTERVAL-to-Character Conversion Syntax

### ANSI Compliance

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

#### INTERVAL-to-Fixed CHARACTER Conversion

When the target data type is CHAR( $n$ ), then  $n$  must be equal to or greater than the length of the canonical form of the value as represented by a character string literal.

If  $n$  is greater than that length, trailing pad characters are added to pad the canonical representation.

If  $n$  is too small, then a string truncation error is returned.

#### INTERVAL-to-VARCHAR Conversion

When the target data type is VARCHAR( $n$ ), then  $n$  must be equal to or greater than the length of the canonical form of the value as represented by a varying character string literal.

If  $n$  is too small, then a string truncation error is returned.

### Syntax

```
interval_expression (
  [ data_attribute [, ...] ]
  character_data_type
  [, { data_attribute | CHARACTER SET server_character_set } [, ...] ]
)
```

## Syntax Elements

### *interval\_expression*

An INTERVAL expression to be converted.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

### *character\_data\_type*

The data type to which the expression is to be converted.

### *server\_character\_set*

The server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default server character set is used.

## INTERVAL-to-INTERVAL Conversion

You can convert an INTERVAL value to a different INTERVAL value with either the CAST statement or Teradata conversion syntax.

## INTERVAL-to-INTERVAL Conversion with CAST

### ANSI Compliance

As an extension to ANSI, CAST permits the use of data attribute phrases.

### Syntax

```
CAST (
  interval_expression AS
  { interval_data_type [ data_attribute } |
    [ data_attribute [...] ]
  }
)
```

## Syntax Elements

### *interval\_expression*

An INTERVAL expression to be converted.

### *interval\_data\_type*

An INTERVAL data type to which the expression is to be converted.

### *data\_attribute*

One of the following data attributes:

- NAMED
- TITLE

## Teradata INTERVAL-to-INTERVAL Conversion Syntax

### ANSI Compliance

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

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

### Syntax

```
interval_expression (
  { interval_data_type [, data_attribute ][,...] |

    data_attribute [...]
    [, interval_data_type [, data_attribute ][,...] ] ]
}
```

## Syntax Elements

### *interval\_expression*

An INTERVAL expression to be converted.

### *interval\_data\_type*

An INTERVAL data type to which the expression is to be converted.

***data\_attribute***

One of the following data attributes:

- NAMED
- TITLE

## Usage Notes

### Compatible Types

Both data types must be from the same INTERVAL family: either Year-Month or Day-Time. Types cannot be mixed.

This INTERVAL data type ...	Belongs to this INTERVAL family ...
<ul style="list-style-type: none"> <li>• INTERVAL YEAR</li> <li>• INTERVAL YEAR TO MONTH</li> <li>• INTERVAL MONTH</li> </ul>	Year-Month
<ul style="list-style-type: none"> <li>• INTERVAL DAY</li> <li>• INTERVAL DAY TO HOUR</li> <li>• INTERVAL DAY TO MINUTE</li> <li>• INTERVAL DAY TO SECOND</li> <li>• INTERVAL HOUR</li> <li>• INTERVAL HOUR TO MINUTE</li> <li>• INTERVAL HOUR TO SECOND</li> <li>• INTERVAL MINUTE</li> <li>• INTERVAL MINUTE TO SECOND</li> <li>• INTERVAL SECOND</li> </ul>	Day-Time

Conversion of INTERVAL types is performed only when the fields and precisions are different.

### Precision of Source and Target Types

A conversion can result in an overflow error if the precision of the target data type is smaller than the corresponding precision for the source data type.

If the least significant value of the source is lower than that of the target, then those source values having lower precision than the least significant field of the target are ignored. The result is truncation. Recovery from this action is installation-dependent.

If the most significant field in the source value has higher significance than the most significant field in the target value, then the higher order fields of the source are converted into a scalar value of the precision of the most significant field in the target, using the factors of 12 months per year, 24 hours per day and so on.

If the compared scalar value overflows the defined precision for the target field, an error is returned.

## Implicit INTERVAL-to-INTERVAL Conversion

SQL Engine performs implicit conversion from INTERVAL to INTERVAL data types in some cases. See [Implicit Conversion of DateTime Types](#).

Conversion of INTERVAL types is performed only when both data types are from the same INTERVAL family: either Year-Month or Day-Time. See [Compatible Types](#).

## Examples

### Example: Least Significant Field in Source Lower Than Target

The following query converts '3-11' to '3'. Source is INTERVAL YEAR(2). The truncation completes the conversion.

```
SELECT CAST(INTERVAL '3-11' YEAR TO MONTH AS INTERVAL YEAR(2));
```

### Example: Least Significant Field in Source Lower Than Target

The following query converts '135 12:37:25.26' to '3252'. Source is DAY(3) TO SECOND(2)

```
SELECT CAST(INTERVAL '135 12:37:25.26' DAY(3) TO SECOND(2) AS INTERVAL HOUR(4));
```

### Example: Least Significant Field in Source Higher Than Target

The following query converts '3' to '3-00'. Source is INTERVAL YEAR. The insertion of zeros completes the conversion.

```
SELECT CAST(INTERVAL '3' YEAR AS INTERVAL YEAR TO MONTH);
```

### Example: Least Significant Field in Source Higher Than Target

The following query converts '135 00:00:00.0' to '3240:00:00.00' after you perform the additional conversion of multiplying  $135 * 24$  to obtain 3240, which is the final HOUR value. The source had a data type of DAY.

```
SELECT CAST(INTERVAL ' 135 00:00:00.0' DAY AS INTERVAL HOUR TO SECOND);
```



## Example: Most Significant Field in Source Higher Than Target

The following query first treats the source INTERVAL value as '135 12' and then computes HOURS as  $(135*24)+12=3252$ . The result of the query is INTERVAL '3252' HOUR unless the precision for the target value is less than 4, in which case an error is returned. The source had a data type of DAY TO SECOND.

```
SELECT CAST(INTERVAL '135 12:37:25.26' DAY TO SECOND AS INTERVAL HOUR);
```

## Example: Implicit Type Conversion During Assignment

Consider the following table which has an INTERVAL YEAR TO MONTH column:

```
CREATE TABLE TimeInfo
(YrToMon INTERVAL YEAR TO MONTH);
```

If you insert data into the column using the following parameterized request, and you pass an INTERVAL YEAR or INTERVAL MONTH value to the request, SQL Engine implicitly converts the value to an INTERVAL YEAR TO MONTH value before inserting the value.

```
INSERT INTO TimeInfo
VALUES (?);
```

## INTERVAL-to-Numeric Conversion

You can convert an INTERVAL with only one field to an exact numeric data type with either the CAST statement or Teradata conversion syntax.

This numeric value is the value of the single numeric field in the INTERVAL record.

## INTERVAL-to-Numeric Conversion with CAST

### ANSI Compliance

As an extension to ANSI, CAST permits the use of data attribute phrases.

### Syntax

```
CAST (
  interval_expression AS numeric_data_type
  [ data_attribute [...] ]
)
```

## Syntax Elements

### *interval\_expression*

An INTERVAL expression to be converted.

### *numeric\_data\_type*

The data type to which the expression is to be converted.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Teradata INTERVAL-to-Numeric Conversion Syntax

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

### Syntax

```
interval_expression (
  [ data_attribute, [...] ]
  numeric_data_type
  [ , data_attribute [, ...] ]
)
```

## Syntax Elements

### *interval\_expression*

An INTERVAL expression to be converted.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED

- TITLE

### *numeric\_data\_type*

The data type to which the expression is to be converted.

## Implicit INTERVAL-to-Numeric Conversion

SQL Engine performs implicit conversion of an Interval data type to an exact numeric data type in some cases. See [Implicit Conversion of DateTime Types](#).

## Example: Using CAST to Convert INTERVAL MONTH Values

Consider the following table definition:

```
CREATE TABLE sales_intervals
( sdate DATE
, sinterval INTERVAL MONTH
, stotals DECIMAL(5,0));
```

The following query uses CAST to convert INTERVAL MONTH values in the interval column to INTEGER.

```
SELECT stotals,
       (EXTRACT (MONTH FROM sdate)) + (CAST(sinterval AS INTEGER))
FROM sales_intervals;
```

## INTERVAL-to-UDT Conversion

Converts interval data to UDT data.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

### Syntax

```
CAST ( interval_expression AS UDT_data_type )
```

## Syntax Elements

### *interval\_expression*

An INTERVAL expression to be converted.

### *UDT\_data\_type*

The data type to which the expression is to be converted.

## Usage Notes

Explicit INTERVAL-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement.

## Implicit INTERVAL-to-UDT Conversion

Performing an implicit data type conversion requires a cast definition (see [Usage Notes](#)) that specifies the following:

- the AS ASSIGNMENT clause
- a source data type that is in the same INTERVAL family as the source of the implicit cast:

This INTERVAL data type ...	Belongs to this INTERVAL family ...
<ul style="list-style-type: none"> <li>• INTERVAL YEAR</li> <li>• INTERVAL YEAR TO MONTH</li> <li>• INTERVAL MONTH</li> </ul>	Year-Month
<ul style="list-style-type: none"> <li>• INTERVAL DAY</li> <li>• INTERVAL DAY TO HOUR</li> <li>• INTERVAL DAY TO MINUTE</li> <li>• INTERVAL DAY TO SECOND</li> <li>• INTERVAL HOUR</li> <li>• INTERVAL HOUR TO MINUTE</li> <li>• INTERVAL HOUR TO SECOND</li> <li>• INTERVAL MINUTE</li> <li>• INTERVAL MINUTE TO SECOND</li> <li>• INTERVAL SECOND</li> </ul>	Day-Time

The source data type of the cast definition does not have to be an exact match to the source of the implicit type conversion.

SQL Engine performs implicit INTERVAL-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

## Related Information

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

## Numeric-to-Character Conversion

You can convert a numeric value to a character value with either the CAST statement or Teradata conversion syntax.

## Numeric-to-Character Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

### Syntax

```
CAST (
    numeric_expression AS character_data_type
    [ data_attribute [...] ]
)
```

### Syntax Elements

#### *numeric\_expression*

The numeric data expression to be cast to a character type.

#### *character\_data\_type*

The data type to which the expression is to be converted.

#### *data\_attribute*

One of the following data attributes:

- CHARACTER SET

- FORMAT
- NAMED
- TITLE

## Teradata Numeric-to-Character Conversion Syntax

### ANSI Compliance

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

### Syntax

```
numeric_expression (
  [ data_attribute, [...] ]
  character_data_type
  [, { data_attribute | CHARACTER SET server_character_set } [,...] ]
)
```

### Syntax Elements

#### *numeric\_expression*

The numeric data expression to be cast to a character type.

#### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

#### *character\_data\_type*

The data type to which the expression is to be converted.

If *character\_data\_definition* does not specify a CHARACTER SET clause to indicate which server character set to use, the user default server character set is used.

#### *server\_character\_set*

The server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default server character set is used.

## Usage Notes

To convert a numeric type value to a character string, the character description must contain a data type declaration. A FORMAT phrase, by itself, cannot be used to convert a numeric type value to a character type value. The phrase only controls how to display the resultant value.

If the character description does not include a FORMAT phrase, then the format of the original numeric value determines how to display the data.

The Teradata conversion syntax form of numeric-to-character conversion uses explicit or default FORMATS to convert to a character representation. It then truncates or extends with pad characters, depending what length the character string dictates. This can lead to a loss of significance.

Attempting to convert from a numeric type to a character type that uses a GRAPHIC server character set generates an error.

As a general rule, you should store numbers as numeric data, not as character data. For example, a table is created with the following code:

```
CREATE TABLE job AS
  (job_code CHAR(6) PRIMARY KEY
   ,description CHAR(70) );
```

Subsequently, the following query is made:

```
SELECT job_code, description
FROM job
WHERE job_code = 1234;
```

The problem here is that '1234', '1234', '01234', '001234', '+1234', and so on, are all valid character representations of the numeric literal value, and the system cannot tell which value to use for hashing. Therefore, the system must do a full table scan to convert all job\_code values to their numeric equivalents so that it can do the comparisons.

## How CAST Differs from Teradata Conversion Syntax

The process for the CAST function is as follows:

1. Convert the numeric value to a character string using the default or specified format for the numeric value.
2. Trim leading and trailing pad characters.
3. Extend to the right as required by the target string length.
4. If truncation of non-pad characters is required to conform to the target string length, report string truncation error.

The CAST operation differs from the Teradata SQL conversion as follows:

- Results are left justified. Column displays are not aligned.
- Truncation of significant data generates a string truncation error.

Using Teradata conversion syntax (that is, not using CAST) for explicit conversion of *numeric* -to- *character* data requires caution.

The process is as follows:

1. Convert the numeric value to a character string using the default or specified FORMAT for the numeric value.

Leading and trailing pad characters are not trimmed.

2. Extend to the right with pad characters if required, or truncate from the right if required, to conform to the target length specification.

If non-pad characters are truncated, no string truncation error is reported.

## Supported Character Types

Numeric to character conversion is supported for CHAR and VARCHAR types only. Numeric types cannot be converted to CLOB types.

## Implicit Numeric-to-Character Conversion

If a numeric argument in an SQL string function is implicitly converted to a CHAR or VARCHAR character type, and the format of the numeric argument includes any of the following formatting characters, the server character set of the character type is UNICODE:

<ul style="list-style-type: none"> <li>• G</li> <li>• F</li> <li>• O</li> <li>• A</li> <li>• D</li> </ul>	<ul style="list-style-type: none"> <li>• L</li> <li>• U</li> <li>• I</li> <li>• C</li> <li>• N</li> </ul>
---	---

For all other formats, the server character set is LATIN.

Numeric items cannot be converted to CLOB types or GRAPHIC characters.

## Examples

### Example: Converting an INTEGER Data Type to a Character

T1.Field1 has a numeric INTEGER data type with the default format '-(10)9'. The user has values such as 123456, with no values of over 999999. The values, defined as being in INTEGER format, are to be converted to CHAR(8).



The following example illustrates the Teradata syntax for performing this numeric-to-character conversion.

```
SELECT Field1(CHAR(8)) FROM T1;
```

returns ' 123' for the value 123456, where the result includes 5 leading pad characters and truncates significant digits.

## Example: Converting Salaries

Based on the following description of Salary, data is converted as illustrated in the following table (Δ = pad character):

```
Salary (DECIMAL(8,2), FORMAT '$$$,$$9.99')
```

Data	Conversion	Result
20000.00	Salary (CHAR(10))	'\$20,000.00'
9000.00	Salary (CHAR(10))	'Δ \$9,000.00'
20000.00	Salary (FORMAT'9(5)') (CHAR (5))	'20000'
9000.00	CAST (Salary AS CHAR(10))	'\$9,000.00Δ '

The resultant character string is either extended with pad characters or truncated to conform to the given character description.

## Example: Converting Employee Numbers

Suppose EmpNo was defined as SMALLINT with the default format of '9(6)'. Suppose a value in EmpNo is 12501. The statement:

```
SELECT EmpNo(CHAR(5)) FROM Employee;
```

returns the '1250', with a leading pad character and the low order digit missing. The CAST function used for the same conversion, converts to the character representation of the numeric value, trims leading pad characters, and finally truncates or pads on the right. For example, the following SELECT statement returns '12501'.

```
SELECT CAST (EmpNo AS CHAR(5)) FROM Employee;
```

## Related Information

- For an example of numeric to character conversion that results in truncation of significant data, see [Example: Converting an INTEGER Data Type to a Character](#).
- For information on data type formats, formatting characters, and the FORMAT phrase, see [Data Type Formats and Format Phrases](#).

## Numeric-to-DATE Conversion

You can convert a numeric expression to a DATE value with either the CAST statement or Teradata conversion syntax.

### Numeric-to-DATE Conversion with CAST

#### ANSI Compliance

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

#### Syntax

```
CAST (
  numeric_expression AS DATE
  [ data_attribute [...] ]
)
```

#### Syntax Elements

##### *numeric\_expression*

An expression or existing field having a numeric data type.

##### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Teradata Numeric-to-DATE Conversion Syntax

#### ANSI Compliance

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

Syntax

```
numeric_expression ( [ data_attribute, [...] ] DATE [, data_attribute [,...] ] )
```

Syntax Elements

**numeric\_expression**  
An expression or existing field having a numeric data type.

**data\_attribute**  
One of the following data attributes:

- CHARACTER SET
- FORMAT
- NAMED
- TITLE

Specifying a FORMAT clause enables an alternative format.

Usage Notes

Translation of Numbers to Dates

Although not recommended, you can explicitly convert numbers to dates.  
SQL Engine stores each DATE value as a four-byte integer using the following formula:

```
(year - 1900) * 10000 + (month * 100) + day
```

For example, December 31, 1985 would be stored as the integer 851231; July 4, 1776 stored as -1239296; and March 30, 2041 stored as 1410330.

The following table demonstrates how numeric dates are interpreted when inserted into a column. Note the translation of the third date, which was probably intended to be 1990-12-01.

This numeric value ...	Translates to this date value ...
901201	1990-12-01
1001201	2000-12-01
19901201	3890-12-01

Notice that this formula best fits two-digit dates in the 1900s. Because of the difficulty of using this format outside of the 1900s, dates are best specified as ANSI date literals instead.

## Range of Allowable Values

Allowable date values range from AD January 1, 0001 (-18989899) to AD December 31, 9999 (80991231).

If the numeric value does not represent a valid date, an error is reported.

## Numeric-to-DATE Implicit Type Conversion

Although not recommended, you can specify a numeric type in the assignment of a DATE type. SQL Engine performs implicit numeric-to-DATE type conversion prior to the assignment. The value of the numeric type must represent a valid date.

However, for comparison operations involving a numeric type operand and a DATE type operand, Vantage converts the DATE type to a numeric type. If you compare a numeric type and a DATE type and expect the comparison to be between two DATE types, you must explicitly convert the numeric type to a DATE type.

## Example: Converting a Numeric Integer Expression to a Date Format

This example casts the numeric integer expression to a date format.

```
SELECT CAST (1071201 AS DATE);
```

The result looks like this when the DateForm mode of the session is set to ANSIDate:

```
1071201
-----
2007-12-01
```

## Related Information

- For implicit type conversion of operands for comparison operations, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.
- For data type compatibility rules for assignments involving DateTime types, see *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211.

## Numeric-to-INTERVAL Conversion

You can convert numeric data to an INTERVAL value with a single DateTime field with either the CAST statement or Teradata conversion syntax.

### Numeric-to-INTERVAL Conversion with CAST

#### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of interval data attribute phrases.

#### Syntax

```
CAST (
  numeric_expression AS interval_data_type
  [ data_attribute [...] ]
)
```

#### Syntax Elements

##### *numeric\_expression*

An expression or existing field having a numeric data type.

##### *interval\_data\_type*

The data type to which the expression is to be converted.

##### *data\_attribute*

One of the following data attributes:

- NAMED
- TITLE

## Teradata Numeric-to-INTERVAL Conversion Syntax

#### ANSI Compliance

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

#### Syntax

```
numeric_expression (
  [ data_attribute, [...] ]
  interval_data_type
```

```
[, data_attribute [,... ] ]
)
```

## Syntax Elements

### *numeric\_expression*

An expression or existing field having a numeric data type.

### *interval\_data\_type*

The data type to which the expression is to be converted.

### *data\_attribute*

One of the following data attributes:

- NAMED
- TITLE

## Usage Notes

Numeric data is converted to an INTERVAL value with a single DateTime field.

If the numeric value is in the value range allowed for the INTERVAL, the value is used as the single field of the INTERVAL. Otherwise, an overflow error is returned.

## Implicit Numeric-to-INTERVAL Conversion

SQL Engine performs implicit conversion of an exact numeric data type to an Interval data type in some cases. See [Implicit Conversion of DateTime Types](#).

## Example: Converting Numeric Data to an INTERVAL Value

The following query returns ' -5' (with three leading pad characters).

```
SELECT CAST(-5 AS INTERVAL YEAR(4));
```

## Numeric-to-Numeric Conversion

You can convert a numeric expression to a different numeric data type with either the CAST statement or Teradata conversion syntax.

## Numeric-to-Numeric Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI SQL, CAST permits data attributes such as the FORMAT phrase that enables an alternative format for *numeric\_expression*.

### Syntax

```
CAST ( numeric_expression AS
{
    numeric_data_type [ data_attribute [...] ] |
    [ data_attribute [...] ]
}
)
```

### Syntax Elements

#### *numeric\_expression*

An expression or existing field having a numeric data type.

#### *numeric\_data\_type*

The data type to which the expression is to be converted.

#### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Teradata Numeric-to-Numeric Conversion Syntax

### ANSI Compliance

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

### Syntax

```
numeric_expression (
{
```

```

    numeric_data_type
    [, data_attribute [, ...] ] |

    data_attribute [, ...]
    [, numeric_data_type [, data_attribute [, ...] ] ]
  }
)

```

## Syntax Elements

### *numeric\_expression*

An expression or existing field having a numeric data type.

### *numeric\_data\_type*

The data type to which the expression is to be converted.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Usage Notes

### Conversion to FLOAT/REAL/DOUBLE PRECISION

Because floating point numbers are not exact values, conversion of DECIMAL and integer values to FLOAT values might result in a loss of precision or produce a number that cannot be represented exactly. For example, a value like 0.1, when cast to FLOAT, no longer exactly equals to 0.1.

### Truncation and Rounding During Conversion

Conversion of DECIMAL/NUMERIC to BIGINT, INTEGER, BYTEINT, or SMALLINT truncates any decimal portion. Conversion to DECIMAL produces a rounded result. If a range violation occurs, the operation may fail.

Conversion to FLOAT/REAL/DOUBLE PRECISION rounds to the nearest value available. Neither decimal fractions nor numbers greater than 9,007,199,254,740,992 can be guaranteed to be represented exactly, so the nearest representable value is chosen. If there are two representable values that qualify as the nearest value, then the representation with a '0' in the least significant



bit is chosen. For example, 0.1, when stored in a FLOAT column, is rounded to a value slightly higher: 0.1000000000000000055511151231257827021181583404541015625.

For details on rounding, see [DECIMAL/NUMERIC Data Types](#).

Some examples of numeric conversions are:

Value	Converted To	Result
20000.99	INTEGER	20000
20000.99	DECIMAL(6,1)	20001.0
20000.99	DECIMAL(4, 1)	error
200000	SMALLINT	error

## Using CAST in Applications With DECIMAL Type Size Restrictions

Some applications require DECIMAL types to have 15 digits or less.

Applications with this requirement may need to access DECIMAL columns that have more than 15 digits or use expressions that may produce DECIMAL results with more than 15 digits. To help with DECIMAL type size requirements, you can use CAST to convert DECIMAL types to a size of 15 or fewer digits.

For example, consider the following expression where A, B, and C are columns defined as DECIMAL(8,2):

```
SELECT (A*B)/C FROM table1;
```

The resulting value may be less than 15 digits, but A\*B could be up to 18.

To ensure a result of less than 16 digits, use CAST:

```
SELECT CAST ((A*B)/C AS DECIMAL(15,2)) FROM table1;
```

## Using CAST To Avoid Numeric Overflow

Because of the way the Teradata SQL compiler works, it is essential that you CAST the arguments of your expressions whenever large values are expected.

For example, suppose f1 is defined as DECIMAL(14,2) and you are going to multiply by an integer or get SUM(f1).

In this case, the following operations:

```
CAST(f1 AS DECIMAL(18,2))*100
```

or

```
SUM(CAST(f1 AS DECIMAL(18,2)))
```

are proper techniques for ensuring correct answer sets.

On the other hand, if you were to cast the results of the expressions, such as the following:

```
CAST(f1*100 AS DECIMAL(18,2))
```

or

```
CAST(SUM(f1) AS DECIMAL(18,2))
```

then you will likely experience overflow during the computations (and before the CAST is made)—not the desired result.

## Implicit Numeric-to-Numeric Conversion

Numeric items are converted to the same numeric type before any arithmetic or comparison operation is performed. The result returned is of this same underlying type.

For example, before an INTEGER value is added to a FLOAT value, the INTEGER value is converted to FLOAT, the data type of the result.

## Examples

### Example: Casting a Numeric Integer Expression

This example casts the numeric integer expression named IntegerField to DECIMAL(7,2).

```
CAST (IntegerField AS DECIMAL (7,2))
```

### Example: Changing the FORMAT Phrase to Display a Numeric Value

Although the FORMAT phrase cannot be used to change the underlying data type defined for a column, the phrase may be used to change the display for a numeric value.

For example, if the field values for columns Wholesale and Retail, both defined as DECIMAL(7,2), are 12467.75 and 21500.50, respectively, the result of the expression:

```
CAST (Wholesale - Retail AS FORMAT '-99999')
```

is:

```
-09033
```

A FORMAT phrase does not affect data that is returned to the client system in Record Mode (client system internal format).

In the previous example, the value returned to the client system is still in packed decimal format (for example, -9032.75).

The use of FORMAT in CAST is a Teradata extension to the ANSI standard.

## Related Information

- For details on implicit type conversions for binary arithmetic expressions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.
- For details on implicit type conversions for comparison operations, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Numeric-to-UDT Conversion

Converts numeric data to UDT data.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

### Syntax

```
CAST ( numeric_expression AS UDT_data_type )
```

### Syntax Elements

#### *numeric\_expression*

A numeric expression to be cast to a UDT.

#### *UDT\_data\_type*

The UDT type, followed by any FORMAT, NAMED or TITLE data attribute phrases, to which the expression is to be converted.

## Usage Notes

Explicit numeric-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Implicit Numeric-to-UDT Conversion

SQL Engine performs implicit Numeric-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see [Usage Notes](#)) exists that specifies the AS ASSIGNMENT clause.

The source numeric type of the cast definition does not have to be an exact match to the source numeric type of the implicit conversion. Vantage can use an implicit cast definition that specifies a BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, NUMBER, or REAL/FLOAT/DOUBLE target type.

If multiple implicit cast definitions exist for converting different numeric types to the UDT, SQL Engine uses the implicit cast definition for the numeric type with the highest precedence. The following list shows the precedence of numeric types in order from lowest to highest precedence:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- DECIMAL/NUMERIC
- NUMBER
- REAL/FLOAT/DOUBLE

If no numeric-to-UDT implicit cast definitions exist, SQL Engine looks for other cast definitions that can substitute.

IF the following combination of implicit cast definitions exists ...		THEN SQL Engine ...
DATE-to-UDT	Character-to-UDT	
X		uses the DATE-to-UDT implicit cast definition.
	X	uses the character-to-UDT implicit cast definition. The character type cannot be CLOB. If multiple character-to-UDT implicit cast definitions exist, then SQL Engine returns an SQL error.

IF the following combination of implicit cast definitions exists ...		THEN SQL Engine ...
DATE-to-UDT	Character-to-UDT	
X	X	reports an error.

Substitutions are valid because Vantage can implicitly cast a numeric type to the substitute data type, and then use the implicit cast definition to cast from the substitute data type to the UDT.

## Period-to-Character Conversion

You can convert a period value to its canonical character string representation with either the CAST statement or Teradata conversion syntax.

Period-to-character conversion is supported for CHAR and VARCHAR types only. The target type cannot be CLOB.

## Period-to-Character Conversion with CAST

### ANSI Compliance

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

As an extension to ANSI, CAST permits the use of character data attribute phrases.

### Syntax

```
CAST (
  period_expression AS character_data_type
  [ CHARACTER SET server_character_set ]
  [ data_attribute [...] ]
)
```

### Syntax Elements

#### *period\_expression*

The Period expression to be converted.

#### *character\_data\_type*

The data type to which the expression is to be converted.

***server\_character\_set***

The server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default server character set is used.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Teradata Period-to-Character Conversion Syntax

```
period_expression (
  [ data_attribute, [...] ]
  character_data_type
  [, { data_attribute | CHARACTER SET server_character_set } [,... ] ]
)
```

***period\_expression***

The Period expression to be converted.

***character\_data\_type***

The data type to which the expression is to be converted.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

***server\_character\_set***

The server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default server character set is used.

## Usage Notes

A Period expression can be cast as a character string representation using the CAST function or the Teradata cast syntax, or when forming the output for field mode. Assume  $L$  is the maximum length of the formatted character string for the format associated with the Period expression being cast. The resulting character string contains two strings representing the beginning and ending bounds of the Period expression, each up to length  $L$ , and each enclosed in apostrophes (' '), separated by comma and a space ( , ), and then enclosed within a left parenthesis and a right parenthesis [( )]. Thus, the maximum length of the resulting character string is  $2*L + 8$ . Assume the actual length is  $K$  (which may be less than  $2*L + 8$ , for example, if the format includes the full names of months and the specific month for a bound is July) and the target type is CHARACTER( $n$ ) or VARCHAR( $n$ ):

- If  $n$  is equal to  $K$ , the period is cast into the resulting character string of length  $K$ .
- If  $n$  is greater than  $K$  and the target is VARCHAR( $n$ ), the period is cast into the resulting character string with length  $K$ .
- If  $n$  is greater than  $K$  and the target is CHARACTER( $n$ ), the period is cast into the resulting character string and trailing pad characters are added to extend to length  $n$ .
- If  $n$  less than  $K$  and the session is in ANSI mode, a truncation error is reported.
- If  $n$  less than  $K$  and the session is in Teradata mode, a truncated string of length  $n$  is returned.

For data of Period data types with TIME and TIMESTAMP element types, the UTC value of the Period expression is adjusted to the time zone of the value or the current session time zone if the value does not have a time zone. The exception to conversion from UTC is for an ending bound of a PERIOD(TIMESTAMP( $n$ )) value equal to the maximum value that is used to represent UNTIL\_CHANGED; in this case, the value is not changed. Due to such adjustments, the ending bound may appear less than the beginning bound in the result, although in UTC the ending bound is greater than the beginning bound. This happens since the hour value for the TIME data type wraps over every 24 hours (that is, the hour value is obtained using 'module 24').

## Example: Converting a PERIOD Data Type to Its Canonical Character String Representation

Assume pts is a PERIOD(TIMESTAMP(2)) column in table t with a value of PERIOD '(2005-02-02 12:12:12.34, 2006-02-03 12:12:12.34)'.

In the following example, a PERIOD(TIMESTAMP(2)) column is cast as CHARACTER(52) using the CAST function.

```
SELECT CAST(pts AS CHARACTER(52)) FROM t;
```

The following is returned:

```
('2005-02-02 12:12:12.34', '2006-02-03 12:12:12.34')
```

## Period-to-DATE Conversion

Converts a period value to a DATE value.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of DATE data attribute phrases.

### Syntax

```
CAST (
  period_expression AS DATE
  [ data_attribute [...] ]
)
```

### Syntax Elements

#### *period\_expression*

The Period expression to be converted.

#### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Usage Notes

A PERIOD(DATE) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as DATE using the CAST function. The source last value must be equal to the source beginning bound; otherwise, an error is reported.

If the source type is PERIOD(DATE), the result is the source beginning bound.

If the source type is PERIOD(TIMESTAMP(n) [WITH TIME ZONE]), the result is the date portion of the source beginning bound after adjusting to the current session time zone.

If the source type is PERIOD(TIME(n) [WITH TIME ZONE]), an error is reported.



## Example: Converting Period Data to a DATE Value

Assume `pd` is a `PERIOD(DATE)` column in table `t` with a value of `PERIOD '(2005-02-02, 2005-02-03)'`.

In the following example, a `PERIOD(DATE)` column is cast as `DATE`. The result is the beginning bound of the column.

```
SELECT CAST(pd AS DATE) FROM t;
```

The following is returned:

```
2005-02-02
```

## Period-to-Period Conversion

Converts a period value to a different period value.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, `CAST` permits the use of data attribute phrases.

### Syntax

```
CAST (
  period_expression AS
  { period_data_type [ data_attribute [...] ] |
    [ data_attribute [...] ]
  }
)
```

### Syntax Elements

#### *period\_expression*

The Period expression to be converted.

#### *period\_data\_type*

The data type to which the expression is to be converted.

#### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Usage Notes

### Compatible Types

The following table describes the allowed combinations of source and target types when both the source and the target types are Period data types.

Source Type	Target Type
PERIOD(DATE)	PERIOD(DATE)
	PERIOD(TIMESTAMP[( <i>m</i> )] [WITH TIME ZONE])
PERIOD(TIME[( <i>n</i> )] [WITH TIME ZONE])	PERIOD(TIME[( <i>m</i> )] [WITH TIME ZONE]) where <i>m</i> is the target precision, <i>m</i> must be greater than or equal to the source precision <i>n</i> . The default for <i>m</i> is 6.
	PERIOD(TIMESTAMP[( <i>m</i> )] [WITH TIME ZONE]) where <i>m</i> is the target precision, <i>m</i> must be greater than or equal to the source precision <i>n</i> . The default for <i>m</i> is 6.
PERIOD(TIMESTAMP[( <i>n</i> )] WITH TIME ZONE)	PERIOD(DATE)
	PERIOD(TIME[( <i>m</i> )] [WITH TIME ZONE]) where <i>m</i> is the target precision, <i>m</i> must be greater than or equal to the source precision <i>n</i> . The default for <i>m</i> is 6.
	PERIOD(TIMESTAMP[( <i>m</i> )] [WITH TIME ZONE]) where <i>m</i> is the target precision, <i>m</i> must be greater than or equal to the source precision <i>n</i> . The default for <i>m</i> is 6.

### PERIOD(DATE) to PERIOD(TIMESTAMP)

A PERIOD(DATE) value can be cast as PERIOD(TIMESTAMP[(*n*)] [WITH TIME ZONE]) using the CAST function.

The UTC value of the result elements are obtained after adjustment with respect to the current session time zone from the timestamps created by setting the date portion to the corresponding source elements and the time portions to 0. If the target type is PERIOD(TIMESTAMP[(*n*)] WITH TIME ZONE), both result time zone fields are set to the current session time zone displacement. An exception to this is if the source

ending bound is the maximum DATE value; in that case, the result ending bound is set to the maximum TIMESTAMP value.

## PERIOD(TIME) to PERIOD(TIME)

A PERIOD(TIME(n) [WITH TIME ZONE]) value can be cast as PERIOD(TIME[(n)] [WITH TIME ZONE]) using the CAST function.

The UTC value of the source is copied to the UTC value in the result. If the target type specifies WITH TIME ZONE and the source contains time zones, the time zone displacements from the source are copied to the corresponding result elements. If the source does not contain time zones, the current session time zone displacement is copied to both result elements. For example, assume the current session time zone displacement is INTERVAL - "08:00" HOUR TO MINUTE and the source PERIOD(TIME(0) WITH TIME ZONE) has the value PERIOD ('(12:12:12+08:00, 12:12:13+08:00)'). The UTC value of this source is ('04:12:12', '04:12:13'). The UTC value of the result is set to this value. On output of this result, the UTC value is adjusted to the current session time zone and the result is ('20:12:12', '20:12:13').

---

### Note:

This value is actually for a previous day and, assuming that the CURRENT\_DATE at UTC is DATE '2006-07-28', the output beginning bound would be '2006-07-27 20:12:12' if it was a timestamp element.

---

If the target precision is higher than the source precision, trailing zeros are appended to the fractional seconds. If the target precision is lower than the source precision, an error is reported.

## PERIOD(TIME) to PERIOD(TIMESTAMP)

A PERIOD(TIME(n) [WITH TIME ZONE]) value can be cast as PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function.

The source time values get adjusted with respect to the session time zone displacement from the corresponding UTC value. The date portion of each result element is set to CURRENT\_DATE. The hour, minute, and, second are copied from the source after the above adjustment and the timestamp value is converted to corresponding UTC value.

If the target type specifies WITH TIME ZONE and the source contains time zones, the time zone displacements from the source are copied to the corresponding result elements. If the source does not contain time zones, the current session time zone displacement is copied to both result elements.

If the target precision is higher than the source precision, trailing zeros are appended to the fractional seconds. If the target precision is lower than the source precision, an error is reported.

## **PERIOD(TIMESTAMP) to PERIOD(DATE)**

A PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as PERIOD(DATE) using the CAST function.

The result elements are each set to the date portion of the corresponding source bound after the source bound is adjusted according to the current session time zone (the adjustment is not done for the source ending bound if it is the maximum value). If the adjustment for time zone changes the date, the changed value is used. If the result date portions are the same, an error is reported.

## **PERIOD(TIMESTAMP) to PERIOD(TIME)**

A PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as PERIOD(TIME[(n)] [WITH TIME ZONE]) using the CAST function.

The date portion in the beginning and ending UTC values of the source must have the same DATE value. Otherwise, an error is reported. The time portions of the result elements are copied from the corresponding source time portions. If the target type specifies WITH TIME ZONE and the source also contains time zones, the source time zone displacements are copied to the corresponding result elements. If the source does not contain time zones, the current session time zone displacement is copied to both result elements.

If the target precision is higher than the source precision, trailing zeros are added to the fractional seconds. If the target precision is lower than the source precision, an error is reported.

## **PERIOD(TIMESTAMP) to PERIOD(TIMESTAMP)**

A PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function.

The result date and time portions are set to the corresponding source date and time portions. If the target type specifies WITH TIME ZONE and the source also contains time zones, the time zone displacements in the source are copied to the corresponding result elements. If the source does not contain time zones, the current session time zone displacement is copied to both result elements except if the source ending bound is the maximum value, the time zone for the result ending bound is +00:00.

If the target precision is higher than the source precision, trailing zeros are added in the fractional seconds. If the target precision is lower than the source precision, an error is reported.

## Examples

### Example: PERIOD(DATE) to PERIOD(TIMESTAMP)

Assume p is a PERIOD(DATE) column in table t1 with a value of PERIOD '(2005-02-02, 2006-02-03)' and the current session time zone displacement is INTERVAL '-08:00' HOUR TO MINUTE.

In the following example, a PERIOD(DATE) column is cast as PERIOD(TIMESTAMP(6)). The date portion is obtained from the source for the corresponding result element and the time portions are set to zero.

```
SELECT CAST(p AS PERIOD(TIMESTAMP(6))) FROM t1;
```

The following is returned:

```
('2005-02-02 00:00:00.000000', '2006-02-03 00:00:00.000000')
```

### Example: Least Significant Field in Source Lower Than Target

Assume p is a PERIOD(TIME(2)) column in table t with a value of PERIOD '(12:12:12.45, 13:12:12.67)' and the current session time zone displacement is INTERVAL '-08:00' HOUR TO MINUTE.

In the following example, a PERIOD(TIME(2)) column is cast as PERIOD(TIME(6) WITH TIME ZONE). The time portion is obtained from the source with trailing zeros added to the fractional seconds to make the precision 6 for the corresponding result element and both result time zone fields are set to the current session time zone displacement.

```
SELECT CAST(p AS PERIOD(TIME(6)WITH TIME ZONE)) FROM t;
```

The following is returned:

```
('12:12:12.450000-08:00', '13:12:12.670000-08:00')
```

## Period-to-TIME Conversion

Converts Period data to a TIME value.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of TIME data attribute phrases.

## Syntax

```
CAST (
  period_expression AS TIME
  [ ( fractional_seconds_precision ) ]
  [ WITH TIME ZONE ]
  [ data_attribute [...] ]
)
```

## Syntax Elements

### *period\_expression*

The Period expression to be converted.

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Usage Notes

A PERIOD(TIME(n) [WITH TIME ZONE]) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as TIME[(n)] [WITH TIME ZONE] using the CAST function. The source last value must be equal to the source beginning bound; otherwise, an error is reported.

If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision. If the target precision is lower than the source precision, an error is reported.

If the source type is PERIOD(TIME(n) [WITH TIME ZONE]) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]), the result time portion is obtained from time portion of the source beginning bound. If both the source and target type are WITH TIME ZONE, the result time zone field is set to the time zone displacement of the source beginning bound. If only the target type is WITH TIME ZONE, the result time zone field is set to the current session time zone displacement.

If the source type is PERIOD(DATE), an error is reported.

## Example

Assume `pt` is a `PERIOD(TIME(2))` column in table `t` with a value of `PERIOD '(12:12:12.34, 12:12:12.35)'`.

In the following example, a `PERIOD(TIME(2))` column is cast as `TIME(6)`. The `TIME(6)` result is obtained from the source beginning element with trailing zeros added to the fractional seconds to make the precision 6.

```
SELECT CAST(pt AS TIME(6)) FROM t;
```

The following is returned:

```
12:12:12.340000
```

## Period-to-TIMESTAMP Conversion

Converts a period value to a `TIMESTAMP` value.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, `CAST` permits the use of the `FORMAT` phrase to enable alternative output formatting of `DateTime` data.

### Syntax

```
CAST (
  period_expression AS TIMESTAMP
  [ ( fractional_seconds_precision ) ]
  [ WITH TIME ZONE ]
  [ data_attribute [...] ]
)
```

### Syntax Elements

#### *period\_expression*

The Period expression to be converted.

#### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the `SECOND` field. The valid range is 0 through 6. The default is 6.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Usage Notes

A `PERIOD(DATE)`, `PERIOD(TIME(n) [WITH TIME ZONE])`, or `PERIOD(TIMESTAMP(n) [WITH TIME ZONE])` value can be cast as `TIMESTAMP[(n)] [WITH TIME ZONE]` using the `CAST` function. The source last value must be equal to the source beginning bound; otherwise, an error is reported.

If the source type is `PERIOD(TIME(n) [WITH TIME ZONE])` or `PERIOD(TIMESTAMP(n) [WITH TIME ZONE])`:

- If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision.
- If the target precision is lower than the source precision, an error is reported.

If the source type is `PERIOD(DATE)`, the result is formed from the source beginning bound and a time portion of 0 adjusted with respect to the current session time zone, and, if the target type is `WITH TIME ZONE`, the current session time zone displacement.

If the source type is `PERIOD(TIME(n) [WITH TIME ZONE])`, the source beginning bound (in UTC) is adjusted with respect to the current session time zone displacement. The timestamp portion of the result is formed from `CURRENT_DATE` and the time portion of the source beginning bound obtained after the above adjustment. The resulting timestamp value is converted to UTC. If both the source and target type are `WITH TIME ZONE`, the result time zone field is set to the time zone displacement of the source beginning bound. If only the target type is `WITH TIME ZONE`, the result time zone field is set to the current session time zone displacement.

If the source type is `PERIOD(TIMESTAMP(n) [WITH TIME ZONE])`, the result timestamp portion is the timestamp portion of the source beginning bound. If both the source and target type are `WITH TIME ZONE`, the result time zone field is set to the time zone displacement of the source beginning bound. If only the target type is `WITH TIME ZONE`, the result time zone field is set to the current session time zone displacement.

## Example

Assume `pts` is a `PERIOD(TIMESTAMP(2))` column in table `t` with a value of `PERIOD '(2005-02-03 12:12:12.34, 2005-02-03 12:12:12.35)'`.

In the following example, column `pts` is cast as `TIMESTAMP(6)`. The result is the source beginning bound with trailing zeros added to the fractional seconds to make the precision 6.



```
SELECT CAST(pts AS TIMESTAMP(6)) FROM t;
```

The following is returned:

```
2005-02-03 12:12:12.340000
```

## Signed Zone DECIMAL Conversion

Teradata SQL can convert input data that is in signed zone (external) DECIMAL format to a NUMERIC data type, thus allowing numeric operations to be performed on row values. The column in which the signed zone decimal data is to be stored may be any numeric data type.

A FORMAT phrase incorporating the S sign character filters the data as it passes in and out of Vantage.

The rightmost character of the input data string is assumed to contain the zone (overpunch) bit.

The following table shows the characters representing zone-numeric combinations.

Last Character (Input String)	Numeric Conversion	Last Character (Input String)	Numeric Conversion	Last Character (Input String)	Numeric Conversion
{	n ... 0	}	-n ... 0	0	n ... 0
A	n ... 1	J	-n ... 1	1	n ... 1
B	n ... 2	K	-n ... 2	2	n ... 2
C	n ... 3	L	-n ... 3	3	n ... 3
D	n ... 4	M	-n ... 4	4	n ... 4
E	n ... 5	N	-n ... 5	5	n ... 5
F	n ... 6	O	-n ... 6	6	n ... 6
G	n ... 7	P	-n ... 7	7	n ... 7
H	n ... 8	Q	-n ... 8	8	n ... 8
I	n ... 9	R	-n ... 9	9	n ... 9

The sign FORMAT phrase can be included in a CREATE TABLE or ALTER TABLE statement when the column is defined, or in the INSERT statement when the data is loaded. The chosen method depends on how the stored value is to be used.

When a sign FORMAT phrase is specified at column creation time, it is considered attached to the column because it translates data at the column level; that is, both when the data is loaded and when it is retrieved.

## Using FORMAT in CREATE TABLE

When the FORMAT phrase is used in the CREATE TABLE statement, as follows:

```
CREATE TABLE Test1 (Col1 DECIMAL(4) FORMAT '9999S');
```

then zoned input character strings can be loaded with standard INSERT statements, whether the data is defined:

```
INSERT INTO Test1 (Col1) VALUES ('123J');
```

or read from a client system data record via the USING modifier:

```
USING Ext1 (CHAR(4))
INSERT INTO Test1 (Col1)
VALUES (:Ext1);
```

The data record contains the string '123J'.

Subsequently, a simple select, such as:

```
SELECT Col1 FROM Test1;
```

returns:

```
Col1
----
123J
```

## Using Another FORMAT in the SELECT Statement

To override an attached format, another FORMAT phrase is needed in the retrieval statement. Using the preceding table, one of the two following statements must be used to retrieve the numeric value:

```
SELECT Col1 (FORMAT '+9999') FROM Test1;
```

or

```
SELECT CAST (Col1 AS INTEGER) FROM Test1;
```

The result is as follows.

```
Col1
-----
-1231
```

## If FORMAT is Not Attached to the Column

If the format is not attached to the column, the sign FORMAT phrase must be used each time signed zoned decimal data is loaded and each time the row value is to be retrieved in signed zoned decimal format.

For example, if a table is defined using a CREATE TABLE statement like this:

```
CREATE TABLE Test2 (Col2 DECIMAL(5));
```

then the sign FORMAT phrase must be included whenever signed zoned decimal strings are inserted.

This is true whether the definition is explicitly defined, as it is in Examples 1 and 2, or defined implicitly by being read from a client system data record as it is in Examples 3 and 4.

## Examples

### Example

```
INSERT INTO Test2 (Col2)
VALUES ('5678B' (DECIMAL(5), FORMAT '99999S'));
```

### Example

```
INSERT INTO Test2 (Col2)
VALUES ('9012L' (DECIMAL(5), FORMAT '99999S'));
```

### Example

```
USING Ext2 (CHAR(5))
INSERT INTO Test2 (Col2)
VALUES (:Ext2 (DECIMAL(5), FORMAT '99999S'));
```

### Example

```
USING Ext2 (CHAR(5))
INSERT INTO Test2 (Col2)
VALUES (:Ext2 (DECIMAL(5), FORMAT '99999S'));
```

where Ext2 contains the strings '5678B' and '9012L'.

Because Col2 does not have an attached FORMAT phrase, a simple SELECT, such as the following example, returns the results as seen immediately following.

```
SELECT Col2 FROM Test2;
```

```
Col2
-----

 56782.
-90123.
```

A sign FORMAT phrase must be included in the SELECT statement in order to retrieve the values '5678B' and '9012L'.

It is important to remember this rule when manipulating signed zoned decimal values, especially when using sophisticated facilities like subqueries.

## Example

This example is based on the data from the previous example.

Consider a column created with a CHARACTER data type.

```
CREATE TABLE Test3 (Col3 CHAR(5));
```

The column is loaded by selecting, without a sign FORMAT phrase, values from an “unattached” column, as follows.

```
INSERT INTO Test3 (Col3)
SELECT Col2 FROM Test2 ;
```

The values that are inserted are the following:

```
Col3
-----

 5678
-9012
```

The sign FORMAT phrase *must* be included in the query specification in order to insert the values '5678B' and '9012L'.

## Related Information

For information on data types, data type formats, formatting characters, and the FORMAT phrase, see [Data Type Formats and Format Phrases](#).

## TIME-to-Character Conversion

You can convert a TIME value to a character string with either the CAST statement or Teradata conversion syntax..

### TIME-to-Character Conversion with CAST

#### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting for the character representations of DateTime data.

#### Syntax

```
CAST (
  time_expression AS character_data_type
  [ CHARACTER SET server_character_set ]
  [ data_attribute [...] ]
)
```

#### Syntax Elements

##### *time\_expression*

The TIME expression to be converted.

##### *character\_data\_type*

The data type to which the expression is to be converted.

##### *server\_character\_set*

The server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default server character set is used.

##### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Teradata TIME-to-Character Conversion Syntax

```
time_expression (
  [ data_attribute, [...] ]
  character_data_type
  [, { data_attribute | CHARACTER SET server_character_set } [,... ] ]
)
```

### ANSI Compliance

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

### Syntax

```
time_expression (
  [ data_attribute, [...] ]
  character_data_type
  [, { data_attribute | CHARACTER SET server_character_set } [,... ] ]
)
```

### Syntax Elements

#### *time\_expression*

The TIME expression to be converted.

#### *character\_data\_type*

The data type to which the expression is to be converted.

#### *server\_character\_set*

The server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default server character set is used.

#### *data\_attribute*

One of the following data attributes:

- FORMAT

- NAMED
- TITLE

## Usage Notes

When converting TIME to CHAR(*n*) or VARCHAR(*n*), then *n* must be equal to or greater than the length of the TIME value as represented by a character string literal.

IF the target data type is ...	AND <i>n</i> is ...	THEN ...
CHAR( <i>n</i> )	greater than the length of the TIME value as represented by a character string literal	trailing pad characters are added to pad the representation
	too small	a string truncation error is returned
VARCHAR( <i>n</i> )	greater than the length of the TIME value as represented by a character string literal	no blank padding is added to the character representation
	too small	a string truncation error is returned

TIME to CLOB conversion is not supported.

You cannot convert a TIME value to a character string when the server character set is GRAPHIC.

## Forcing a FORMAT on CAST for Converting TIME to Character

The default format for TIME to character conversion is the format in effect for the TIME value.

You can convert a TIME value to a character string using a FORMAT phrase. The resulting format, however, is the same as the TIME value. If you want a different format for the string value, you need to also use CAST as described here.

You must use nested CAST operations in order to convert values from TIME to CHAR and force an explicit FORMAT on the result regardless of the format associated with the TIME value. This is because of the rules for matching FORMAT phrases to data types.

## Example

Field T1 in the table INTTIME is a TIME(6) value with the explicit format 'HH:MM:SSDS(6)'. Assume that you want to convert this to a value of CHAR(6), and an explicit output format of 'HHhMIm'.

```
SELECT T1 FROM INTTIME ;
```

The result (without a type change) is the following report:

```

          T1
-----
05:57:11.362271

```

Now use nested CAST phrases and a FORMAT to obtain the desired result: a report in character format.

```

SELECT
  CAST( (CAST (T1 AS FORMAT 'HHhMim'))
  AS CHAR(6))
FROM INTTIME;

```

The result after the nested CASTs is the following report.

```

T1
-----
05h57m

```

The inner CAST establishes the display format for the TIME value and the outer CAST indicates the data type of the desired result.

## TIME-to-Period Conversion

Converts TIME data as PERIOD(TIME[(n)] [WITH TIME ZONE]) or PERIOD(TIMESTAMP[(n)])[WITH TIME ZONE]).

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

### Syntax

```

CAST (
  time_expression AS period_data_type
  [ data_attribute [...] ]
)

```

### Syntax Elements

#### *time\_expression*

The TIME expression to be converted.



***period\_data\_type***

The data type to which the expression is to be converted.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Usage Notes

A TIME(n) [WITH TIME ZONE] value can be cast as PERIOD(TIME[(n)] [WITH TIME ZONE]) or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function.

If the target precision is higher than the source precision, trailing zeros are added in the result bounds to adjust the precision. If the target precision is lower than the source precision, an error is reported.

If the TIME source value contains leap seconds, the seconds portion gets adjusted to 59.999999 with the precision truncated to the target precision.

If the target type is PERIOD(TIME[(n)] [WITH TIME ZONE]), the result beginning element is set to the source value (in UTC). If the target type is PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]), the source time value get adjusted with respect to the current session time zone displacement from the corresponding UTC value; the date portion in the result beginning element is set to CURRENT\_DATE, the time portion is set to the source value obtained after the above adjustment, and the resulting timestamp value is converted to UTC. If both the source and target are WITH TIME ZONE, the time zone field of the result beginning element is set to the source time zone field. If only the target has WITH TIME ZONE, the time zone field of the result beginning element is set to the current session time zone displacement. The result ending element is set to the result beginning bound plus one granule of the target type. If the result ending bound has a lower value than the result beginning bound for a target type of PERIOD(TIME[(n)] [WITH TIME ZONE]) or the result ending element value exceeds the maximum corresponding TIMESTAMP value for a target type of PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]), an error is reported.

---

**Note:**

If the target type is WITH TIME ZONE, the result beginning and ending bounds have the same time zones.

---

Also, note that the result has the same value for the beginning bound and last value.

## Example

Assume pt is a TIME(0) column in table t with a value of TIME '12:12:12' and the current session time zone displacement is INTERVAL '-08:00' HOUR TO MINUTE.

In the following example, a TIME(0) column is cast as PERIOD(TIME(4) WITH TIME ZONE). The result beginning bound is formed from the source (in UTC) with trailing zeros added to make the precision 4 and the current session time zone displacement. The result ending element is set to the result beginning bound plus INTERVAL '0.0001' SECOND.

---

**Note:**

The time zones of the result beginning and ending elements are the same.

```
SELECT CAST(pt AS PERIOD(TIME(4) WITH TIME ZONE)) FROM t;
```

---

Returns a PERIOD(TIME(4) WITH TIME ZONE) value as follows:

```
('12:12:12.0000-08:00', '12:12:12.0001-08:00')
```

## TIME-to-TIME Conversion

You can convert a TIME or TIME WITH TIME ZONE value to a TIME or TIME WITH TIME ZONE value using optional data attributes, with either the CAST statement or Teradata conversion syntax.

### TIME-to-TIME Conversion with CAST

#### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

#### Syntax

```
CAST (
  time_expression AS TIME

  [ ( fractional_seconds_precision ) ]

  [ WITH TIME ZONE ]

  [ AT { LOCAL | SOURCE [ TIME ZONE ] |
        [ TIME ZONE ] { expression | time_zone_string }
    }
  ]

  [ data_attribute [...] ]
)
```

## Syntax Elements

### *time\_expression*

The TIME expression to be converted.

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

### AT LOCAL

Use the time zone displacement based on the current session time zone.

### AT SOURCE [TIME ZONE]

Use the time zone associated with *timestamp\_expression* when:

- AT SOURCE TIME ZONE is specified.
- AT SOURCE is specified without TIME ZONE and there is no column named source in the scope.

Otherwise, if AT SOURCE is specified without TIME ZONE and a column named source exists, then SOURCE references this column, and the value of the column is used as the time zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE.

### 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.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting for DateTime data.

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using CAST to convert from TIME (with or without time zone) to TIME WITH TIME ZONE. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

---

**Note:**

TIME (without time zone) is not ANSI SQL:2011 compliant. Vantage internally converts a TIME value to UTC based on the current session time zone or on a specified time zone.

---

## Teradata TIME-to-TIME Conversion Syntax

### ANSI Compliance

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

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using Teradata conversion syntax to convert from TIME (with or without time zone) to TIME WITH TIME ZONE. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

---

**Note:**

TIME (without time zone) is not ANSI SQL:2011 compliant. Vantage internally converts a TIME value to UTC based on the current session time zone or on a specified time zone.

---

### Syntax

```
time_expression (
  [ data_attribute [, ...] ] TIME

  [ ( fractional_seconds_precision ) ]

  [, WITH TIME ZONE ]

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

```
[, data_attribute [,... ] ]
)
```

## Syntax Elements

### *time\_expression*

The TIME expression to be converted.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

### AT LOCAL

Use the time zone displacement based on the current session time zone.

### AT SOURCE [TIME\_ZONE]

Use the time zone associated with timestamp\_expression when:

- AT SOURCE TIME\_ZONE is specified.
- AT SOURCE is specified without TIME\_ZONE and there is no column named source in the scope.

Otherwise, if AT SOURCE is specified without TIME\_ZONE and a column named source exists, then SOURCE references this column, and the value of the column is used as the time zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE.

### 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.

**Usage Notes**

If you specify an AT clause for a TIME[(n)] without time zone target data type, an error is returned.

If you specify an AT clause for a TIME[(n)] WITH TIME ZONE target data type, the following table shows the result of the CAST function or Teradata conversion based on the various options specified. If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision. If the target precision is lower than the source precision, an error is returned.

IF you specify...	AND the data type of <i>time_expression</i> is...	THEN...
AT LOCAL	with or without TIME ZONE	THEN...
AT SOURCE (where SOURCE is a keyword and not a column reference)	WITH TIME ZONE	the result is formed from the source <i>time_expression</i> (in UTC) and the time zone displacement based on the current session time zone. If the data type of <i>time_expression</i> is without time zone, this is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	without TIME ZONE	the result is formed from the time portion of the source <i>time_expression</i> (in UTC) and the time zone displacement associated with <i>time_expression</i> . Note that this is the same as not specifying the AT clause.
AT SOURCE TIME ZONE	WITH TIME ZONE	the result is formed from the time portion of the source <i>time_expression</i> (in UTC) and the time zone displacement associated with <i>time_expression</i> . Note that this is the same as not specifying the AT clause.
AT SOURCE TIME ZONE	without TIME ZONE	an error is returned.
AT expression or AT TIME ZONE <i>expression</i>	with or without TIME ZONE	the result is formed from the time portion of the source <i>time_expression</i> (in UTC) and the time zone displacement defined by <i>expression</i> .
AT <i>time_zone_string</i> or AT TIME ZONE <i>time_zone_string</i>	with or without TIME ZONE	the result is formed from the time portion of the source <i>time_expression</i> (in UTC) and the time zone displacement based on <i>time_zone_string</i> . The time zone displacement is determined based on <i>time_zone_string</i> , CURRENT_TIMESTAMP AT '00:00', and the TIME value of <i>time_expression</i> at UTC.

## Examples

### Example

In this example, the current session time zone displacement, INTERVAL '01:00' HOUR TO MINUTE, is used to determine the UTC value, '07:30:00' of the TIME literal.

The result of the CAST is the time formed from the time portion of the source expression value '07:30:00' at UTC and the current time zone displacement, INTERVAL '01:00' HOUR TO MINUTE.

The result value of the CAST '07:30:00' at UTC is adjusted to its time zone displacement, INTERVAL '01:00' HOUR TO MINUTE, and the result of the SELECT statements is: TIME '08:30:00+01:00'.

The result of the SELECT statements is equal to TIME '07:30:00+00:00' since values are compared based on their UTC values.

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;

SELECT CAST(TIME '08:30:00' AS TIME(0) WITH TIME ZONE);
SELECT CAST(TIME '08:30:00' AS TIME(0) WITH TIME ZONE AT LOCAL);
```

### Example

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '04:30:00' for the TIME literal.

The result of the CAST is the time formed from the time portion of the source expression value '04:30:00' at UTC and the current session time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE.

The result value of the CAST '04:30:00' at UTC is adjusted to its time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIME '20:30:00-08:00'.

The result of the SELECT statement is equal to TIME '04:30:00+00:00'.

```
SET TIME ZONE INTERVAL '-08:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIME(0)
  WITH TIME ZONE AT LOCAL);
```

### Example

The following SELECT statement returns an error because the source expression does not have a time zone displacement.

```
SELECT CAST(TIME '08:30:00' AS TIME(0)
  WITH TIME ZONE AT SOURCE TIME ZONE);
```

## Example

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '04:30:00' for the TIME literal.

The result of the CAST is the time formed from the time portion of the source expression value '04:30:00' at UTC, and the time zone displacement of the source expression, INTERVAL '04:00' HOUR TO MINUTE.

The result value of the CAST '04:30:00' at UTC is adjusted to its time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, and the result of the SELECT statements is: TIME '08:30:00+04:00'.

The result of the SELECT statements is equal to TIME '04:30:00+00:00'. The current session time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, has no effect.

```
SET TIME ZONE INTERVAL '-08:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIME(0) WITH TIME ZONE);
SELECT CAST(TIME '08:30:00+04:00' AS TIME(0)
  WITH TIME ZONE AT SOURCE);
```

## Example

In this example, the current session time zone displacement, INTERVAL '-04:00' HOUR TO MINUTE, is used to determine the UTC value '12:30:00' for the TIME literal.

The result of the CAST is the time formed from the time portion of the source expression value '12:30:00' at UTC, and the specified time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE.

The result value of the CAST '12:30:00' at UTC is adjusted to its time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIME '04:30:00-08:00'.

The result of the SELECT statement is equal to TIME '12:30:00+00:00'.

```
SET TIME ZONE INTERVAL '-04:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00' AS TIME(0) WITH TIME ZONE AT -8);
```

## Example

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '04:30:00' for the TIME literal.

The result of the CAST is the time formed from the time portion of the source expression value '04:30:00' at UTC, and the specified time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE.



The result value of the CAST '04:30:00' at UTC is adjusted to its time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIME '20:30:00-08:00'.

This result of the SELECT statement is equal to TIME '04:30:00+00:00'. The current session time zone displacement, INTERVAL '08:00' HOUR TO MINUTE, has no effect.

```
SET TIME ZONE INTERVAL '08:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIME(0)
      WITH TIME ZONE AT -8);
```

## Example

In this example, the current timestamp is:

```
Current TimeStamp(6)
-----
2010-03-09 19:23:27.620000+00:00
```

The following statement converts the TIME value '08:30:00' to a TIME WITH TIME ZONE value, where the time zone displacement is based on the time zone string, 'America Pacific'.

```
SELECT CAST(TIME '08:30:00' AS TIME(0) WITH TIME ZONE
      AT 'America Pacific');
      08:30:00
-----
00:30:00-08:00
```

## Related Information

For details, see *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211. If there are multiple columns named *source* in the scope, an error is returned.

## TIME-to-TIMESTAMP Conversion

You can convert a TIME or TIME WITH TIME ZONE value to a TIMESTAMP or TIMESTAMP WITH TIME ZONE value using optional data attributes, with either the CAST statement or Teradata conversion syntax.

## TIME-to-TIMESTAMP Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting of DateTime data.

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using CAST to convert from TIME to TIMESTAMP. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

---

**Note:**

TIME (without time zone) and TIMESTAMP (without time zone) are not ANSI SQL:2011 compliant. Vantage internally converts a TIME or TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

---

## Syntax

```
CAST (
  time_expression AS TIMESTAMP

  [ ( fractional_seconds_precision ) ]

  [ WITH TIME ZONE ]

  [ AT { LOCAL | SOURCE [ TIME ZONE ] |
        [ TIME ZONE ] { expression | time_zone_string }
    }
  ]

  [ data_attribute [...] ]
)
```

## Syntax Elements

### *time\_expression*

The TIME expression to be converted.

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

**AT LOCAL**

Use the time zone displacement based on the current session time zone.

**AT SOURCE [TIME ZONE]**

Use the time zone associated with `timestamp_expression` when:

- AT SOURCE TIME ZONE is specified.
- AT SOURCE is specified without TIME ZONE and there is no column named source in the scope.

Otherwise, if AT SOURCE is specified without TIME ZONE and a column named source exists, then SOURCE references this column, and the value of the column is used as the time zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE.

**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.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Teradata TIME-to-TIMESTAMP Conversion Syntax

**ANSI Compliance**

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

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using Teradata conversion syntax to convert from TIME to TIMESTAMP. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

**Note:**

TIME (without time zone) and TIMESTAMP (without time zone) are not ANSI SQL:2011 compliant. Vantage internally converts a TIME or TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

**Syntax**

```
time_expression (
  [ data_attribute [, ...] ] TIMESTAMP

  [ ( fractional_seconds_precision ) ]

  [, WITH TIME ZONE ]

  [ AT { LOCAL | SOURCE [ TIME ZONE ] |
        [ TIME ZONE ] { expression | time_zone_string }
    }
  ]

  [, data_attribute [, ...] ]
)
```

**Syntax Elements*****time\_expression***

The TIME expression to be converted.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

***fractional\_seconds\_precision***

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

**AT LOCAL**

Use the time zone displacement based on the current session time zone.

**AT SOURCE [TIME ZONE]**

Use the time zone associated with `timestamp_expression` when:

- AT SOURCE TIME ZONE is specified.
- AT SOURCE is specified without TIME ZONE and there is no column named source in the scope.

Otherwise, if AT SOURCE is specified without TIME ZONE and a column named source exists, then SOURCE references this column, and the value of the column is used as the time zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE.

**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.

**Usage Notes**

If you specify the AT clause for a `TIMESTAMP[(n)]` without time zone target data type, the following table shows the result of the CAST function or Teradata conversion based on the various options specified. If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision. If the target precision is lower than the source precision, an error is returned.

IF you specify...	AND the data type of <i>time_expression</i> is...	THEN...
AT LOCAL	with or without TIME ZONE	the source <i>time_expression</i> (in UTC) is adjusted by adding the time zone displacement based on the current session time zone. A local timestamp value is formed from <code>CURRENT_DATE</code> (at the above time zone displacement) and the time portion of <i>time_expression</i> obtained after the previous adjustment. The result is this local timestamp value adjusted to UTC by subtracting the above time zone displacement. This is the same as not specifying the AT clause.

IF you specify...	AND the data type of <i>time_expression</i> is...	THEN...
AT SOURCE (where SOURCE is a keyword and not a column reference)	WITH TIME ZONE	the source <i>time_expression</i> (in UTC) is adjusted by adding the time zone displacement of <i>time_expression</i> . A local timestamp value is formed from CURRENT_DATE (based on the time zone displacement of <i>time_expression</i> ) and the time portion of <i>time_expression</i> obtained after the previous adjustment. The result is this local timestamp value adjusted to UTC by subtracting the time zone displacement of <i>time_expression</i> .
AT SOURCE (where SOURCE is a keyword and not a column reference)	without TIME ZONE	an error is returned.
AT SOURCE TIME ZONE	WITH TIME ZONE	the source <i>time_expression</i> (in UTC) is adjusted by adding the time zone displacement of <i>time_expression</i> . A local timestamp value is formed from CURRENT_DATE (based on the time zone displacement of <i>time_expression</i> ) and the time portion of <i>time_expression</i> obtained after the previous adjustment. The result is this local timestamp value adjusted to UTC by subtracting the time zone displacement of <i>time_expression</i> .
AT SOURCE TIME ZONE	without TIME ZONE	an error is returned.
AT expression or AT TIME ZONE <i>expression</i>	with or without TIME ZONE	the source <i>time_expression</i> (in UTC) is adjusted by adding the time zone displacement defined by <i>expression</i> . A local timestamp value is formed from CURRENT_DATE at the above time zone displacement and the time portion of <i>time_expression</i> obtained after the above adjustment. The result is this local timestamp value adjusted to UTC by subtracting the above time zone displacement.
AT time_zone_string or AT TIME ZONE time_zone_string	with or without TIME ZONE	the source <i>time_expression</i> (in UTC) is adjusted by adding the time zone displacement based on <i>time_zone_string</i> . The time zone displacement is determined based on <i>time_zone_string</i> , CURRENT_TIMESTAMP AT '00:00', and the TIME value of <i>time_expression</i> at UTC. A local timestamp value is formed from CURRENT_DATE at the above time zone displacement and the time portion of <i>time_expression</i> obtained after the above adjustment. The result is this local timestamp value adjusted to UTC by subtracting the above time zone displacement.

If you specify the AT clause for a TIMESTAMP[(n)] WITH TIME ZONE target data type, the following table shows the result of the CAST function or Teradata conversion based on the various options specified. If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision. If the target precision is lower than the source precision, an error is returned.

IF you specify...	AND the data type of <i>time_expression</i> is...	THEN...
AT LOCAL	with or without TIME ZONE	the source <i>time_expression</i> (in UTC) is adjusted by adding the time zone displacement based on the current session time zone. A local timestamp value is formed from <i>CURRENT_DATE</i> (at the above time zone displacement) and the time portion of <i>time_expression</i> obtained after the above adjustment. This resulting timestamp is adjusted to UTC, and the result value of the CAST at UTC is adjusted to the above time zone displacement.  If the data type of <i>time_expression</i> is without time zone, this is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	WITH TIME ZONE	the source <i>time_expression</i> (in UTC) is adjusted by adding the time zone displacement of <i>time_expression</i> . A local timestamp value is formed from <i>CURRENT_DATE</i> (based on the time zone displacement of <i>time_expression</i> ) and the time portion of <i>time_expression</i> obtained after the previous adjustment. This resulting timestamp is adjusted to UTC, and the result value of the CAST at UTC is adjusted to the time zone displacement of <i>time_expression</i> .
AT SOURCE (where SOURCE is a keyword and not a column reference)	without TIME ZONE	an error is returned.
AT SOURCE TIME ZONE	WITH TIME ZONE	the source <i>time_expression</i> (in UTC) is adjusted by adding the time zone displacement of <i>time_expression</i> . A local timestamp value is formed from <i>CURRENT_DATE</i> (based on the time zone displacement of <i>time_expression</i> ) and the time portion of <i>time_expression</i> obtained after the previous adjustment. This resulting timestamp is adjusted to UTC, and the result value of the CAST at UTC is adjusted to the time zone displacement of <i>time_expression</i> .
AT SOURCE TIME ZONE	without TIME ZONE	an error is returned.
AT expression or AT TIME ZONE <i>expression</i>	with or without TIME ZONE	the source <i>time_expression</i> (in UTC) is adjusted by adding the time zone displacement defined by <i>expression</i> . A local timestamp value is formed from <i>CURRENT_DATE</i> (at the above time zone displacement) and the time portion of <i>time_expression</i> obtained after the above adjustment. This resulting timestamp is adjusted to UTC, and the result value of the CAST at UTC is adjusted to the above time zone displacement.
AT <i>time_zone_string</i> or AT TIME ZONE <i>time_zone_string</i>	with or without TIME ZONE	the source <i>time_expression</i> (in UTC) is adjusted by adding the time zone displacement based on <i>time_zone_string</i> . The time zone displacement is determined based on <i>time_zone_string</i> , <i>CURRENT_TIMESTAMP AT '00:00'</i> , and the TIME value of <i>time_expression</i> at UTC.  A local timestamp value is formed from <i>CURRENT_DATE</i> (at the above time zone displacement) and the time portion of <i>time_expression</i> .

IF you specify...	AND the data type of <i>time_expression</i> is...	THEN...
		expression obtained after the above adjustment. This resulting timestamp is adjusted to UTC, and the result value of the CAST at UTC is adjusted to the above time zone displacement.

## Implicit TIME-to-TIMESTAMP Conversion

SQL Engine performs implicit conversion from TIME to TIMESTAMP data types in some cases. However, implicit conversion from TIME to TIMESTAMP is not supported for comparisons. See [Implicit Conversion of DateTime Types](#).

The following conversions are supported:

From source type...	To target type...
TIME	TIMESTAMP
	TIMESTAMP WITH TIME ZONE
TIME WITH TIME ZONE	TIMESTAMP
	TIMESTAMP WITH TIME ZONE

## Examples

### Example

Assuming the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, the following SELECT statements return the result: TIMESTAMP '2008-05-14 08:30:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0));
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) AT LOCAL);
```

The current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is used to determine the UTC value '23:30:00' of the literal.

For the CAST, the source expression value '23:30:00' at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' TO MINUTE, to yield '08:30:00'. A timestamp is formed from the current date '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, and the time portion of the source expression value '08:30:00'. Then, this timestamp, '2008-05-14 08:30:00', at time



zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-13 23:30:00' at UTC.

The result value of the CAST at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, so the result of the SELECT statements is: TIMESTAMP '2008-05-14 08:30:00'.

## Example

Assuming the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, the following SELECT statements return the result: TIMESTAMP '2008-05-14 13:30:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0));
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0) AT LOCAL);
```

The time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, in the literal is used to determine the UTC value '04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal. For the CAST, the source expression value '04:30:00' at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE to yield '13:30:00'.

A timestamp is formed from the current date '2008-05-14' at time zone displacement, INTERVAL HOUR '09:00' TO MINUTE, and the time portion of the source expression value '13:30:00'. Then this timestamp, '2008-05-14 13:30:00', at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-14 04:30:00' at UTC.

The result value of the CAST at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, so the result of the SELECT statements is: TIMESTAMP '2008-05-14 13:30:00'.

## Example

An error is returned for the following SELECT statements because the source expression does not have a time zone.

```
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) AT SOURCE TIME ZONE);
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) AT SOURCE);
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) WITH TIME ZONE
  AT SOURCE TIME ZONE);
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) WITH TIME ZONE
  AT SOURCE);
```

## Example

Assume that the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '9:00' HOUR TO MINUTE, but the current date is DATE '2008-05-13' at time zone displacement, INTERVAL '04:00' HOUR TO MINUTE. The following SELECT statement returns the result: TIMESTAMP '2008-05-13 13:30:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0)
           AT SOURCE TIME ZONE);
```

The time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, in the literal is used to determine the UTC value '04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal. For the CAST, the source expression value '04:30:00' at UTC is adjusted to the time zone displacement of the source, INTERVAL '04:00' HOUR TO MINUTE, to yield '08:30:00'.

A timestamp is formed from the current date '2008-05-13' at time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, and the time portion of the source expression value '08:30:00' obtained after the above adjustment. Then this timestamp '2008-05-13 08:30:00' at time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-13 04:30:00' at UTC.

The result value of the CAST at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-13 13:30:00'.

## Example

Assume that the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, but the current date is DATE '2008-05-13' at time zone, INTERVAL '-08:00' HOUR TO MINUTE. The following SELECT statement returns the result: TIMESTAMP '2008-05-14 08:30:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) AT -8);
```

The current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is used to determine the UTC value '23:30:00' of the literal. For the CAST, the source expression value '23:30:00' at UTC is adjusted to the target time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, to yield '15:30:00'.

A timestamp is formed from the current date '2008-05-13' at time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, and the time portion of the source expression value '15:30:00' obtained after the above adjustment. Then this resulting timestamp '2008-05-13 15:30:00' at time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-13 23:30:00' at UTC.

The result value of the CAST at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-14 08:30:00'.

## Example

Assume that the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, but the current date is DATE '2008-05-13' at time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE. The following SELECT statement returns the result: TIMESTAMP '2008-05-14 13:30:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0) AT -8);
```

The time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, in the literal is used to determine the UTC value '04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal. For the CAST, the source expression value '04:30:00' at UTC is adjusted to the target time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, to yield '20:30:00'.

A timestamp is formed from the current date '2008-05-13' at time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, and the time portion of the source expression value '20:30:00' obtained after the above adjustment. Then this timestamp '2008-05-13 20:30:00' at time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-14 04:30:00' at UTC.

The result value of the CAST at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-14 13:30:00'.

## Example

Assuming the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, the following SELECT statements return the result: TIMESTAMP '2008-05-14 08:30:00+09:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) WITH TIME ZONE);
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) WITH TIME ZONE AT LOCAL);
```

The current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is used to determine the UTC value '23:30:00' of the literal. For the CAST, the source expression value '23:30:00' at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, to yield '08:30:00'.

A timestamp is formed from the current date '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, and the time portion of the source expression value '08:30:00' obtained after the

above adjustment. Then this timestamp '2008-05-14 08:30:00' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-13 23:30:00' at UTC with time zone displacement, INTERVAL '09:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, so the result of the SELECT statements is: TIMESTAMP '2008-05-14 08:30:00+09:00'.

## Example

Assuming the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, the following SELECT statement returns the result: TIMESTAMP '2008-05-14 13:30:00+09:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0)
           WITH TIME ZONE AT LOCAL);
```

The time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, in the literal is used to determine the UTC value '04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal. For the CAST, the source expression value '04:30:00' at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, to yield '13:30:00'.

A timestamp is formed from the current date '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, and the time portion of the source expression value '13:30:00' obtained after the above adjustment. Then this timestamp '2008-05-14 13:30:00' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-14 04:30:00' at UTC with time zone displacement, INTERVAL '09:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-14 13:30:00+09:00'.

## Example

Assume that the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, but the current date is DATE '2008-05-13' at time zone displacement, INTERVAL '04:00' HOUR TO MINUTE. The following SELECT statement returns the result: TIMESTAMP '2008-05-14 08:30:00+04:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0) WITH TIME ZONE);
```

The time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, in the literal is used to determine the UTC value '04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal. For the CAST, the source expression value '04:30:00' at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, to yield '13:30:00'.

A timestamp is formed from the current date '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, and the time portion of the source expression value '13:30:00' obtained after the above adjustment. Then this timestamp '2008-05-14 13:30:00' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-14 04:30:00' at UTC with time zone displacement, INTERVAL '04:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL '04:00' INTERVAL TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-14 08:30:00+04:00'.

## Example

Assume that the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, but the current date is DATE '2008-05-13' at time zone displacement, INTERVAL '04:00' HOUR TO MINUTE. The following SELECT statement returns the result: TIMESTAMP '2008-05-13 08:30:00+04:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0) WITH TIME ZONE
  AT SOURCE);
```

The time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, in the literal is used to determine the UTC value '04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal. For the CAST, the source expression value '04:30:00' at UTC is adjusted to the time zone displacement of the source expression, INTERVAL '04:00' HOUR TO MINUTE, to yield '08:30:00'.

A timestamp is formed from the current date '2008-05-13' at time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, and the time portion of the source expression value '08:30:00' obtained after the above adjustment. Then this timestamp '2008-05-13 08:30:00' at time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-13 04:30:00' at UTC with time zone displacement, INTERVAL '04:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone, INTERVAL '04:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-13 08:30:00+04:00'. The current session time zone has no effect.

## Example

Assume that the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, but the current date is DATE '2008-05-13' at time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE. The following SELECT statement returns the result: TIMESTAMP '2008-05-13 15:30:00-08:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) WITH TIME ZONE AT -8);
```

The current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is used to determine the UTC value '23:30:00' of the literal. For the CAST, the source expression value '23:30:00' at UTC is adjusted to the target time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, to yield '15:30:00'.

A timestamp is formed from the current date '2008-05-13' at time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, and the time portion of the source expression value '15:30:00' obtained after the above adjustment. Then this timestamp '2008-05-13 15:30:00' at time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-13 23:30:00' at UTC with time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-13 15:30:00-08:00'.

## Example

Assume that the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, but the current date is DATE '2008-05-13' at time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE. The following SELECT statement returns the result: TIMESTAMP '2008-05-13 20:30:00-08:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0) WITH TIME ZONE
  AT -8);
```

The time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, in the literal is used to determine the UTC value '04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal. For the CAST, the source expression value '04:30:00' at UTC is adjusted to the target time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, to yield '20:30:00'.

A timestamp is formed from the current date '2008-05-13' at time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, and the time portion of the source expression value '20:30:00' obtained after the above adjustment. Then this timestamp '2008-05-13 20:30:00' at time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-14 04:30:00' at UTC with time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-13 20:30:00-08:00'. The current session time zone has no effect.

## Example

In this example, the current timestamp is:

```

Current TimeStamp(6)
-----
2010-03-09 19:23:27.620000+00:00

```

The following statement converts the TIME value '08:30:00' to a TIMESTAMP value, where the time zone displacement is based on the time zone string, 'America Pacific'.

```
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) AT 'America Pacific');
```

The result of the query is:

```

08:30:00
-----
2010-03-09 08:30:00

```

## Example

In this example, the current timestamp is:

```

Current TimeStamp(6)
-----
2010-03-09 19:23:27.620000+00:00

```

The following statement converts the TIME value '08:30:00+04:00' to a TIMESTAMP value, where the time zone displacement is based on the time zone string, 'America Pacific'.

```
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0)
AT 'America Pacific');
```

The result of the query is:

```

08:30:00+04:00
-----
2010-03-10 04:30:00

```

## TIME-to-UDT Conversion

Converts TIME data to UDT data.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

## Syntax

```
CAST ( time_expression AS UDT_data_type )
```

## Syntax Elements

### *time\_expression*

The TIME expression to be converted.

### *UDT\_data\_type*

The UDT type, followed by any FORMAT, NAMED or TITLE data attribute phrases, to which the expression is to be converted.

## Implicit TIME-to-UDT Conversion

SQL Engine performs implicit TIME-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see [Usage Notes](#)) exists that specifies the AS ASSIGNMENT clause.

If no TIME-to-UDT implicit cast definition exists, Vantage looks for a CHAR-to-UDT or VARCHAR-to-UDT implicit cast definition that can substitute for the TIME-to-UDT implicit cast definition. Substitutions are valid because Vantage can implicitly cast a TIME type to the character data type, and then use the implicit cast definition to cast from the character data type to the UDT. If multiple character-to-UDT implicit cast definitions exist, then SQL Engine returns an SQL error.

## Usage Notes

Explicit TIME-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.



## TIMESTAMP-to-Character Conversion

You can convert a TIMESTAMP value to a character string with either the CAST statement or Teradata conversion syntax.

### TIMESTAMP-to-Character Conversion with CAST

#### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of character data attribute phrases.

#### Syntax

```
CAST (
    timestamp_expression AS character_data_type
    [ CHARACTER SET server_character_set ]
    [ data_attribute [...] ]
)
```

#### Syntax Elements

##### *timestamp\_expression*

The TIME expression to be converted.

##### *character\_data\_type*

The data type to which the expression is to be converted.

##### *server\_character\_set*

The server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default server character set is used.

##### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Teradata TIMESTAMP-to-Character Conversion Syntax

### ANSI Compliance

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

### Syntax

```
timestamp_expression (
  [ data_attribute, [...] ]
  character_data_type
  [, { data_attribute | CHARACTER SET server_character_set } [,... ] ]
)
```

### Syntax Elements

#### *timestamp\_expression*

The TIME expression to be converted.

#### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

#### *character\_data\_type*

The data type to which the expression is to be converted.

#### *server\_character\_set*

The server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default server character set is used.

## Usage Notes

When converting TIMESTAMP to CHAR(*n*) or VARCHAR(*n*), then *n* must be equal to or greater than the length of the TIMESTAMP value as represented by a character string literal.

IF the target data type is ...	AND <i>n</i> is ...	THEN ...
CHAR( <i>n</i> )	greater than the length of the TIMESTAMP value as represented by a character string literal	trailing pad characters are added to pad the representation.
	too small	a string truncation error is returned.
VARCHAR( <i>n</i> )	greater than the length of the TIMESTAMP value as represented by a character string literal	no blank padding is added to the character representation.
	too small	a string truncation error is returned.

TIMESTAMP to CLOB conversion is not supported.

You cannot convert a TIME value to a character string if the server character set is GRAPHIC.

## Forcing a FORMAT on CAST for Converting TIMESTAMP to Character

The default format for TIMESTAMP to character conversion is the format in effect for the TIMESTAMP value.

To override the format, you can convert a TIMESTAMP value to a string using a FORMAT phrase. The resulting format, however, is the same as the TIMESTAMP value. If you want a different format for the string value, you need to also use CAST as described here.

You must use nested CAST operations in order to convert values from TIMESTAMP to CHAR and force an explicit FORMAT on the result regardless of the format associated with the TIMESTAMP value. This is because of the rules for matching FORMAT phrases to data types.

## Example

Field TS1 in the table INTTIMESTAMP is a TIMESTAMP value with the explicit format 'Y4-MM-DDBHH:MI:SSDS(6)'. Assume that you want to convert this to a value of CHAR(19), and an explicit output format of 'M3BDD,BY4BHHhMIm'.

```
SELECT TS1 FROM INTTIMESTAMP;
```

The result (without a type change) is the following report:

```

              TS1
-----
1900-12-31 08:25:37.899231
```

Now use nested CAST phrases and a FORMAT to obtain the desired result: a report in character format.

```
SELECT
  CAST( (CAST (TS1 AS FORMAT 'M3BDD,BY4BHHhMI'))
  AS CHAR(19))
FROM INTTIMESTAMP;
```

The result after the nested CASTs is the following report.

```
TS1
-----
Dec 31, 1900 08h25m
```

The inner CAST establishes the display format for the TIMESTAMP value and the outer CAST indicates the data type of the desired result.

## TIMESTAMP-to-DATE Conversion

You can convert a TIMESTAMP value to a DATE value with either the CAST statement or Teradata conversion syntax.

### TIMESTAMP-to-DATE Conversion with CAST

#### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of DATE data attribute phrases, such as FORMAT that enables an alternative format.

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using CAST to convert from TIMESTAMP to DATE. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

---

#### Note:

TIMESTAMP (without time zone) is not ANSI SQL:2011 compliant. Vantage internally converts a TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

---

#### Syntax

```
CAST (
  timestamp_expression AS DATE

  [ WITH TIME ZONE ]
```

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

[ data_attribute [...] ]
)
```

## Syntax Elements

### *timestamp\_expression*

The timestamp expression to be converted.

### AT LOCAL

Use the time zone displacement based on the current session time zone.

### AT SOURCE [TIME ZONE]

Use the time zone associated with *timestamp\_expression* when:

- AT SOURCE TIME ZONE is specified.
- AT SOURCE is specified without TIME ZONE and there is no column named source in the scope.

Otherwise, if AT SOURCE is specified without TIME ZONE and a column named source exists, then SOURCE references this column, and the value of the column is used as the time zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE.

### 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.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED

- TITLE

## Teradata TIMESTAMP-to-DATE Conversion Syntax

### ANSI Compliance

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

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using Teradata conversion syntax to convert from TIMESTAMP to DATE. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

---

#### Note:

TIMESTAMP (without time zone) is not ANSI SQL:2011 compliant. Vantage internally converts a TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

---

### Syntax

```
timestamp_expression (
  [ data_attribute [, ...] ] DATE

  [ AT { LOCAL | SOURCE [ TIME_ZONE ] |
        [ TIME_ZONE ] { expression | time_zone_string }
      }
  ]

  [, data_attribute [, ...] ]
)
```

### Syntax Elements

#### *timestamp\_expression*

The timestamp expression to be converted.

#### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

**AT LOCAL**

Use the time zone displacement based on the current session time zone.

**AT SOURCE [TIME ZONE]**

Use the time zone associated with *timestamp\_expression* when:

- AT SOURCE TIME ZONE is specified.
- AT SOURCE is specified without TIME ZONE and there is no column named source in the scope.

Otherwise, if AT SOURCE is specified without TIME ZONE and a column named source exists, then SOURCE references this column, and the value of the column is used as the time zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE.

**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.

**Usage Notes**

The following table shows the result of the CAST function or Teradata conversion based on the various options specified. Note that the time zone adjustment may change the YEAR, MONTH, and DAY fields of the DATE value.

IF you specify...	AND the data type of <i>timestamp_expression</i> is...	THEN...
AT LOCAL	with or without TIME ZONE	the result is the date portion of the source <i>timestamp_expression</i> after adjusting its UTC value by adding the time zone displacement based on the current session time zone. This is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	WITH TIME ZONE	the result is the date portion of the source <i>timestamp_expression</i> after adjusting its UTC value by adding the time zone displacement associated with <i>timestamp_expression</i> .

IF you specify...	AND the data type of <i>timestamp_expression</i> is...	THEN...
AT SOURCE (where SOURCE is a keyword and not a column reference)	without TIME ZONE	an error is returned.
AT SOURCE TIME ZONE	WITH TIME ZONE	the result is the date portion of the source <i>timestamp_expression</i> after adjusting its UTC value by adding the time zone displacement associated with <i>timestamp_expression</i> .
AT SOURCE TIME ZONE	without TIME ZONE	an error is returned.
AT expression or AT TIME ZONE <i>expression</i>	with or without TIME ZONE	the result is the date portion of the source <i>timestamp_expression</i> after adjusting its UTC value by adding the time zone displacement defined by <i>expression</i> .
AT <i>time_zone_string</i> or AT TIME ZONE <i>time_zone_string</i>	with or without TIME ZONE	the result is the date portion of the source <i>timestamp_expression</i> after adjusting its UTC value by adding the time zone displacement based on <i>time_zone_string</i> . The time zone displacement is determined based on <i>time_zone_string</i> and the TIMESTAMP value of <i>timestamp_expression</i>

## Implicit TIMESTAMP-to-DATE Conversion

SQL Engine performs implicit conversion from TIMESTAMP types to DATE in some cases. See [Implicit Conversion of DateTime Types](#).

The following conversions are supported:

From source type...	To target type...
TIMESTAMP	DATE
TIMESTAMP WITH TIME ZONE	ANSIDate dateform mode or IntegerDate dateform mode

The TIMESTAMP value is always converted to DATE in case of comparison.

## Examples

### Example

A single column table has three rows of type TIMESTAMP(0) WITH TIME ZONE.



A query that requests the field values and CASTs them as DATE is performed during a session that has its Local Time Zone defined as `-'08:00'`.

The results table is as follows.

TimeStampWithTimeZone	CastAsDate
-----	
1997-10-07 15:43:00+08:00	1997-10-06
1997-10-07 15:47:52-08:00	1997-10-07
1997-10-07 15:43:00-00:00	1997-10-07

Notice that the difference between the stored Time Zone and the Local Time Zone is 16 hours in the first row, but at the same time the TimeStamp value is 15:43, which is less than 16.

This puzzling result can be clarified using a similar query that casts `TIMESTAMP(0) WITH TIME ZONE` as `TIMESTAMP(0)`, omitting the Time Zone information.

The results table for this query is as follows.

TimeStampWithTimeZone	CastAsTimeStamp
-----	
1997-10-07 15:43:00+08:00	1997-10-06 23:43:00
1997-10-07 15:47:52-08:00	1997-10-07 15:47:52
1997-10-07 15:43:00-00:00	1997-10-07 07:43:00

After the CAST, the values are all displayed at Local Time Zone, and the value in the first row indicates that the 16 hour adjustment rolled the date back 1, to a time near the end of that date.

## Example

Consider the following statements:

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;
SELECT CAST(TIMESTAMP '2008-05-31 22:30:00-08:00'
  AS DATE AT SOURCE TIME ZONE);
SELECT TIMESTAMP '2008-06-01 06:30:00+00:00' AT '-08:00'
  (DATE, AT SOURCE);
SELECT TIMESTAMP '2008-06-01 06:30:00+00:00' (DATE, AT -8);
SELECT TIMESTAMP '2008-06-01 07:30:00' (DATE, AT -8);
```

These SELECT statements return the date for time zone displacement, `INTERVAL '-08:00' HOUR TO MINUTE`; that is, the statements return `'08/05/31'`. If the SELECT statements were specified without an `AT` clause or with an `AT LOCAL` clause, these statements would return `'08/06/01'` for the current session time zone displacement, `INTERVAL '01:00' HOUR TO MINUTE`.

The following shows the results of the SELECT statements if the AT clause was not specified:

```

SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;
SELECT CAST(TIMESTAMP '2008-05-31 22:30:00-08:00' AS DATE);
2008-05-31 22:30:00-08:00
-----
                08/06/01
SELECT TIMESTAMP '2008-06-01 06:30:00+00:00'
      AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;
2008-06-01 06:30:00+00:00 AT TIME ZONE INTERVAL -8:00 HOUR TO MINUTE
-----
                                2008-05-31 22:30:00-08:00
SELECT TIMESTAMP '2008-06-01 06:30:00+00:00'
      AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE (DATE);
2008-06-01 06:30:00+00:00 AT TIME ZONE INTERVAL -8:00 HOUR TO MINUTE
-----
                                                08/06/01
SELECT TIMESTAMP '2008-06-01 06:30:00+00:00' (DATE);
2008-06-01 06:30:00+00:00
-----
                08/06/01
SELECT TIMESTAMP '2008-06-01 07:30:00' (DATE);
2008-06-01 07:30:00
-----
                08/06/01

```

The following shows the results of the SELECT statements if the AT clause was not specified, and the current session time zone displacement is INTERVAL -'08:00' HOUR TO MINUTE.

```

SET TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;
SELECT CAST(TIMESTAMP '2008-05-31 22:30:00-08:00' AS DATE);
2008-05-31 22:30:00-08:00
-----
                08/05/31
SELECT TIMESTAMP '2008-06-01 06:30:00+00:00'
      AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE (DATE);
2008-06-01 06:30:00+00:00 AT TIME ZONE INTERVAL -8:00 HOUR TO MINUTE
-----
                                08/05/31
SELECT TIMESTAMP '2008-06-01 06:30:00+00:00' (DATE);
2008-06-01 06:30:00+00:00
-----
                08/05/31
SELECT CAST(TIMESTAMP '2008-06-01 07:30:00+01:00'

```

```

    AS TIMESTAMP(0)) (DATE);
2008-06-01 07:30:00+01:00
-----
                        08/05/31

```

## Example

Consider the following statements:

```

SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;
SELECT CAST(TIMESTAMP '2008-06-02 04:30:00+09:00'
    AS DATE AT SOURCE TIME ZONE);
SELECT TIMESTAMP '2008-06-01 20:30:00+01:00'
    AT TIME ZONE INTERVAL '09' HOUR (DATE, AT SOURCE);
SELECT TIMESTAMP '2008-06-01 20:30:00' (DATE, AT +9);

```

These SELECT statements return the date for time zone displacement, INTERVAL '09:00' HOUR TO MINUTE; that is, the statements return '08/06/02'. If the SELECT statements were specified without an AT clause or with an AT LOCAL clause, these statements would return '08/06/01' for the current session time zone displacement, INTERVAL '01:00' HOUR TO MINUTE.

The following shows the results of the SELECT statements if the AT clause was not specified:

```

SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;
SELECT CAST(TIMESTAMP '2008-06-02 04:30:00+09:00' AS DATE);
2008-06-02 04:30:00+09:00
-----
                        08/06/01
SELECT TIMESTAMP '2008-06-01 20:30:00+01:00'
    AT TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
2008-06-01 20:30:00+01:00 AT TIME ZONE INTERVAL  9:00 HOUR TO MINUTE
-----
                        2008-06-02 04:30:00+09:00
SELECT TIMESTAMP '2008-06-01 20:30:00+01:00'
    AT TIME ZONE INTERVAL '09:00' HOUR TO MINUTE (DATE);
2008-06-01 20:30:00+01:00 AT TIME ZONE INTERVAL  9:00 HOUR TO MINUTE
-----
                        08/06/01
SELECT TIMESTAMP '2008-06-01 20:30:00' (DATE);
2008-06-01 20:30:00
-----
                        08/06/01

```

The following shows the results of the SELECT statements if the AT clause was not specified, and the current session time zone displacement is INTERVAL '09:00' TO MINUTE.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIMESTAMP '2008-06-02 04:30:00+09:00' AS DATE);
2008-06-02 04:30:00+09:00
-----
                        08/06/02
SELECT TIMESTAMP '2008-06-01 20:30:00+01:00'
      AT TIME ZONE INTERVAL '09:00' HOUR TO MINUTE (DATE);
2008-06-01 20:30:00+01:00 AT TIME ZONE INTERVAL  9:00 HOUR TO MINUTE
-----
                                                08/06/02
SELECT CAST(TIMESTAMP '2008-06-01 20:30:00+01:00'
      AS TIMESTAMP(0)) (DATE);
2008-06-01 20:30:00+01:00
-----
                        08/06/02
```

## Example

Consider the following statements:

```
SET TIME ZONE INTERVAL '10:00' HOUR TO MINUTE;
SELECT CAST((TIMESTAMP '2008-06-01 18:30:00+01:00' AT '05:45')
      AS DATE AT SOURCE);
SELECT CAST((TIMESTAMP '2008-06-01 18:30:00+01:00' AT 5.75)
      AS DATE AT SOURCE);
SELECT TIMESTAMP '2008-06-01 23:15:00+05:45'
      (DATE, AT SOURCE TIME ZONE);
SELECT TIMESTAMP '2008-06-02 03:30:00' (DATE, AT '05:45');
SELECT TIMESTAMP '2008-06-02 03:30:00' (DATE, AT 5.75);
```

These SELECT statements return the date for time zone displacement, INTERVAL '05:45' HOUR TO MINUTE; that is, the statements return '08/06/01'. If the SELECT statements were specified without an AT clause or with an AT LOCAL clause, these statements would return '08/06/02' for the current session time zone displacement, INTERVAL '10:00' HOUR TO MINUTE.

The following shows the results of the SELECT statements if the AT clause was not specified:

```
SET TIME ZONE INTERVAL '10:00' HOUR TO MINUTE;
SELECT TIMESTAMP '2008-06-01 18:30:00+01:00'
      AT TIME ZONE INTERVAL '05:45' HOUR TO MINUTE;
```

```

2008-06-01 18:30:00+01:00 AT TIME ZONE INTERVAL 5:45 HOUR TO MINUTE
-----
                                2008-06-01 23:15:00+05:45
SELECT CAST((TIMESTAMP '2008-06-01 18:30:00+01:00'
    AT TIME ZONE INTERVAL '05:45' HOUR TO MINUTE) AS DATE);
2008-06-01 18:30:00+01:00 AT TIME ZONE INTERVAL 5:45 HOUR TO MINUTE
-----
                                08/06/02

SELECT TIMESTAMP '2008-06-01 23:15:00+05:45' (DATE);
2008-06-01 23:15:00+05:45
-----
                                08/06/02

SELECT TIMESTAMP '2008-06-02 03:30:00' (DATE);
2008-06-02 03:30:00
-----
                                08/06/02

```

The following shows the results of the SELECT statements if the AT clause was not specified, and the current session time zone displacement is INTERVAL '05:45' HOUR TO MINUTE.

```

SET TIME ZONE INTERVAL '05:45' HOUR TO MINUTE;
SELECT CAST((TIMESTAMP '2008-06-01 18:30:00+01:00'
    AT TIME ZONE INTERVAL '05:45' HOUR TO MINUTE) AS DATE);
2008-06-01 18:30:00+01:00 AT TIME ZONE INTERVAL 5:45 HOUR TO MINUTE
-----
                                08/06/01

SELECT TIMESTAMP '2008-06-01 23:15:00+05:45' (DATE);
2008-06-01 23:15:00+05:45
-----
                                08/06/01

SELECT CAST(TIMESTAMP '2008-06-02 03:30:00+10:00'
    AS TIMESTAMP(0)) (DATE);
2008-06-02 03:30:00+10:00
-----
                                08/06/01

```

## Example

Consider the following statements:

```
SET TIME ZONE +1;
SELECT CAST((TIMESTAMP '2008-06-01 08:30:00' AT TIME ZONE -8)
  AS DATE AT SOURCE TIME ZONE);
```

This SELECT statement returns the date for time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE; that is, the statement returns '08/05/31'. If the SELECT statement was specified without an AT clause or with an AT LOCAL clause, the statement would return '08/06/01' for the current session time zone displacement, INTERVAL HOUR '01:00' MINUTE.

The following shows the result of the SELECT statement if the AT clause was not specified:

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;
SELECT TIMESTAMP '2008-06-01 08:30:00'
  AT TIME ZONE INTERVAL '-08:00' HOUR TO MINUTE;
2008-06-01 08:30:00 AT TIME ZONE INTERVAL -8:00 HOUR TO MINUTE
-----
                                2008-05-31 23:30:00-08:00
SELECT CAST((TIMESTAMP '2008-06-01 08:30:00'
  AT TIME ZONE INTERVAL '-08:00' HOUR TO MINUTE) AS DATE);
2008-06-01 08:30:00 AT TIME ZONE INTERVAL -8:00 HOUR TO MINUTE
-----
                                08/06/01
```

The following shows the result of the SELECT statement if the AT clause was not specified, and the current session time zone displacement is INTERVAL '-08:00' HOUR TO MINUTE.

```
SET TIME ZONE INTERVAL '-08:00' HOUR TO MINUTE;
SELECT CAST((CAST(TIMESTAMP '2008-06-01 08:30:00+01:00'
  AS TIMESTAMP(0)) AT TIME ZONE INTERVAL '-08:00' HOUR TO MINUTE)
  AS DATE);
2008-06-01 08:30:00+01:00 AT TIME ZONE INTERVAL -8:00 HOUR TO MINUTE
-----
                                08/05/31
```

## Example

In this example, the current timestamp is:

```
Current TimeStamp(6)
-----
2010-03-09 19:23:27.620000+00:00
```

The following statement converts the `TIMESTAMP` value '2010-03-09 22:30:00-08:00' to a `DATE` value, where the time zone displacement is based on the time zone string, 'America Pacific'.

```
SELECT CAST(TIMESTAMP '2010-03-09 22:30:00-08:00' AS DATE
           AT 'America Pacific');
```

The result of the query is:

```
2010-03-09 22:30:00-08:00
-----
                        10/03/09
```

## Example

The following `SELECT` statements return an error because the source expression does not have a `TIMESTAMP WITH TIME ZONE` data type.

```
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00' AS DATE AT SOURCE);
SELECT CAST(TIME '08:30:00+03:00' AS DATE AT SOURCE TIME ZONE);
SELECT CAST(TIME '08:30:00' AS DATE AT SOURCE);
SELECT CAST(DATE '2008-06-01' AS DATE AT SOURCE TIME ZONE);
```

## TIMESTAMP-to-Period Conversion

Converts a `TIMESTAMP` value as `PERIOD(DATE)`, `PERIOD(TIME[(n)][WITH TIME ZONE])`, or `PERIOD(TIMESTAMP[(n)][WITH TIME ZONE])`.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, `CAST` permits the use of data attribute phrases.

### Syntax

```
CAST (
    timestamp_expression AS period_data_type
    [ data_attribute [...] ]
)
```

## Syntax Elements

### *timestamp\_expression*

The timestamp expression to be converted.

### *period\_data\_type*

The data type to which the expression is to be converted.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Usage Notes

A `TIMESTAMP(n)` [WITH TIME ZONE] value can be cast as `PERIOD(DATE)`, `PERIOD(TIME[(n)]` [WITH TIME ZONE]), or `PERIOD(TIMESTAMP[(n)]` [WITH TIME ZONE] using the `CAST` function.

If the target type is `PERIOD(TIME[(n)]` [WITH TIME ZONE]) or `PERIOD(TIMESTAMP[(n)]` [WITH TIME ZONE]):

- If the target precision is higher than the source precision, trailing zeros are added in the result bounds to adjust the precision.
- If the target precision is lower than the source precision, an error is reported.

If the target type is `PERIOD(DATE)`, the result beginning bound is the date portion of the source beginning bound adjusted to the current session time zone.

If the target type is `PERIOD(TIME[(n)])`, the result beginning bound is the time portion of the source value (in UTC).

If the target type is `PERIOD(TIME[(n)]` WITH TIME ZONE), the result beginning bound is formed from the time portion of the source value (in UTC) and, if the source type is WITH TIME ZONE, the source time zone displacement and, if not, the current session time zone displacement.

If the target type is `PERIOD(TIMESTAMP[(n)])`, the result beginning bound is the timestamp portion of the source value (in UTC).

If the target type is `PERIOD(TIMESTAMP[(n)]` WITH TIME ZONE), the result beginning bound is formed from the timestamp portion of the source value (in UTC) and, if the source type is WITH TIME ZONE, the source time zone displacement and, if not, the current session time zone displacement.

If the `TIMESTAMP` source value contains leap seconds, the seconds portion gets adjusted to 59.999999 with the precision truncated to the target precision.



The result ending element is set to the result beginning bound plus one granule of the target type. If the result ending bound exceeds the maximum allowed DATE or TIMESTAMP value for a target type of PERIOD(DATE) or PERIOD(TIMESTAMP[(n)]), respectively, or the ending bound has a lower value than the result beginning bound in their UTC forms for a target type of PERIOD(TIME[(n)]), an error is reported.

---

**Note:**

If the target type is WITH TIME ZONE, the result beginning and ending bounds have the same time zones.

---

Also, note that the result has the same value for the beginning bound and last value.

## Example

In the following example, a TIMESTAMP(6) literal is cast as PERIOD(DATE). The result beginning element is set to the date portion of the source value. The result ending element is set to result beginning bound plus INTERVAL '1' DAY.

```
SELECT CAST(TIMESTAMP '2005-02-03 12:12:12.340000' AS PERIOD(DATE));
```

The following is returned:

```
('2005-02-03', '2005-02-04')
```

## TIMESTAMP-to-TIME Conversion

You can convert a TIMESTAMP value to a TIME value with either the CAST statement or Teradata conversion syntax.

### TIMESTAMP-to-TIME Conversion with CAST

#### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of TIME data attribute phrases.

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using CAST to convert from TIMESTAMP to TIME. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

**Note:**

TIME (without time zone) and TIMESTAMP (without time zone) are not ANSI SQL:2011 compliant. Vantage internally converts a TIME or TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

**Syntax**

```
CAST (
    timestamp_expression AS TIME

    [ ( fractional_seconds_precision ) ]

    [ WITH TIME ZONE ]

    [ AT { LOCAL | SOURCE [ TIME ZONE ] |
          [ TIME ZONE ] { expression | time_zone_string }
      }
    ]

    [ data_attribute [...] ]
)
```

**Syntax Elements*****timestamp\_expression***

The timestamp expression to be converted.

***fractional\_seconds\_precision***

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

**AT LOCAL**

Use the time zone displacement based on the current session time zone.

**AT SOURCE [TIME ZONE]**

Use the time zone associated with *timestamp\_expression* when:

- AT SOURCE TIME ZONE is specified.
- AT SOURCE is specified without TIME ZONE and there is no column named source in the scope.

Otherwise, if AT SOURCE is specified without TIME ZONE and a column named source exists, then SOURCE references this column, and the value of the column is used as the time zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE.

#### **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.

#### ***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## **Teradata TIMESTAMP-to-TIME Conversion Syntax**

### **ANSI Compliance**

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

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using Teradata conversion syntax to convert from TIMESTAMP to TIME. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

---

#### **Note:**

TIME (without time zone) and TIMESTAMP (without time zone) are not ANSI SQL:2011 compliant. Vantage internally converts a TIME or TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

---

### **Syntax**

```
timestamp_expression (
  [ data_attribute [, ...] ] TIME
```

```
[ ( fractional_seconds_precision ) ]

[, WITH TIME ZONE ]

[ AT { LOCAL | SOURCE [ TIME ZONE ] |
      [ TIME ZONE ] { expression | time_zone_string }
  }
]

[, data_attribute [, ...] ]
)
```

## Syntax Elements

### *timestamp\_expression*

The timestamp expression to be converted.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

### AT LOCAL

Use the time zone displacement based on the current session time zone.

### AT SOURCE [TIME ZONE]

Use the time zone associated with *timestamp\_expression* when:

- AT SOURCE TIME ZONE is specified.
- AT SOURCE is specified without TIME ZONE and there is no column named source in the scope.

Otherwise, if AT SOURCE is specified without TIME ZONE and a column named source exists, then SOURCE references this column, and the value of the column is used as the time

zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE.

### 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.

## Usage Notes

If you specify an AT clause for a TIME[(n)] without time zone target data type, an error is returned.

If you specify an AT clause for a TIME[(n)] WITH TIME ZONE target data type, the following table shows the result of the CAST function or Teradata conversion based on the various options specified. If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision. If the target precision is lower than the source precision, an error is returned.

IF you specify...	AND the data type of <i>timestamp_expression</i> is...	THEN...
AT LOCAL	with or without TIME ZONE	the result is formed from the source <i>timestamp_expression</i> (in UTC) and the time zone displacement based on the current session time zone. If the data type of <i>timestamp_expression</i> is without time zone, this is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	WITH TIME ZONE	the result is formed from the time portion of the source <i>timestamp_expression</i> (in UTC) and the time zone displacement associated with <i>timestamp_expression</i> . Note that this is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	without TIME ZONE	an error is returned.
AT SOURCE TIME ZONE	WITH TIME ZONE	the result is formed from the time portion of the source <i>timestamp_expression</i> (in UTC) and the time zone displacement associated with <i>timestamp_expression</i> . Note that this is the same as not specifying the AT clause.

IF you specify...	AND the data type of <i>timestamp_expression</i> is...	THEN...
AT SOURCE TIME ZONE	without TIME ZONE	an error is returned.
AT expression or AT TIME ZONE <i>expression</i>	with or without TIME ZONE	the result is formed from the time portion of the source <i>timestamp_expression</i> (in UTC) and the time zone displacement defined by <i>expression</i> .
AT <i>time_zone_string</i> or AT TIME ZONE <i>time_zone_string</i>	with or without TIME ZONE	the result is formed from the time portion of the source <i>timestamp_expression</i> (in UTC) and the time zone displacement based on <i>time_zone_string</i> . The time zone displacement is determined based on <i>time_zone_string</i> and the TIMESTAMP value of <i>timestamp_expression</i> at UTC.

## Implicit TIMESTAMP-to-TIME Conversion

SQL Engine performs implicit conversion from TIMESTAMP to TIME data types in some cases. However, implicit conversion from TIMESTAMP to TIME is not supported for comparisons. See [Implicit Conversion of DateTime Types](#).

The following conversions are supported.

From source type...	To target type...
TIMESTAMP	TIME
	TIME WITH TIME ZONE
TIMESTAMP WITH TIME ZONE	TIME
	TIME WITH TIME ZONE

## Examples

### Example

In this example, the current session time zone displacement, INTERVAL '01:00' HOUR TO MINUTE, is used to determine the UTC value, '2008-06-01 07:30:00', of the TIMESTAMP literal.

The result of the CAST is the time formed from the time portion of the source expression value '2008-06-01 07:30:00' at UTC and the current time zone displacement, INTERVAL '01:00' HOUR TO MINUTE.

The result value of the CAST '07:30:00' at UTC is adjusted to its time zone displacement, INTERVAL '01:00' HOUR TO MINUTE, and the result of the SELECT statements is: TIME '08:30:00+01:00'.

The result of the SELECT statements is equal to TIME '07:30:00+00:00' since values are compared based on their UTC values.

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-06-01 08:30:00' AS TIME(0)
  WITH TIME ZONE);
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00' AS TIME(0)
  WITH TIME ZONE AT LOCAL);
```

## Example

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '2008-06-01 04:30:00' for the TIMESTAMP literal.

The result of the CAST is the time formed from the time portion of the source expression value '2008-06-01 04:30:00' at UTC and the current session time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE.

The result value of the CAST '04:30:00' at UTC is adjusted to its time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIME '20:30:00-08:00'.

The result of the SELECT statement is equal to TIME '04:30:00+00:00'.

```
SET TIME ZONE INTERVAL '-08:00' HOUR TO MINUTE;
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
  AS TIME(0) WITH TIME ZONE AT LOCAL);
```

## Example

The following SELECT statement return an error because the source expression does not have a time zone displacement.

```
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00'
  AS TIME(0) WITH TIME ZONE AT SOURCE);
```

## Example

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '2008-06-01 04:30:00' for the TIMESTAMP literal.

The result of the CAST is the time formed from the time portion of the source expression value '2008-06-01 04:30:00' at UTC, and the time zone displacement of the source expression, INTERVAL '04:00' HOUR TO MINUTE.

The result value of the CAST '04:30:00' at UTC is adjusted to its time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, and the result of the SELECT statements is: TIME '08:30:00+04:00'.

The result of the SELECT statements is equal to TIME '04:30:00+00:00'. The current session time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, has no effect.

```
SET TIME ZONE INTERVAL '-08:00' HOUR TO MINUTE;
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
  AS TIME(0) WITH TIME ZONE);
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
  AS TIME(0) WITH TIME ZONE AT SOURCE TIME ZONE);
```

## Example

In this example, the current session time zone displacement, INTERVAL '-04:00' HOUR TO MINUTE, is used to determine the UTC value '2008-06-01 12:30:00' for the TIMESTAMP literal.

The result of the CAST is the time formed from the time portion of the source expression value '2008-06-01 12:30:00' at UTC, and the specified time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE.

The result value of the CAST '12:30:00' at UTC is adjusted to its time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIME '04:30:00-08:00'.

The result of the SELECT statement is equal to TIME '12:30:00+00:00'.

```
SET TIME ZONE INTERVAL '-04:00' HOUR TO MINUTE;
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00'
  AS TIME(0) WITH TIME ZONE AT -8);
```

## Example

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '2008-06-01 04:30:00' for the TIMESTAMP literal.

The result of the CAST is the time formed from the time portion of the source expression value '2008-06-01 04:30:00' at UTC, and the specified time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE.

The result value of the CAST '04:30:00' at UTC is adjusted to its time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIME '20:30:00-08:00'.

This result of the SELECT statement is equal to TIME '04:30:00+00:00'. The current session time zone displacement, INTERVAL '08:00' HOUR TO MINUTE, has no effect.



```
SET TIME ZONE INTERVAL '08:00' HOUR TO MINUTE;
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
           AS TIME(0) WITH TIME ZONE AT -8);
```

## Example

In this example, the current timestamp is:

```
Current TimeStamp(6)
-----
2010-03-09 19:23:27.620000+00:00
```

The following statement converts the TIMESTAMP value '2010-03-09 08:30:00' to a TIME WITH TIME ZONE value, where the time zone displacement is based on the time zone string, 'America Pacific'.

```
SELECT CAST(TIMESTAMP '2010-03-09 08:30:00' AS TIME(0) WITH TIME ZONE
           AT 'America Pacific');
```

The result of the query is:

```
2010-03-09 08:30:00
-----
00:30:00-08:00
```

## TIMESTAMP-to-TIMESTAMP Conversion

You can convert TIMESTAMP value to a TIMESTAMP value with different precision information or a WITH TIME ZONE definition, with either the CAST statement or Teradata conversion syntax.

## TIMESTAMP-to-TIMESTAMP Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting for the character representations of DateTime and Interval data.

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using CAST to convert from TIMESTAMP to TIMESTAMP. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

**Note:**

TIMESTAMP (without time zone) is not ANSI SQL:2011 compliant. Vantage internally converts a TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

**Syntax**

```
CAST (
  timestamp_expression AS TIMESTAMP

  [ ( fractional_seconds_precision ) ]

  [ WITH TIME ZONE ]

  [ AT { LOCAL | SOURCE [ TIME ZONE ] |
        [ TIME ZONE ] { expression | time_zone_string }
    }
  ]

  [ data_attribute [...] ]
)
```

**Syntax Elements*****timestamp\_expression***

The timestamp expression to be converted.

***fractional\_seconds\_precision***

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

**AT LOCAL**

Use the time zone displacement based on the current session time zone.

**AT SOURCE [TIME ZONE]**

Use the time zone associated with *timestamp\_expression* when:

- AT SOURCE TIME ZONE is specified.
- AT SOURCE is specified without TIME ZONE and there is no column named source in the scope.

Otherwise, if AT SOURCE is specified without TIME ZONE and a column named source exists, then SOURCE references this column, and the value of the column is used as the time zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE.

#### **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.

#### ***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## **Teradata TIMESTAMP-to-TIMESTAMP Conversion Syntax**

### **ANSI Compliance**

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

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using Teradata conversion syntax to convert from TIMESTAMP to TIMESTAMP. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

---

#### **Note:**

TIMESTAMP (without time zone) is not ANSI SQL:2011 compliant. Vantage internally converts a TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

---

### **Syntax**

```
timestamp_expression (
  [ data_attribute [, ...] ] TIMESTAMP

  [ ( fractional_seconds_precision ) ]
```

```
[ , WITH TIME ZONE ]

[ AT { LOCAL | SOURCE [ TIME ZONE ] |
    [ TIME ZONE ] { expression | time_zone_string }
  }
]

[ , data_attribute [ , ... ] ]
)
```

## Syntax Elements

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

### *timestamp\_expression*

The timestamp expression to be converted.

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

### AT LOCAL

Use the time zone displacement based on the current session time zone.

### AT SOURCE [TIME ZONE]

Use the time zone associated with *timestamp\_expression* when:

- AT SOURCE TIME ZONE is specified.
- AT SOURCE is specified without TIME ZONE and there is no column named source in the scope.

Otherwise, if AT SOURCE is specified without TIME ZONE and a column named source exists, then SOURCE references this column, and the value of the column is used as the time

zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE.

### 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.

## Usage Notes

If you specify an AT clause for a TIMESTAMP[(n)] without time zone target data type, an error is returned.

If you specify an AT clause for a TIMESTAMP[(n)] WITH TIME ZONE target data type, the following table shows the result of the CAST function or Teradata conversion based on the various options specified. If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision. If the target precision is lower than the source precision, an error is returned.

IF you specify...	AND the data type of <i>timestamp_expression</i> is...	THEN...
AT LOCAL	with or without TIME ZONE	the result is formed from the timestamp portion of the source <i>timestamp_expression</i> (in UTC) with the result time zone displacement based on the current session time zone. If the source data type is without time zone, this is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	WITH TIME ZONE	the result is formed from the timestamp portion of the source <i>timestamp_expression</i> (in UTC) and the time zone displacement associated with <i>timestamp_expression</i> . Note that this is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	without TIME ZONE	an error is returned.
AT SOURCE TIME ZONE	WITH TIME ZONE	the result is formed from the timestamp portion of the source <i>timestamp_expression</i> (in UTC) and the time zone displacement associated with <i>timestamp_expression</i> . Note that this is the same as not specifying the AT clause.

IF you specify...	AND the data type of <i>timestamp_expression</i> is...	THEN...
AT SOURCE TIME ZONE	without TIME ZONE	an error is returned.
AT expression or AT TIME ZONE <i>expression</i>	with or without TIME ZONE	the result is formed from the timestamp portion of the source <i>timestamp_expression</i> (in UTC) and the time zone displacement defined by <i>expression</i> .
AT <i>time_zone_string</i> or AT TIME ZONE <i>time_zone_string</i>	with or without TIME ZONE	the result is formed from the timestamp portion of the source <i>timestamp_expression</i> (in UTC) and the time zone displacement based on <i>time_zone_string</i> . The time zone displacement is determined based on <i>time_zone_string</i> and the TIMESTAMP value of <i>timestamp_expression</i> at UTC.

## Examples

### Example

The following SELECT statements return an error because the target data type does not have a TIMESTAMP WITH TIME ZONE data type.

```
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00' AS TIMESTAMP(0)
  AT LOCAL);
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+01:00' AS TIMESTAMP(0)
  AT LOCAL);
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00' AS TIMESTAMP(0)
  AT SOURCE TIME ZONE);
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+01:00' AS TIMESTAMP(0)
  AT SOURCE);
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00' AS TIMESTAMP(0) AT +3);
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+01:00' AS TIMESTAMP(0)
  AT -6);
```

### Example

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '2008-06-01 04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal.

The CAST result is the source expression value '2008-06-01 04:30:00' at UTC with the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIMESTAMP '2008-06-01 13:30:00+09:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
  AS TIMESTAMP(0) WITH TIME ZONE AT LOCAL);
```

## Example

The following SELECT statements return an error because the source expression does not have a time zone displacement.

```
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00'
  AS TIMESTAMP(0) WITH TIME ZONE AT SOURCE TIME ZONE);
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00'
  AS TIMESTAMP(0) WITH TIME ZONE AT SOURCE);
```

## Example

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '2008-06-01 04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal.

The CAST result is source expression value '2008-06-01 04:30:00' at UTC with its time zone displacement, INTERVAL '04:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, and the result of the SELECT is: TIMESTAMP '2008-06-01 08:30:00+04:00'. The current session time zone has no effect.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
  AS TIMESTAMP(0) WITH TIME ZONE);
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
  AS TIMESTAMP(0) WITH TIME ZONE AT SOURCE TIME ZONE);
```

## Example

In this example, the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is used to determine the UTC value '2008-05-31 23:30:00' of the literal.

The CAST result is the source expression value '2008-05-31 23:30:00' at UTC with the target time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIMESTAMP '2008-05-31 15:30:00-08:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00' AS TIMESTAMP(0)
      WITH TIME ZONE AT -8);
```

## Example

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '2008-06-01 04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal.

The CAST result is the source expression value '2008-06-01 04:30:00' at UTC with the target time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL '-08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIMESTAMP '2008-05-31 20:30:00-08:00'. The current session time zone has no effect.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
      AS TIMESTAMP(0) WITH TIME ZONE AT -8);
```

## Example

In this example, the current timestamp is:

```
Current Timestamp(6)
-----
2010-03-09 19:23:27.620000+00:00
```

The following statement converts the TIMESTAMP value '2010-03-09 08:30:00' to a TIMESTAMP WITH TIME ZONE value, where the time zone displacement is based on the time zone string, 'America Pacific'.

```
SELECT CAST(TIMESTAMP '2010-03-09 08:30:00' AS TIMESTAMP(0)
      WITH TIME ZONE AT 'America Pacific');
```

The result of the query is:



```

2010-03-09 08:30:00
-----
2010-03-09 00:30:00-08:00

```

## Example

In this example, the tswz column in table1 contains the following timestamp data:

```
SELECT * FROM table1;
```

The result of the query is:

```

                                tswz
-----
2011-11-04 13:14:00.860000-07:00

```

You can use an AT clause with a time zone string in a SELECT query to get the same time in a different time zone. For example:

```
SELECT CAST(tswz AS TIMESTAMP WITH TIME ZONE AT 'gmt')
FROM table1;
```

The result of the query is:

```

                                tswz
-----
2011-11-04 20:14:00.860000+00:00

```

Similarly, you can use an AT clause with a time zone displacement to get the same result.

```
SELECT CAST(tswz AS TIMESTAMP WITH TIME ZONE AT TIME ZONE '00:00')
FROM table1;
```

The result of the query is:

```

                                tswz
-----
2011-11-04 20:14:00.860000+00:00

```

The following query produces the same result without an explicit CAST:

```
SELECT tswz AT 'gmt' FROM table1;
```

The result of the query is:

```

      tswz AT TIME ZONE 'gmt'
-----
2011-11-04 20:14:00.860000+00:00

```

## Related Information

For more information, see *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211.

## TIMESTAMP-to-UDT Conversion

Converts TIMESTAMP data to UDT data.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

### Syntax

```
CAST ( timestamp_expression AS UDT_data_type )
```

### Syntax Elements

#### *timestamp\_expression*

The timestamp expression to be converted.

#### *UDT\_data\_type*

The UDT type, followed by any FORMAT, NAMED or TITLE data attribute phrases, to which the expression is to be converted.

## Usage Notes

Explicit TIMESTAMP-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement.

## Implicit TIMESTAMP-to-UDT Conversion

SQL Engine performs implicit TIMESTAMP-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the `DisableUDTImplCastForSysFuncOp` field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see [Usage Notes](#)) exists that specifies the AS ASSIGNMENT clause.

If no TIMESTAMP-to-UDT implicit cast definition exists, Vantage looks for a CHAR-to-UDT or VARCHAR-to-UDT implicit cast definition that can substitute. Substitutions are valid because Vantage can implicitly cast a TIMESTAMP type to the character data type, and then use the implicit cast definition to cast from the character data type to the UDT. If multiple character-to-UDT implicit cast definitions exist, then SQL Engine returns an SQL error.

## Related Information

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

## TRYCAST

TRYCAST takes a string and tries to cast it to a data type specified after the AS keyword (similar to CAST). If the conversion fails, TRYCAST returns a NULL instead of failing.

The result of the conversion is returned unless there is an error, in which case a NULL is returned. The result data type is whatever data type was specified by the *data\_type* input.

### Syntax

```
TRYCAST ( instring AS data_type )
```

### Syntax Elements

#### *instring*

A CHAR or VARCHAR expression in the LATIN or UNICODE character set.

#### *data\_type*

One of the following supported data types:

- BYTEINT

- SMALLINT
- INT
- BIGINT
- FLOAT
- DECIMAL
- NUMBER
- CHAR (LATIN or UNICODE)
- VARCHAR (LATIN or UNICODE)
- DATE
- TIME (with zone)
- TIMESTAMP (with zone)
- All INTERVAL types

## Examples

### Example

```
SELECT TRYCAST('-2.5' AS DECIMAL(5,2));
```

```
*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.
```

```
TRYCAST('-2.5')
-----
-2.50
```

### Example: TRYCAST Conversion Failure

An example of when the TRYCAST conversion fails:

```
SELECT TRYCAST('abc' AS DECIMAL(5,2));
```

```
*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.
```

```
TRYCAST('abc')
-----
?
```

## UDT-to-Byte Conversion

You can convert a UDT expression to a byte value with either the CAST statement or Teradata conversion syntax.

### UDT-to-Byte Conversion with CAST

#### ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

#### Syntax

```
CAST ( UDT_expression AS byte_data_type )
```

#### Syntax Elements

##### *UDT\_expression*

An expression that results in a UDT data type.

##### *byte\_data\_type*

The BLOB, BYTE or VARBYTE byte type followed by optional FORMAT, NAMED, or TITLE attribute phrases to which UDT\_expression is to be converted.

## Teradata UDT-to-Byte Conversion Syntax

#### ANSI Compliance

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

#### Syntax

```
UDT_expression ( [ data_attribute, [...] ] byte_data_type [, data_attribute [, [...] ] )
```

#### Syntax Elements

##### *UDT\_expression*

An expression that results in a UDT data type.

##### *data\_attribute*

One of the following data attributes:

- FORMAT

- NAMED
- TITLE

***byte\_data\_type***

The BLOB, BYTE or VARBYTE byte type to which UDT\_expression is to be converted.

## Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement.

## Implicit Type Conversion

SQL Engine performs implicit UDT-to-byte conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit UDT-to-byte data type conversion requires a cast definition (see [Usage Notes](#)) that specifies the following:

- the AS ASSIGNMENT clause
- a BYTE, VARBYTE, or BLOB target data type

The target data type of the cast definition does not have to be an exact match to the target of the implicit type conversion.

If multiple implicit cast definitions exist for converting the UDT to different byte types, Vantage uses the implicit cast definition for the byte type with the highest precedence. The following list shows the precedence of byte types in order from lowest to highest precedence:

- BYTE
- VARBYTE
- BLOB

## Example

Consider the following table definition, where image is a UDT:

```
CREATE TABLE history
  (id INTEGER
   ,information image );
```

Assuming an appropriate cast definition exists for the image UDT, the following statement converts the values in the information column to BYTE:

```
SELECT CAST (information AS BYTE(20))
FROM history
WHERE id = 100121;
```

## Related Information

- For details on expressions that can result in UDT data types, see SQL UDF in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.
- For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## UDT-to-Character Conversion

You can convert a UDT expression to a character string with either the CAST statement or Teradata conversion syntax.

## UDT-to-Character Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

### Syntax

```
CAST ( UDT_expression AS character_data_type )
```

### Syntax Elements

#### *UDT\_expression*

An expression that results in a UDT data type.

#### *character\_data\_type*

The data type, followed by any FORMAT, NAMED, or TITLE attribute phrases, to which the expression is to be converted.

## Teradata UDT-to-Character Conversion Syntax

### ANSI Compliance

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

### Syntax

```
UDT_expression ( [ data_attribute, [...] ] character_data_type [, data_attribute [,...] ] )
```

### Syntax Elements

#### *UDT\_expression*

An expression that results in a UDT data type.

#### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

#### *character\_data\_type*

The data type, for example CHAR OR VARCHAR, to which the expression is to be converted.

## Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement.

## Implicit Type Conversion

SQL Engine performs implicit UDT-to-character conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see [Usage Notes](#)) exists that specifies the AS ASSIGNMENT clause.



The target character type of the cast definition does not have to be an exact match to the target character type of the implicit conversion. Vantage can use an implicit cast definition that specifies a CHAR, VARCHAR, or CLOB target type.

If multiple implicit cast definitions exist for converting the UDT to different character types, Vantage uses the implicit cast definition for the character type with the highest precedence. The following list shows the precedence of character types in order from lowest to highest precedence:

- CHAR
- VARCHAR
- CLOB

If no UDT-to-character implicit cast definitions exist, Vantage looks for other cast definitions that can substitute for the UDT-to-character implicit cast definition:

IF the following combination of implicit cast definitions exists ...				THEN SQL Engine ...
UDT-to-numeric	UDT-to-DATE	UDT-to-TIME	UDT-to-TIMESTAMP	
X				uses the UDT-to-numeric implicit cast definition. If multiple UDT-to-numeric implicit cast definitions exist, then SQL Engine returns an SQL error.
	X			uses the UDT-to-DATE implicit cast definition.
		X		uses the UDT-to-TIME implicit cast definition.
			X	uses the UDT-to-TIMESTAMP implicit cast definition.
X	X			reports an error.
X		X		
X			X	
	X	X		
	X		X	
		X	X	
X	X	X		
X	X		X	
X		X	X	
	X	X	X	

IF the following combination of implicit cast definitions exists ...				THEN SQL Engine ...
UDT-to-numeric	UDT-to-DATE	UDT-to-TIME	UDT-to-TIMESTAMP	
X	X	X	X	

Substitutions are valid because SQL Engine can use the implicit cast definition to cast the UDT to the substitute data type, and then implicitly cast the substitute data type to a character type.

## Example

Consider the following table definition, where euro is a UDT:

```
CREATE TABLE euro_sales_table
  (quarter INTEGER
  ,region VARCHAR(20)
  ,sales euro );
```

Assuming an appropriate cast definition exists for the euro UDT, the following statement converts the values in the sales column to CHAR(10):

```
SELECT region, CAST (sales AS CHAR(10))
FROM euro_sales_table
WHERE quarter = 1;
```

## Related Information

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

## UDT-to-DATE Conversion

You can convert a UDT expression to a DATE value with either the CAST statement or Teradata conversion syntax.

## UDT-to-DATE Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

**Syntax**

```
CAST (
  UDT_expression AS DATE
  [ data_attribute [...] ]
)
```

**Syntax Elements*****UDT\_expression***

An expression that results in a UDT data type.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

**Teradata UDT-to-DATE Conversion Syntax****ANSI Compliance**

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

**Syntax**

```
UDT_expression ( [ data_attribute, [...] ] DATE [, data_attribute [,...] ] )
```

**Syntax Elements*****UDT\_expression***

An expression that results in a UDT data type.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement.

## Implicit Type Conversion

Performing an implicit data type conversion requires that an appropriate cast definition exists that specifies the AS ASSIGNMENT clause.

SQL Engine performs implicit UDT-to-DATE conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

If no UDT-to-DATE implicit cast definition exists, Vantage looks for other cast definitions that can substitute for the UDT-to-DATE implicit cast definition:

Combination of Implicit Cast Definitions		Description
UDT-to-Numeric	UDT-to-Character (non-CLOB)	
X		Vantage uses the UDT-to-numeric implicit cast definition. If multiple UDT-to-numeric implicit cast definitions exist, then SQL Engine returns an SQL error.
	X	Vantage uses the UDT-to-character implicit cast definition. If multiple UDT-to-character implicit cast definitions exist, then SQL Engine returns an SQL error.
X	X	SQL Engine reports an error.

Substitutions are valid because SQL Engine can use the implicit cast definition to cast the UDT to the substitute data type, and then implicitly cast the substitute data type to a DATE type.

## Example

Consider the following table definition, where datetime\_record is a UDT:

```
CREATE TABLE support
  (id INTEGER
   ,information datetime_record );
```

Assuming an appropriate cast definition exists for the `datetime_record` UDT, the following statement converts the values in the `information` column to `DATE`:

```
SELECT id, CAST (information AS DATE) FROM support;
```

## Related Information

For more information on `CREATE CAST`, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## UDT-to-INTERVAL Conversion

You can convert a UDT expression to an `INTERVAL` value with either the `CAST` statement or Teradata conversion syntax.

## UDT-to-INTERVAL Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, `CAST` permits the use of data attribute phrases such as `FORMAT`.

### Syntax

```
CAST ( UDT_expression AS interval_data_type )
```

### Syntax Elements

#### *UDT\_expression*

An expression that results in a UDT data type.

#### *interval\_data\_type*

The target predefined interval type followed by optional `NAMED` or `TITLE` attribute phrases.

## Teradata UDT-to-INTERVAL Conversion Syntax

### ANSI Compliance

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

## Syntax

```
UDT_expression ( [ data_attribute, [...] ] interval_data_type [, data_attribute, [...] ] )
```

## Syntax Elements

### *UDT\_expression*

An expression that results in a UDT data type.

### *data\_attribute*

One of the following data attributes:

- NAMED
- TITLE

### *interval\_data\_type*

The target predefined interval type to which UDT\_expression is to be converted.

## Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement.

## Implicit Type Conversion

Performing an implicit data type conversion requires a cast definition (see [Usage Notes](#)) that specifies the following:

- the AS ASSIGNMENT clause
- a target data type that is in the same INTERVAL family as the target of the implicit cast:

This INTERVAL data type ...	Belongs to this INTERVAL family ...
<ul style="list-style-type: none"> <li>• INTERVAL YEAR</li> <li>• INTERVAL YEAR TO MONTH</li> <li>• INTERVAL MONTH</li> </ul>	Year-Month
<ul style="list-style-type: none"> <li>• INTERVAL DAY</li> <li>• INTERVAL DAY TO HOUR</li> <li>• INTERVAL DAY TO MINUTE</li> <li>• INTERVAL DAY TO SECOND</li> <li>• INTERVAL HOUR</li> <li>• INTERVAL HOUR TO MINUTE</li> <li>• INTERVAL HOUR TO SECOND</li> </ul>	Day-Time

This INTERVAL data type ...	Belongs to this INTERVAL family ...
<ul style="list-style-type: none"> <li>• INTERVAL MINUTE</li> <li>• INTERVAL MINUTE TO SECOND</li> <li>• INTERVAL SECOND</li> </ul>	

The target data type of the cast definition does not have to be an exact match to the target of the implicit type conversion.

SQL Engine performs implicit UDT-to-INTERVAL conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

## Example

Consider the following table definition, where `datetime_record` is a UDT:

```
CREATE TABLE support
(id INTEGER
,information datetime_record );
```

Assuming an appropriate cast definition exists for the `datetime_record` UDT, the following statement converts the values in the `information` column to `INTERVAL MONTH`:

```
SELECT id, CAST (information AS INTERVAL MONTH) FROM support;
```

## Related Information

For more information on `CREATE CAST`, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## UDT-to-Numeric Conversion

You can convert a UDT expression to a numeric value with either the `CAST` statement or Teradata conversion syntax.

## UDT-to-Numeric Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

### Syntax

```
CAST ( UDT_expression AS numeric_data_type )
```

### Syntax Elements

#### *UDT\_expression*

An expression that results in a UDT data type.

#### *numeric\_data\_type*

The target predefined numeric type followed by any optional FORMAT, NAMED, or TITLE attribute phrases.

## Teradata UDT-to-Numeric Conversion Syntax

### ANSI Compliance

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

### Syntax

```
UDT_expression ( [ data_attribute, [...] ] numeric_data_type [, data_attribute [, [...] ] )
```

### Syntax Elements

#### *UDT\_expression*

An expression that results in a UDT data type.

#### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE



***numeric\_data\_type***

A predefined numeric type to which UDT\_expression is to be converted.

# Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement.

# Implicit Type Conversion

SQL Engine performs implicit UDT-to-numeric conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see [Usage Notes](#)) exists that specifies the AS ASSIGNMENT clause.

The target numeric type of the cast definition does not have to be an exact match to the target numeric type of the implicit conversion. Vantage can use an implicit cast definition that specifies a BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, NUMBER, or REAL/FLOAT/DOUBLE target type.

If multiple implicit cast definitions exist for converting the UDT to different numeric types, Vantage uses the implicit cast definition for the numeric type with the highest precedence. The following list shows the precedence of numeric types in order from lowest to highest precedence:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- DECIMAL/NUMERIC
- NUMBER
- REAL/FLOAT/DOUBLE

If no UDT-to-numeric implicit cast definitions exist, Vantage looks for other cast definitions that can substitute for the UDT-to-numeric implicit cast definition:

IF the following combination of implicit cast definitions exists ...		THEN SQL Engine ...
UDT-to-DATE	UDT-to-Character	
X		uses the UDT-to-DATE implicit cast definition.

IF the following combination of implicit cast definitions exists ...		THEN SQL Engine ...
UDT-to-DATE	UDT-to-Character	
	X	uses the UDT-to-character implicit cast definition. The character type cannot be CLOB. If multiple UDT-to-character implicit cast definitions exist, then SQL Engine returns an SQL error.
X	X	reports an error.

Substitutions are valid because SQL Engine can use the implicit cast definition to cast the UDT to the substitute data type, and then implicitly cast the substitute data type to a numeric type.

## Example

Consider the following table definition, where euro is a UDT:

```
CREATE TABLE euro_sales_table
(quarter INTEGER
,region VARCHAR(20)
,sales euro );
```

Assuming an appropriate cast definition exists for the euro UDT, the following statement converts the values in the sales column to DECIMAL(10,2):

```
SELECT SUM (CAST (sales AS DECIMAL(10,2))) FROM euro_sales_table;
```

## Related Information

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

## UDT-to-TIME Conversion

You can convert a UDT expression to a TIME value with either the CAST statement or Teradata conversion syntax.

## UDT-to-TIME Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

**Syntax**

```
CAST (
  UDT_expression AS TIME
  [ ( fractional_seconds_precision ) ]
  [ WITH TIME ZONE ]
  [ data_attribute [...] ]
)
```

**Syntax Elements*****UDT\_expression***

An expression that results in a UDT data type.

***fractional\_seconds\_precision***

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

**Teradata UDT-to-TIME Conversion Syntax****ANSI Compliance**

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

**Syntax**

```
UDT_expression (
  [ data_attribute, [...] ] TIME
  [ ( fractional_seconds_precision ) ]
  [, { data_attribute | WITH TIME ZONE } [,...] ]
)
```

## Syntax Elements

### *UDT\_expression*

An expression that results in a UDT data type.

### *data\_attribute*

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

### *fractional\_seconds\_precision*

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

## Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement.

## Implicit Type Conversion

SQL Engine performs implicit UDT-to-TIME conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see [Usage Notes](#)) exists that specifies the AS ASSIGNMENT clause.

If no UDT-to-TIME implicit cast definition exists, Vantage looks for a UDT-to-CHAR or UDT-to-VARCHAR cast definition that can substitute for the UDT-to-TIME implicit cast definition. Substitutions are valid because SQL Engine can use the implicit cast definition to cast the UDT to a character data type, and then implicitly cast the character data type to a DATE type. If multiple UDT-to-character implicit cast definitions exist, then SQL Engine returns an SQL error.

## Example

Consider the following table definition, where datetime\_record is a UDT:

```
CREATE TABLE support
(id INTEGER
,information datetime_record );
```

Assuming an appropriate cast definition exists for the `datetime_record` UDT, the following statement converts the values in the `information` column to `TIME WITH TIME ZONE`:

```
SELECT id, CAST (information AS TIME WITH TIME ZONE) FROM support;
```

## Related Information

For more information on `CREATE CAST`, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## UDT-to-TIMESTAMP Conversion

You can convert a UDT expression to a `TIMESTAMP` value with either the `CAST` statement or Teradata conversion syntax.

## UDT-to-TIMESTAMP Conversion with CAST

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, `CAST` permits the use of data attribute phrases such as `FORMAT`.

### Syntax

```
CAST (
  UDT_expression AS TIMESTAMP
  [ ( fractional_seconds_precision ) ]
  [ WITH TIME ZONE ]
  [ data_attribute [...] ]
)
```

### Syntax Elements

#### *UDT\_expression*

An expression that results in a UDT data type.

***fractional\_seconds\_precision***

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

## Teradata UDT-to-TIMESTAMP Conversion Syntax

### ANSI Compliance

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

### Syntax

```
UDT_expression (
  [ data_attribute, [...] ] TIMESTAMP
  [ ( fractional_seconds_precision ) ]
  [, { data_attribute | WITH TIME ZONE } [,...] ]
)
```

### Syntax Elements

***UDT\_expression***

An expression that results in a UDT data type.

***data\_attribute***

One of the following data attributes:

- FORMAT
- NAMED
- TITLE

***fractional\_seconds\_precision***

A single digit representing the number of significant digits in the fractional portion of the SECOND field. The valid range is 0 through 6. The default is 6.

## Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Implicit Type Conversion

SQL Engine performs implicit UDT-to-TIME conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see [Usage Notes](#)) exists that specifies the AS ASSIGNMENT clause.

If no UDT-to-TIME implicit cast definition exists, Vantage looks for a UDT-to-CHAR or UDT-to-VARCHAR cast definition that can substitute for the UDT-to-TIME implicit cast definition. Substitutions are valid because Vantage can use the implicit cast definition to cast the UDT to a character data type, and then implicitly cast the character data type to a DATE type. If multiple UDT-to-character implicit cast definitions exist, then SQL Engine returns an SQL error.

## Example

Consider the following table definition, where datetime\_record is a UDT:

```
CREATE TABLE support
  (id INTEGER
   ,information datetime_record );
```

Assuming an appropriate cast definition exists for the datetime\_record UDT, the following statement converts the values in the information column to TIME WITH TIME ZONE:

```
SELECT id, CAST (information AS TIME WITH TIME ZONE) FROM support;
```

## Related Information

- For details on expressions that can result in UDT data types, see SQL UDF in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

- For information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## UDT-to-UDT Conversion

Converts a UDT expression to a different UDT type.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

### Syntax

```
CAST ( UDT_expression AS UDT_data_type )
```

### Syntax Elements

#### *UDT\_expression*

An expression that results in a UDT data type.

#### *UDT\_data\_type*

The UDT type, followed by any FORMAT, NAMED or TITLE data attribute phrases, to which the expression is to be converted.

## Usage Notes

Explicit UDT-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement.

## Implicit Type Conversion

SQL Engine performs implicit UDT-to-UDT casts for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this document, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE



An implicit data type conversion involving a UDT can only be performed if the cast definition specifies the AS ASSIGNMENT clause. For more information, see [Implicit Type Conversions](#) and CREATE CAST in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Example

Consider the following table definitions, where euro and us\_dollar are UDTs:

```
CREATE TABLE euro_sales_table
  (euro_quarter INTEGER
  ,euro_region VARCHAR(20)
  ,euro_sales euro );
CREATE TABLE us_sales_table
  (us_quarter INTEGER
  ,us_region VARCHAR(20)
  ,us_sales us_dollar );
```

Assuming an appropriate cast definition exists for converting the euro UDT to a us\_dollar UDT, the following statement performs a us\_dollar UDT to euro UDT conversion:

```
INSERT INTO euro_sales_table
  SELECT us_quarter, us_region, CAST (us_sales AS euro)
  FROM us_sales_table;
```

## Related Information

- [CAST in Explicit Data Type Conversions](#)
- CREATE CAST in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144

# Data Type Conversion Functions

## TO\_BYTES

Decodes a sequence of characters in a given encoding into a sequence of bits. The following encodings are supported:

- BaseX
- BaseY
- Base64M (MIME)
- ASCII

where X is a power of 2 (for example, 2, 8, 16) and Y is not a power of 2 (for example, 10 and 36).

The result data type is VARBYTE or BLOB.

### ANSI Compliance

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

### Syntax

```
[TD_SYSFNLIB.] TO_BYTES ( instring, in_encoding )
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *instring*

The string value, that is, sequence of characters to be decoded.

#### *in\_encoding*

The encoding TO\_BYTES uses to return the sequence of characters specified by *instring*.

## Argument Types and Rules

Expressions passed to this function must have the following data types:

- *instring* = VARCHAR or CLOB
- *in\_encoding* = VARCHAR(64)

The two input parameters to TO\_BYTES must use the same character set. The parameters cannot use different character sets, such as Latin and Unicode.

If *in\_encoding* is not one of the supported encodings, an error is returned.

If either *instring* or *in\_encoding* is NULL, the result is NULL.

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.

The maximum input or output for CLOB or BLOB size is 2 GB. An error is reported if the size is exceeded.

The TO\_BYTES function converts from a character string to a VARBYTE. The character strings are treated as positive unless there is an explicit negative sign. Byte strings are interpreted as negative if the high order bit is 1.

TO\_BYTES should not be used for byte to byte encodings, since TO\_BYTES tries to interpret the sign of the byte string.

If you must use byte to byte conversion, use the two `TO_BYTE` functions. (Note the syntax is `TO_BYTE`, not `TO_BYTES`.)

## Examples

### Example: Decoding a Sequence of Characters to Base16

The following query:

```
SELECT CAST (TO_BYTES ('5A', 'base16') as BYTE (16));
TO_BYTES('5A','base16')
```

returns '5A-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00 '

### Example: Decoding a Sequence of Characters to Base36

The following query:

```
SELECT CAST (TO_BYTES ('-22EEVX', 'base36') as BYTE (16));
TO_BYTES ('-22EEVX', 'base36')
```

```
returns 'F8-8D-33-23-00-00-00-00-00-00-00-00-00-00-00'
```

## Related Information

For more information about compatible types, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## FROM\_BYTES

Encodes a sequence of bits into a sequence of characters representing its encoding. The following encodings are supported:

- BaseX
- BaseY
- Base64M (MIME)
- ASCII

where  $X$  is a power of 2 (for example, 2, 8, 16) and  $Y$  is not a power of 2 (for example, 10 and 36).

The result data type is VARCHAR or CLOB. Note that the return value may be NULL.

## ANSI Compliance

This statement is ANSI SQL:2011 compliant.

## Syntax

[TD\_SYSFNLIB.] TO\_BYTES ( *instring*, *out\_encoding* )

## Syntax Elements

**TD\_SYSFNLIB.**

Name of the database where the function is located.

***instring***

The binary string value, that is, the sequence of characters to be encoded.

***out\_encoding***

The encoding FROM\_BYTES uses to encode the sequence of characters specified by *instr*.

## Argument Types and Rules

Expressions passed to this function must have the following data types:

- *instring* = VARBYTE or BLOB

- `out_encoding = VARCHAR(64)`

If `out_encoding` is not one of the supported encodings, an error is returned.

If the resulting conversion overflows the result data type length, SQL Engine returns an error.

If either `instring` or `out_encoding` is NULL, the result is NULL.

The character set of the return type is set to either Latin or Unicode. The character set of the return type is set to match the character set of the encoding input parameter.

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.

---

The maximum input or output for CLOB or BLOB size is 2 GB. An error is reported if the size is exceeded.

The `FROM_BYTES` function converts from a `VARBYTE` and a character string. Byte strings are interpreted as negative if the high order bit is 1.

`FROM_BYTES` does not convert from byte to byte encodings, as the data is interpreted differently. Additionally, the byte encodings do not use positive or negative signs, which `FROM_BYTES` needs to decipher if the data is a positive or negative number.

If you must use byte to byte conversion, use two `TO_BYTE` functions. (Note the syntax is `TO_BYTE`, not `TO_BYTES`.)

## Examples

### Example: Encoding a Sequence of Base10 Bits into a Character Sequence

The following query:

```
SELECT from_bytes('5A1B'XB, 'base10');
```

returns '23067'.

### Example: Encoding a Sequence of ASCII Bits into a Character Sequence

The following query:

```
SELECT from_bytes('5A3F'XB, 'ASCII');
```

returns 'Z\ESC '.

### Example: Encoding a Sequence of Base16 Bits into a Character Sequence

The following query:

```
SELECT from_bytes('5A1B'XB, 'base16');
```

returns '5A1B'.

## Related Information

For more information about compatible types, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## TO\_NUMBER

Converts *string\_expr* to a NUMBER data type.

TO\_NUMBER is a scalar function whose return value data type is NUMBER.

### ANSI Compliance

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

### Syntax

```
[TD_SYSFNLIB.] TO_NUMBER (
  string_expr [, format_arg [, NLS_param ] ]
)
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *string\_expr*

A character expression. *string\_expr* contains a number in the format specified by *format\_arg*. If the conversion fails, NULL is returned.

#### *format\_arg*

A character expression.

If this syntax element is not valid, an error is returned.

If this syntax element is NULL, NULL is returned.

For more information, see [format\\_arg Format Elements](#).

### ***NLS\_param***

A character expression.

*NLS\_param* specifies characters that are returned by number format elements:

- Decimal character
- Group separator
- Local currency symbol
- Dual currency symbol
- International currency symbol

Valid values for *param* are:

- NUMERIC\_CHARACTERS = 'dg'
- CURRENCY = 'text'
- DUAL\_CURRENCY = 'text'
- ISO\_CURRENCY = 'text'

The characters *d* and *g* represent the decimal character and group separator respectively. They must be different single-byte characters. *Text* must be enclosed in apostrophes. Ten characters are available for the currency symbol.

The SDF (Specification for Data Formatting) file is used to determine any default formatting. If *NLS\_param* is specified, it overrides any defaults specified in the SDF file.

If *NLS\_param* is NULL, NULL is returned.

## **Argument Types and Rules**

Expressions passed to this function must be VARCHAR data types.

You can also pass arguments with data types that can be converted to the above data type 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.

---

**format\_arg Format Elements**

Format Element...	Example...	Returns ...
, (comma)	9,999	a comma in the specified position. <ul style="list-style-type: none"> <li>A comma cannot begin a number format.</li> <li>A comma cannot appear to the right of a decimal character or period in a number format.</li> </ul>
. (period)	9.99	a decimal point. You can only specify one period in a number format.
\$	\$9999	a value with a leading dollar sign.
0	09999 9990	leading zeros. trailing zeros.
9	9999	a value with the specified number of digits with a leading space if positive or with a leading minus if negative.
B	B9999	blanks for the integer part of a fixed point number when the integer part is zero.
C	C999	the ISO currency symbol as specified in the ISOCurrency element in the SDF file.
D	99D99	the character that separates the integer and fractional part of non-monetary values. This is specified in the RadixSeparator element in the SDF file.
EEEE	9.9EEEE	a value in scientific notation.
G	9G999	(group separator) the character that separates groups of digits in the integer part of non-monetary values. This is specified in the GroupSeparator element in the SDF file.
L	L999	(local currency) the string representing the local currency as specified in the Currency element in the SDF file.
MI	9999MI	a trailing minus sign if the value is negative. The MI format element can appear only in the last position of a number format.
PR	9999PR	a negative value in <angle brackets>, or a positive value with a leading and trailing blank. The PR format element can appear only in the last position of a number format.
S	S9999 9999S	a negative value with a leading or trailing minus sign. a positive value with a leading or trailing plus sign. The S format element can appear only in the first or last position of a number format.



Format Element...	Example...	Returns ...
TM	TM TM9 TME	(text minimum format) returns the smallest number of characters possible. This element is case insensitive. TM or TM9 return the number in fixed notation unless the output exceeds 64 characters. If the output exceeds 64 characters, the number is returned in scientific notation. TME returns the number in scientific notation with the smallest number of characters. You cannot precede this element with an other element. You can follow this element only with one 9 or one E (or e), but not with any combination of these.
U	U9999	(dual currency) the string that represents the dual currency as specified in the DualCurrency element in the SDF file.
V	999V99	a value multiplied by 10 to the n (and, if necessary, rounded up), where <i>n</i> is the number of 9's after the V.
X	XXXXX xxxxx	the hexadecimal value of the specified number of digits. If the specified number is not an integer, the function will round it to an integer. This element accepts only positive values or zero. Negative values return an error. You can precede this element only with zero (which returns leading zeros) or FM. Any other elements return an error. If you do not specify zero or FM, the return always has one leading blank.

## Usage Notes

The following are examples of the TO\_NUMBER syntax that Teradata supports. The examples deviate from the TO\_NUMBER syntax Oracle supports. The function:

- Does not support not having a closing angle bracket for the PR format element. This example returns NULL because the number is not in the correct format.:

```
SELECT TO_NUMBER ( '<123', '999PR' )
```

- Allows a space at the end of the number if the PR format element is used and the number is positive. This example returns 123:

```
SELECT TO_NUMBER ( ' 123 ', '999PR' )
```

- Allows multiple integer digits in scientific notation. This example returns 12000:

```
SELECT TO_NUMBER ( '12E3', '99EEEE' )
```

- Allows commas in the integer portion of scientific notation. This example returns 1234000:

```
SELECT TO_NUMBER ( '1,234E3', '9,999EEEE' )
```

- Validates commas when there are two arguments, even when there is no decimal in the format. This example returns NULL because the number is not in the correct format:

```
SELECT TO_NUMBER ( '1234', '9,999' )
```

- Treats an empty string like any other string. This example returns an error, rather than NULL, because the format is not valid:

```
SELECT TO_NUMBER ( '', 'BAD Format' )
```

- Allows the use of period (.) and comma (,) with D and G are in the same format string. This example returns 1234.99:

```
SELECT TO_NUMBER ( '1,234.99', '9,999D99' )
```

## Examples

### Example: Converting *string\_expr* to a NUMBER Data Type

The following query returns the result 1789.96:

```
SELECT TO_NUMBER ( '1789.96', '9999.99' );
```

### Example: Converting a Dollar Amount to a NUMBER Data Type

The following query returns 123:

```
SELECT TO_NUMBER('dollar123','L999','NLS_CURRENCY=''dollar'');
```

## Related Information

For more information about compatible types, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## TO\_CHAR(Numeric)

Converts *numeric\_expr* to a character string.

TO\_CHAR(Numeric) is a scalar function whose return data type is VARCHAR CHARACTER SET UNICODE.

## ANSI Compliance

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

## Syntax

```
[TD_SYSFNLIB.] TO_CHAR (
  numeric_expr [, format_arg [, NLS_param ] ]
)
```

## Syntax Elements

### TD\_SYSFNLIB.

Name of the database where the function is located.

### *numeric\_expr*

A numeric argument.

If the conversion fails, '#' characters are returned. When the integer portion of the number is less than the digits specified in the format string, a string with a variable number of '#' characters is returned. The number of '#' characters returned is equal to the length of the maximum possible character string result based on the format string or data type of *numeric\_expr*. This includes a '#' character representing the optional plus/minus sign.

For example:

```
SELECT TO_CHAR(12345678, '99')
```

returns '###'.

```
SELECT TO_CHAR(12345678, '99999')
```

returns '#####'.

```
SELECT TO_CHAR(12345678, '99V9')
```

returns '#####'.

### *format\_arg*

A character expression.

*format\_arg* is used to format the numeric values. If *format\_arg* is omitted, *numeric\_expr* is converted to a string exactly long enough to hold its significant digits.

If this syntax element is not valid, an error is returned.

For `format_arg` values, see [TO\\_NUMBER](#).

### ***NLS\_param***

A character expression.

`NLS_param` specifies characters that are returned by number format elements:

- Decimal character
- Group separator
- Local currency symbol
- Dual currency symbol
- International currency symbol

Valid values for `param` are:

- `NUMERIC_CHARACTERS = 'dg'`
- `CURRENCY = 'text'`
- `DUAL_CURRENCY = 'text'`
- `ISO_CURRENCY = 'text'`

The characters *d* and *g* represent the decimal character and group separator respectively. They must be different single-byte characters. *Text* must be enclosed in apostrophes. Ten characters are available for the currency symbol.

The SDF (Specification for Data Formatting) file is used to determine any default formatting. If `NLS_param` is specified, it overrides any defaults specified in the SDF file.

If `NLS_param` is NULL, NULL is returned.

For `NLS_param` values, see [TO\\_NUMBER](#).

## **Argument Types and Rules**

Expressions passed to this function must have the following data types:

- `numeric_expr` = BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER
- `format_arg` = CHAR or VARCHAR
- `NLS_param` = CHAR or VARCHAR

## **Example: Converting a numeric\_expr to a Character String**

The following query:

```
SELECT TO_CHAR(50000, '$99,999.99');
```

returns '\$50,000.00'.

## TO\_CHAR(DateTime)

Converts a *date\_timestamp\_value* to a character string.

TO\_CHAR(DateTime) is a scalar function whose return data type is VARCHAR CHARACTER SET UNICODE.

### ANSI Compliance

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

### Syntax

```
[TD_SYSFNLIB.] TO_CHAR (
    date_timestamp_value [, format_arg ]
)
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *date\_timestamp\_value*

A datetime or interval argument.

#### *format\_arg*

A character expression.

For valid *format\_arg* values, see [TO\\_DATE](#).

*format\_arg* is used to format the numeric values. If *format\_arg* is omitted, *numeric\_expr* is converted to a string exactly long enough to hold its significant digits.

If this syntax element is not valid, an error is returned.

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, TIME, TIME WITH TIME ZONE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, or an Interval data type, such as INTERVAL YEAR TO MONTH.
  - *format\_arg* = CHAR or VARCHAR

- Digits in fractional seconds are truncated in the timestamp value depending on the result format string that is specified.

## Example: Converting a `date_timestamp_value` to a Character String

The following query:

```
SELECT TO_CHAR(DATE '2003-12-23', 'DD-MON-YYYY');
```

returns a result of '23-DEC-2003'.

## Example: Truncating Digits in Fractional Seconds

This is an example of truncating digits in fractional seconds according to the requested format of `FF[1..9]` for milliseconds in the timestamp value:

```
select to_char(timestamp '2020-09-01 23:45:09.999999', 'yyyy-mm-dd hh:mi:ss.FF1');
```

The result is:

```
to_char(2020-09-01 23:45:09.999999, 'yyyy-mm-dd hh:mi:ss.FF1')
-----
2020-09-01 11:45:09.9
```

## TO\_DATE

Converts *string\_expr* to a DATE data type.

TO\_DATE does not convert data to any of the other datetime data types. Do not use the TO\_DATE function with a DATE value for *string\_expr*. The first two digits of the returned DATE value can differ from the original *string\_expr* depending on *format\_arg* or the default date format.

TO\_DATE is an overloaded scalar function whose return value data type is DATE.

### ANSI Compliance

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

### Syntax

```
[TD_SYSFNLIB.] TO_BYTES ( string_expr, format_arg )
```

## Syntax Elements

### TD\_SYSFNLIB.

Name of the database where the function is located.

### *string\_expr*

A character expression.

If the conversion fails, TO\_DATE returns an error.

### *format\_arg*

A character argument.

*format\_arg* is a data format specifying the format of *string\_expr*.

If *format\_arg* is omitted, the following default date format is used: YYYY-MM-DD

If *format\_arg* is NULL, TO\_DATE returns NULL.

If *format\_arg* is not valid, an error is returned.

For more information, see [format\\_arg Format Elements](#).

## Argument Types and Rules

Expressions passed to this function must be a VARCHAR data type.

You can also pass arguments with data types that can be converted to the above type 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.

## format\_arg Format Elements

Element	Description
- / , . ; :	Punctuation characters are ignored and text enclosed in quotation marks is ignored.

Element	Description
"text"	
AD A.D.	AD indicator.
AM A.M.	Meridian indicator.
BC B.C.	BC indicator.
CC SCC	<p>Century.</p> <p>If the last 2 digits of a 4-digit year are between 01 and 99 inclusive, the century is 1 greater than the first 2 digits of that year.</p> <p>If the last 2 digits of a 4-digit year are 00, the century is the same as the first 2 digits of that year.</p> <p><b>Note:</b> CC and SCC are only supported by the TO_CHAR function.</p>
D	Day of week (1-7).
DAY	Name of day.
DD	Day of month (1-31).
DDD	Day of year (1-366).
DL	<p>Date Long. Equivalent to the format string 'FMDay, Month FMDD, YYYY'.</p> <p><b>Note:</b> DL is only supported by the TO_CHAR function.</p>
DS	<p>Date Short. Equivalent to the format string 'FMMM/DD/YYYYFM'.</p> <p><b>Note:</b> DS is only supported by the TO_CHAR function.</p>
DY	abbreviated name of day.
E	not supported.
EE	not supported.
FF [1..9]	<p>Fractional seconds.</p> <p>Use [1..9] to specify the number of fractional digits.</p> <p>FF without any number following it prints a decimal followed by digits equal to the number of fractional seconds in the input data type. If the data type has no fractional digits, FF prints nothing.</p> <p><b>Note:</b> Any fractional digits beyond 6 digits are truncated.</p>



Element	Description
FM	Format Minimum mode. The value is returned with no leading or trailing blanks. This may be toggled on and off by adding an 'FM' before and after a section that should use minimum space. <b>Note:</b> FM is only supported by the TO_CHAR function.
FX	Requires the character data and the format model to be an exact match. <b>Note:</b> FX is not supported by the TO_CHAR function.
HH HH12	Hour of day (1-12).
HH24	Hour of the day (0-23).
IW	Week of year (1-52 or 1-53) based on ISO model. <b>Note:</b> IW is only supported by the TO_CHAR function.
IYY IY I	Last 3, 2, or 1 digits of ISO year. <b>Note:</b> I, IY, and IYY are only supported by the TO_CHAR function.
IYYY	4-digit year based on the ISO standard. <b>Note:</b> IYYY is only supported by the TO_CHAR function.
J	Julian day, the number of days since January 1, 4713 BC. Number specified with J must be integers. Teradata uses the Gregorian calendar in calculations to and from Julian Days.
MI	Minute (0-59).
MM	Month (01-12).
MON	Abbreviated name of month.
MONTH	Name of month.
PM P.M.	Meridian indicator.
Q	Quarter of year (1, 2, 3, 4). <b>Note:</b> Q is only supported by the TO_CHAR function.
RM	Roman numeral month (I - XII).

Element	Description
RR	Stores 20th century dates in the 21st century using only 2 digits. If the current year and the specified year are both in the range of 0-49, the date is in the current century. <b>Note:</b> RR is not supported by the TO_CHAR function.
RRRR	Round year. Accepts either 4-digit or 2-digit input. 2-digit input provides the same return as RR. <b>Note:</b> RRRR is not supported by the TO_CHAR function.
SP	Spelled. Any numeric element followed by SP is spelled in English words. The words are capitalized according to how the element is capitalized. For example: 'DDDSP' specifies all uppercase, 'DddSP' specifies that the first letter is capitalized, and 'dddSP' specifies all lowercase. <b>Note:</b> SP is only supported by the TO_CHAR function.
SS	Second (0-59).
SSSSS	Seconds past midnight (0-86399).
TS	Time Short. Equivalent to the format string 'HH:MI:SS AM'. <b>Note:</b> TS is only supported by the TO_CHAR function.
TZD	not supported.
TZH	Time zone hour.
TZM	Time zone minute.
TZR	Time zone region. Equivalent to the format string 'TZH:TZM'. <b>Note:</b> TZR is only supported by the TO_CHAR function.
WW	Week of year (1-53) where week 1 starts on the first day of the year and continues to the 7th day of the year. <b>Note:</b> WW is only supported by the TO_CHAR function.
W	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh. <b>Note:</b> W is only supported by the TO_CHAR function.
X	Local radix character.
Y,YYY	Year with comma in this position.

Element	Description
YEAR SYEAR	Year, spelled out. S prefixes BC dates with a minus sign. <b>Note:</b> YEAR and SYEAR are only supported by the TO_CHAR function.
YYYY SYYYY	4-digit year. S prefixes BC dates with a minus sign.
YYY YY Y	Last 3, 2, or 1 digit of year. If the current year and the specified year are both in the range of 0-49, the date is in the current century.

## Example: Converting a string\_expr to a DATE Data Type

The following query:

```
SELECT TO_DATE ('January 15, 1989, 11:00 A.M.', 'Month dd, YYYY, HH: MI A.M.');
```

returns the result 15-JAN-89.

## Related Information

For more information about compatible types, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## TO\_TIMESTAMP

Converts *string\_expr* or *integer\_expr* to a TIMESTAMP data type.

TO\_TIMESTAMP is an overloaded scalar function whose return value data type is TIMESTAMP(6).

### ANSI Compliance

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

### Syntax

```
[TD_SYSFNLIB.] TO_TIMESTAMP ( { string_expr, format_arg | integer } )
```

### Syntax Elements

**TD\_SYSFNLIB.**

Name of the database where the function is located.

***string\_expr***

A character argument.

If the conversion fails, TO\_TIMESTAMP returns an error.

***format\_arg***

A character expression.

*format\_arg* is a data format specifying the format of *string\_expr*.

For valid *format\_arg* values, see [TO\\_DATE](#).

Any fractional seconds that are specified beyond 6 digits are truncated.

If *format\_arg* is omitted, TO\_TIMESTAMP uses a default timestamp format of YYYY-MM-DD HH24:MI:SS.FF6.

If this syntax element is NULL, NULL is returned.

If this syntax element is not valid, an error is returned.

***integer\_expr***

a number argument.

POSIX epoch conversion is implicit in the TO\_TIMESTAMP function when *integer\_expr* is passed to the function.

POSIX epoch is the number of seconds that have elapsed since midnight Coordinated Universal Time (UTC) of January 1, 1970.

## Argument Types and Rules

Expressions passed to this function must be of VARCHAR data type or the following Numeric data types:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT

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.

---

## Examples

### Example: Converting *string\_expr* or *integer\_expr* to a **TIMESTAMP** Data Type

The following query:

```
SELECT TO_TIMESTAMP ( 'January 15, 1989, 11:00 A.M.', 'Month dd, YYYY, HH:
MI A.M.' );
```

returns the result 1989-01-15 11:00:00.000000.

### Example: Converting a **POSIX Epoch**

The following query, an example of POSIX epoch conversion:

```
SELECT TO_TIMESTAMP(242525);
```

returns the result 1970-01-03 19:22:05.000000

## Related Information

For more information about compatible types, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## TO\_TIMESTAMP\_TZ

Converts *string\_expr* to a **TIMESTAMP WITH TIME ZONE** data type.

TO\_TIMESTAMP\_TZ is an overloaded scalar function whose return value data type is **TIMESTAMP(6) WITH TIME ZONE**.

### ANSI Compliance

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

### Syntax

```
[TD_SYSFNLIB.] TO_TIMESTAMP_TZ ( string_expr [, format_arg ] )
```

## Syntax Elements

### TD\_SYSFNLIB.

Name of the database where the function is located.

### *string\_expr*

A character argument.

If the conversion fails, TO\_TIMESTAMP\_TZ returns an error.

### *format\_arg*

A character expression.

*format\_arg* is a data format specifying the format of *string\_expr*.

*format\_arg* does not convert data to any of the other datetime data types.

Any fractional seconds that are specified beyond 6 digits are truncated.

If *format\_arg* is omitted, a default timestamp with time zone format of YYYY-MM-DD HH24:MI:SS.FF6 TZH:TZM is used.

For valid *format\_arg* values, see [TO\\_DATE](#).

If this syntax element is NULL, NULL is returned.

If this syntax element is not valid, an error is returned.

## Argument Types and Rules

Expressions passed to this function must be a VARCHAR data type.

You can also pass arguments with data types that can be converted to the above type 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.

---

## Example: Converting *string\_expr* to a TIMESTAMP WITH TIME ZONE Data Type

The following query:

```
SELECT TO_TIMESTAMP_TZ ('January 15, 1989, 11:00 A.M.', 'Month dd, YYYY, HH:MI A.M.');
```

returns the result 1989-01-15 11:00:00.000000+00:00.

## Related Information

For more information about compatible types, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## TO\_YMINTERVAL

Converts *string\_value* specified as either:

- SQL interval format
- ISO duration format

into an INTERVAL YEAR(4) TO MONTH value.

Any value that overflows the INTERVAL(4) YEAR TO MONTH value results in an error.

The return value data type is INTERVAL YEAR(4) TO MONTH.

### ANSI Compliance

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

### Syntax

```
[TD_SYSFNLIB.] TO_YMINTERVAL ( string_value )
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *string\_value*

A character expression.

If *string\_value* is an empty string or NULL, NULL is returned.

For more information, see [string\\_value Formats](#).

## Argument Types and Rules

Expressions passed to this function must have one of the following data types:

CHAR, VARCHAR

## string\_value Formats

- SQL Interval Format

The format for an interval year to month is [+ | -]YY-MM. YY is an integer between 0 and 9999, and MM is an integer between 0 and 11.

- ISO Duration Format

The format for a duration is [-]P[n]Y[n]M[n] where n specifies the value of the element (for example, 4Y is 4 years).

This represents a subset of the ISO duration format and the use of any arguments other than P, Y, and M are ignored. Leading and trailing zeroes are optional. Spaces are not allowed within the element format. Elements may be omitted if they are zero, but at least one element and its number is required.

A leading negative sign denotes a negative interval. If omitted, the interval is positive. A positive sign is not allowed.

Element values are integers between 0 and any value that will not cause an interval overflow. The values can exceed their normal time limit. For example, P14M represents 14 months, even though there are 12 months in a year. The ISO duration letter designators have the following meaning (only the P, Y, and M designators are allowed for the YEAR to MONTHS interval):

ISO Designator	Definition
P	Duration. It is always at the beginning.
Y	Year. It follows the number of years.
M	Month. It follows the number of months.

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.

---

## Examples

### Example: Converting *string\_value* Specified as an SQL Interval Format

The following query, which specifies an SQL interval format:



```
SELECT TO_YMINTERVAL('04-10');
```

returns the result '04-10'.

### Example: Converting *string\_value* Specified as an ISO duration

The following query, which specifies an ISO duration:

```
SELECT TO_YMINTERVAL('P100Y4M');
```

returns the result '100-04', which is 100 years, 4 months.

### Example: *string\_value* Error

The following query:

```
SELECT TO_YMINTERVAL('P20Y3M4DT12H23M34.234S');
```

returns an error because only the P, Y, and M designators are allowed.

### Example: Converting *string\_value* Specified as an ISO duration of -14 months

The following query, which specifies an ISO duration of -14 months:

```
SELECT TO_YMINTERVAL('-P14M');
```

returns the result '-01-02', which is negative 1 year, 2 months.

## Related Information

For more information about compatible types, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## TO\_DSINTERVAL

Converts *string\_value* specified as either of the following:

- SQL interval format
- ISO duration format

into an INTERVAL DAY(4) TO SECOND(6) value.

Any value that overflows the INTERVAL DAY(4) TO SECOND(6) value results in an error.

The return value data type is INTERVAL DAY(4) TO SECOND(6).

## ANSI Compliance

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

## Syntax

```
[TD_SYSFNLIB.] TO_DSINTERVAL ( string_value )
```

## Syntax Elements

### TD\_SYSFNLIB.

Name of the database where the function is located.

### *string\_value*

A character expression.

If *string\_value* is an empty string or NULL, NULL is returned.

For more information, see [string\\_value Formats](#).

## Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- CHAR
- VARCHAR

## string\_value Formats

- SQL Interval Format

The format for an interval day to second is [+ | -]DD HH:MM:SS[.xxxxxx]. One or more spaces separate the day from hour element. There can be any number of spaces between elements. DD is an integer between 0 and 9999, HH is an integer between 0 and 23, MM is an integer between 0 and 59, and SS is an integer between 0 and 59. The optional fractional seconds can be from .0 to .999999.

- ISO Duration Format

The format for a duration is [-]P[n]DT[n]H[n]M[n].[frac\_secs]S where n specifies the value of the element (for example, 4H is 4 hours). This represents a subset of the ISO duration format and any other duration letters (including valid ISO letters such as Y and M) result in an error.

Leading and trailing zeros are optional. Spaces are not allowed within the element format. Elements may be omitted if they are zero, but at least one element and its number is required.

A leading negative sign is used to denote a negative interval. If omitted, the interval is positive. A positive sign is not allowed. Element values are integers between 0 and any value that will not cause an interval overflow. Fractional seconds are optional and can be between .0 and .999999.

The T designator always precedes the time elements (and removes the ambiguity of M between months and minutes within the full ISO duration syntax). If there are no H, M, or S time elements, T is omitted.

Values can exceed their normal time limit. For example, PT72H represents 72 hours even though there are 24 hours in a day. The ISO duration letter designators have the following meaning (only the P, D, T, H, M, and S designators are allowed for the DAY to SECONDS interval):

ISO Designator	Definition
P	Duration. It is always at the beginning.
D	Day. It follows the number of days.
T	Time. It precedes the time elements H, M, and S.
H	Hour. It follows the number of hours.
M	Minutes. It follows the number of minutes.
S	Second. It follows the number of seconds.

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.

---

## Examples

### Example: Converting *string\_value* specified as an SQL Interval Format

The following query, which specifies an SQL interval format:

```
SELECT TO_DSINTERVAL('100 04:23:59');
```

returns the result '100 04:23:59.000000'.

### Example: Converting Another *string\_value* specified as an SQL Interval Format

The following query, which specifies an SQL interval format:

```
SELECT TO_DSINTERVAL( '100 04:23:59.543' );
```

returns the result '100 04:23:59.543000'.

### Example: Converting *string\_value* specified as an ISO Duration

The following query, which specifies an ISO duration:

```
SELECT TO_DSINTERVAL( 'P100DT4H23M59S' );
```

returns the result '100 04:23:59.000000'.

### Example: Converting *string\_value* specified as an ISO Duration of 73 hours

The following query, which specifies an ISO duration of 73 hours:

```
SELECT TO_DSINTERVAL( 'PT73H' );
```

returns the result '3 01:00:00.000000', which is 3 days, 1 hours.

### Example: Converting *string\_value* Error

The following query:

```
SELECT TO_DSINTERVAL( 'P2MT12H' );
```

returns an error because the month designator, '2M', isn't allowed.

## Related Information

For more information about compatible types, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## NUMTODSINTERVAL

Converts a *numeric\_value* into an INTERVAL DAY(4) TO SECOND(6) value.

The return value data type is INTERVAL DAY(4) TO SECOND(6).

### ANSI Compliance

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

### Syntax

```
[TD_SYSFNLIB.] NUMTODSINTERVAL ( numeric_value, interval_unit )
```

## Syntax Elements

### TD\_SYSFNLIB.

Name of the database where the function is located.

### *numeric\_value*

A numeric argument.

If this syntax element is NULL, NULL is returned.

### *interval\_unit*

A character expression.

*interval\_unit* specifies the unit of value for *numeric\_value*. It must be one of the following case-insensitive values:

- 'DAY'
- 'HOUR'
- 'MINUTE'
- 'SECOND'

Any value that overflows the INTERVAL DAY(4) TO SECOND(6) value results in an error.

If this syntax element is NULL, NULL is returned.

## Argument Types and Rules

Expressions passed to this function must have the following data types:

- *numeric\_value* = BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER
- *interval\_unit* = CHAR or VARCHAR

## Examples

### Example: Converting *numeric\_value* into an INTERVAL DAY(100) TO SECOND Value

The following query:

```
SELECT NUMTODSINTERVAL(100, 'DAY');
```

returns the result '100 00:00:00.000000', which is 100 days.

**Example: Converting *numeric\_value* into an INTERVAL DAY(100) TO HOURS Value**

The following query:

```
SELECT NUMTODSINTERVAL(100, 'HOURS');
```

returns the result '4 04:00:00.000000', which is 4 days, 4 hours.

**Example: Converting *numeric\_value* into an INTERVAL DAY(100) TO MINUTES Value**

The following query:

```
SELECT NUMTODSINTERVAL(1600, 'MINUTES');
```

returns the result '1 02:40:00.000000', which is 1 day, 2 hours, 40 minutes.

**Example: Converting *numeric\_value* into an INTERVAL DAY(100) TO HOURS Value**

The following query:

```
SELECT NUMTODSINTERVAL(3.5, 'HOURS');
```

returns the result 3 hours, 30 minutes.

## NUMTOYMINTERVAL

Converts *numeric\_value* into an INTERVAL YEAR(4) TO MONTH value.

The return value data type is INTERVAL YEAR(4) TO MONTH.

**ANSI Compliance**

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

**Syntax**

```
[TD_SYSFNLIB.] NUMTOYMINTERVAL ( numeric_value, interval_unit )
```

**Syntax Elements**

**TD\_SYSFNLIB.**

Name of the database where the function is located.

***numeric\_value***

A numeric argument.

If this syntax element is NULL, NULL is returned.

***interval\_unit***

A character expression.

*interval\_unit* specifies the unit of value for *numeric\_value*. It must be one of the following case-insensitive values:

- 'YEAR'
- 'MONTH'

Any value that overflows the INTERVAL YEAR(4) TO MONTH value results in an error.

If this syntax element is NULL, NULL is returned.

## Argument Types and Rules

Expressions passed to this function must have the following data types:

- *numeric\_value* = BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER
- *exponent\_value* = CHAR OR VARCHAR

## Examples

### Example: Converting numeric\_value into an INTERVAL YEAR(40) TO MONTH Value

The following query:

```
SELECT NUMTOYMINTERVAL(40, 'YEAR');
```

returns the result '40-00', which is 40 years, 0 months.

### Example: Converting numeric\_value into an INTERVAL YEAR(100) TO MONTH Value

The following query:

```
SELECT NUMTOYMINTERVAL(100, 'MONTHS');
```

returns the result '08-04', which is 8 years, 4 months.



# Default Value Control Phrases

The following sections describe the default value control phrases and the rules and guidelines on how to use them.

## Using Default Value Control Phrases

A default value control phrase determines the action to take when you do not supply a value for a field.

### Rules and Guidelines

The following rules and guidelines apply to default value control phrases:

- Default value control phrases are only valid with the following data definitions.
  - Fields defined in CREATE TABLE and ALTER TABLE statements
  - Parameters defined in CREATE MACRO and REPLACE MACRO statements
- Default value controls are *not* effective for views and expressions.
- The default value for a field is null unless you specify NOT NULL, in which case there is no default value if you do not specify the DEFAULT phrase.
- The presence of nulls can affect query performance negatively; therefore, consider using the NOT NULL default value phrase unless you intend to have nulls in the column.

The default control value phrases are documented individually in the remaining topics in this section.

## NOT NULL Phrase

Specifies that the fields of a column must contain a value; they cannot be null.

### ANSI Compliance

NOT NULL is ANSI SQL:2011 compliant.

### Syntax

```
NOT NULL
```

### Usage Notes

The NOT NULL specification for a column ensures that all fields of that column will contain a value. An error is returned if you attempt to insert row data that does not have a value for the column, or has a NULL for the column.

If a NOT NULL phrase is specified for a column in an ALTER TABLE statement, a DEFAULT or a WITH DEFAULT phrase must also be specified unless the table is empty. This ensures that any empty field in that column will be supplied with default values in compliance with the NOT NULL specification.

**Recommendation:** As a best practice, consider specifying NOT NULL for all columns unless there is a valid reason to allow nulls.

If a column should contain nulls, be sure you understand what NULL means for the column since NULL has many interpretations. In some cases, it is better to define a non-null default value rather than allow nulls. If a non-null default value is specified for the column, consider specifying NOT NULL for the column also.

If a column is nullable (even if no nulls are actually put in the column), every row will have a presence bit for the column to indicate its nullability. This may cause an increase in row size if an additional presence byte is required to accommodate this presence bit. If the column should not have nulls, specifying NOT NULL eliminates the need for the presence bit and potentially saves a byte per row.

Specifying NOT NULL also allows for better optimization of some queries since additional processing to handle nulls, or the potential of nulls in a column, can be avoided.

For more information on the advantages and disadvantages of using nulls, see *Teradata Vantage™ - Database Design*, B035-1094.

### Example: NOT NULL Phrase

In the following table definition, the FullName column is defined as NOT NULL, thus ensuring that every row in the Members table has a value for FullName.

```
CREATE TABLE Members(
  FullName VARCHAR(42) NOT NULL,
  Status CHAR(6),
  Phone CHAR(10));
```

## DEFAULT Phrase

Specifies that a user-defined default value is to be inserted in the field when a value is not specified for a column in an INSERT statement.

### ANSI Compliance

DEFAULT is ANSI SQL:2011 compliant.

Also see the non-ANSI [WITH DEFAULT Phrase](#).

### Syntax

```
DEFAULT {
  constant_value |
  { DATE | TIME | TIMESTAMP } quotestring |
```

```
INTERVAL [ sign ] quotestring qualifier
}
```

## Syntax Elements

### ***constant\_value***

A default value to be inserted when a column in the target table is omitted from the `column_list` specification of an INSERT statement.

When no DEFAULT phrase is provided, a NULL is inserted into the field for nullable columns, otherwise an error is returned.

A default value can be a keyword or built-in function. For more information on built-in functions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

The following defines the valid keywords and functions:

- `number` – specifies to insert a numeric literal value as the default value
- `NULL` – specifies to insert a null (for nullable columns) as the default value
- `USER` – specifies to insert the user name of the current user as the default value
- `CURRENT_DATE` – specifies to insert the current system date as the default value
- `CURRENT_TIME` – specifies to insert the current system time as the default value
- `CURRENT_TIMESTAMP` – specifies to insert the current system date and time as the default value

### **DATE**

A date value in ANSI DATE literal format as the insert default.

### **TIME**

A time value in ANSI TIME literal format as the insert default.

### **TIMESTAMP**

A timestamp value in ANSI TIMESTAMP literal format as the insert default.

### **INTERVAL**

An interval value in ANSI INTERVAL format as the insert default.

### ***quotestring***

A default value represented as a string enclosed in apostrophes.

### ***sign***

The sign of an interval value.

***qualifier***

One of the following interval time periods:

- YEAR
- YEAR TO MONTH
- MONTH
- DAY
- DAY TO HOUR
- DAY TO MINUTE
- DAY TO SECOND
- HOUR
- HOUR TO MINUTE
- HOUR TO SECOND
- MINUTE
- MINUTE TO SECOND
- SECOND

## Usage Notes

### **DEFAULT and INSERT**

If you attempt to insert a row expression of the form DEFAULT VALUES, an error is returned if any column in the table is not defined with a DEFAULT value.

If you attempt to insert an explicit *column\_list* in which a value is omitted, an error is returned if no DEFAULT is defined for the omitted column and if the column was declared NOT NULL.

### **DEFAULT and Period Data Types**

The following data type attributes are supported for a Period column:

- DEFAULT NULL
- DEFAULT value

The specified value must be either a Period literal or a Period value constructor.

When a Period value constructor is used for specifying the default value, the following rules apply:

- A Period value constructor with single argument must not be used as a default value; otherwise, an error is reported.
- The beginning bound must be specified using a DateTime literal, DATE, CURRENT\_DATE, or CURRENT\_TIMESTAMP[(n)]; otherwise, an error is reported.
- If the beginning bound is a DateTime literal, the ending bound must be specified using a DateTime literal or, if the beginning bound has a DATE or TIMESTAMP data type, UNTIL\_CHANGED. Otherwise, an error is reported.

- If the beginning bound is DATE, CURRENT\_DATE, or CURRENT\_TIMESTAMP[(n)], the ending bound must be UNTIL\_CHANGED or a DateTime literal specifying the equivalent value of UNTIL\_CHANGED; otherwise, an error is reported.

The following data type attributes are not supported for a Period column:

- DEFAULT USER
- DEFAULT DATE
- DEFAULT TIME
- DEFAULT CURRENT\_DATE
- DEFAULT CURRENT\_TIME[(n)]
- DEFAULT CURRENT\_TIMESTAMP[(n)]

**Column Data Types and DEFAULT Values**

A default value must be compatible with the data type specified for the column it is inserted into. For example, the phrase INTEGER DEFAULT 3.5 is not valid and returns an error.

For DateTime data types, SQL Engine performs an implicit conversion if the default value specified in a CREATE/ALTER TABLE statement differs from the data type of the column. For example, the following statement is valid:

```
CREATE TABLE tab1 (F1 INT, F2 TIMESTAMP(0) DEFAULT CURRENT_DATE);
```

The following table lists the default values you can specify for each of the column types.

Column Data Type	Supported Default Values
DATE	CURRENT_DATE
	CURRENT_TIMESTAMP
	DATE literal
	TIMESTAMP literal
TIME	CURRENT_TIME
	CURRENT_TIMESTAMP
	TIME literal
	TIMESTAMP literal
TIMESTAMP	CURRENT_DATE
	CURRENT_TIME
	CURRENT_TIMESTAMP
	DATE literal
	TIME literal

Column Data Type	Supported Default Values
	TIMESTAMP literal

In addition, Vantage also supports DEFAULT phrase specifications such as the following:

- CREATE TABLE tab2 (i INT, j INTERVAL DAY DEFAULT 4);
- CREATE TABLE tab3 (i INT, j INTERVAL HOUR TO MINUTE DEFAULT '11:23' );

For more information about implicit conversions of DateTime data types, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

The default value (excepting keywords) for character columns must be in the repertoire of the character data type defined for the column and session character set.

In an ALTER TABLE statement, the DEFAULT phrase can be used with a keyword to override a previously-defined value.

## Unsupported Data Types

You cannot specify the DEFAULT phrase for columns defined with the following data types:

- BLOB or CLOB
- UDT, with the exception of DEFAULT NULL

## DEFAULT Values and CHECK Constraints

A default value must not violate any CHECK constraints specified for the column. If a default value would violate a CHECK constraint, the conflict is not recognized at the time the table is defined or altered. The conflict is recognized the first time that an INSERT or UPDATE attempts to enter a NULL, which would be replaced by the default.

For character data, constraints are checked using the current session collation. Therefore, it is possible for a default value to meet the constraint for one collation and violate the constraint for another collation.

## DEFAULT Values and Identity Columns

You cannot specify the DEFAULT attribute for Identity columns.

## DEFAULT Values and Built-In Functions

A keyword used as a *constant\_value* inserts a string that is already known to the system, such as the system date, the system time, or the name of the user defining the column.

Use of the Teradata SQL built-in functions DATE and TIME as default values is deprecated. They are non-ANSI standard and continue in the language only to maintain backward compatibility. Use the ANSI-standard CURRENT\_DATE and CURRENT\_TIME instead.

## Maximum Length for a DEFAULT Value

The maximum length of *constant\_value* is 510 characters for character columns, and 510 bytes for byte columns.

## System Values for DEFAULT Phrase

Unlike the Teradata WITH DEFAULT phrase, there are no system default values for the DEFAULT phrase.

## Values for DEFAULT Phrase Keywords

The following table lists the DEFAULT phrase forms, the function performed by each variable or keyword, and the corresponding default data types.

This form ...	Inserts ...	Using this data type ...
DEFAULT <i>constant_value</i>	the value defined as the default for the column	the type defined for the table column.
DEFAULT DATE <i>quotestring</i>	the date value specified by <i>quotestring</i> as the default for the column	DATE.
DEFAULT NULL	a NULL	None.
DEFAULT CURRENT_DATE	the current system date	DATE.
DEFAULT CURRENT_TIME	the current system time	TIME.
DEFAULT CURRENT_TIMESTAMP	the current system date and time	TIMESTAMP.
DEFAULT USER Teradata extension to the ANSI standard	the user name of the current user	CHAR( <i>n</i> ) CHARACTER SET UNICODE VARCHAR( <i>n</i> ) CHARACTER SET UNICODE where <i>n</i> is the length of the longest permissible user name.

If DEFAULT NULL is specified with the NOT NULL phrase in CREATE TABLE or ALTER TABLE ADD statements, no error or warning messages are returned for the statements. However, an error occurs the first time an INSERT or UPDATE attempts to enter a null.

You can assign a literal value, such as a blank, as the DEFAULT for a column as follows:

```
CHAR(1) DEFAULT ' ' ...
```

## Examples

**Example: Numeric**

Suppose the DeptNo column is defined in table Departments as follows.

```
CREATE TABLE Departments
  (DeptName CHARACTER(36)
  ,DeptNo SMALLINT DEFAULT 100 FORMAT '999' BETWEEN 100 AND 900
  ,ManagerID INTEGER);
```

When the value for the DeptNo field is not provided in an INSERT statement, then the value 100 with type SMALLINT is inserted automatically.

**Example: Date Literal**

The following example shows a DATE column specified with a DEFAULT date literal:

```
F4 DATE DEFAULT DATE '2000-01-01'
```

**Example: Current Time**

The following example shows a TIME column specified with a DEFAULT of the current time:

```
Stage TIME(3) DEFAULT CURRENT_TIME(3)
```

**Example: Interval**

The following example shows an INTERVAL column specified with a DEFAULT interval:

```
Scale INTERVAL YEAR(2) DEFAULT INTERVAL -'10' YEAR
```

## WITH DEFAULT Phrase

Specifies that a system-defined default value is to be inserted in the field when a value is not specified for a column in an INSERT statement.

**ANSI Compliance**

WITH DEFAULT is a Teradata extension to the ANSI SQL:2011 standard.

**Syntax**

```
WITH DEFAULT
```



## Usage Notes

### General

The ANSI form of this phrase is documented in [DEFAULT Phrase](#).

If you use a WITH DEFAULT phrase with a numeric range constraint, the range must include the system default value assigned to the data type defined for the column.

When a CREATE TABLE statement is processed, all WITH DEFAULT phrases are converted to DEFAULT phrases in which the system default value becomes the default *constant\_value*.

### Incompatibility Note

The WITH DEFAULT phrase is mutually exclusive with the DEFAULT phrase.

### System Values for WITH DEFAULT Phrase

The value of a system default is determined by the data type defined for the column. The data types and associated system default values appear in the following table.

Data Type	System Default
BIGINT	Zero
BYTE[(n)]	Zero if <i>n</i> omitted, or <i>n</i> binary zeros
BYTEINT	Zero
CHAR[(n)]	Depends on the server character set as follows: <ul style="list-style-type: none"> <li>• If LATIN, the system default is ASCII SPACE (0x20).</li> <li>• If UNICODE, the system default is SPACE (U+0020).</li> <li>• If KANJISJIS or KANJI1, the system default is ASCII SPACE (0x20)</li> <li>• If GRAPHIC, the system default is IDEOGRAPHIC SPACE (U+3000)</li> </ul>
DATE	Current date
DECIMAL NUMERIC	Zero
DOUBLE PRECISION	Zero
FLOAT	Zero
INTEGER	Zero
INTERVAL DAY	INTERVAL '0' DAY
INTERVAL DAY TO HOUR	INTERVAL '0 00' DAY TO HOUR
INTERVAL DAY TO MINUTE	INTERVAL '0 00:00' DAY TO MINUTE
INTERVAL DAY TO SECOND	INTERVAL '0 00:00:00' DAY TO SECOND

Data Type	System Default
INTERVAL HOUR	INTERVAL '0' HOUR
INTERVAL HOUR TO MINUTE	INTERVAL '00:00' HOUR TO MINUTE
INTERVAL HOUR TO SECOND	INTERVAL '00:00:00' HOUR TO SECOND
INTERVAL MINUTE	INTERVAL '0' MINUTE
INTERVAL MINUTE TO SECOND	INTERVAL '00:00' MINUTE TO SECOND
INTERVAL MONTH	INTERVAL '0' MONTH
INTERVAL SECOND	INTERVAL '0' SECOND
INTERVAL YEAR	INTERVAL '0' YEAR
INTERVAL YEAR TO MONTH	INTERVAL '0-00' YEAR TO MONTH
NUMBER	Zero
PERIOD(DATE)	The default value is set implicitly using a Period value constructor with the beginning argument set to CURRENT_DATE and the ending argument set to UNTIL_CHANGED.
PERIOD(TIMESTAMP)	The default value is set implicitly using a Period value constructor with the beginning argument set to CURRENT_TIMESTAMP[(n)] and the ending argument set to UNTIL_CHANGED.
REAL	Zero
SMALLINT	Zero
TIME	CURRENT_TIME
TIMESTAMP	CURRENT_TIMESTAMP
TIMESTAMP WITH TIME ZONE	CURRENT_TIMESTAMP
TIME WITH TIME ZONE	CURRENT_TIME
VARBYTE (n)	' ' (null string)

## Unsupported Data Types

You cannot specify the WITH DEFAULT phrase for columns defined with the following data types:

- BLOB or CLOB
- UDT
- PERIOD(TIME)

## Examples

**Example: WITH DEFAULT Phrase**

If the EdLev column is defined as:

```
EdLev BYTEINT FORMAT 'Z9' NOT NULL WITH DEFAULT,
```

then the column definition for EdLev is stored as:

```
EdLev BYTEINT NOT NULL DEFAULT 0 FORMAT 'Z9',
```

**Example: Inserting a Row Using a Default Value**

The following INSERT statement adds a row containing a zero in the EdLev field defined in the previous example:

```
INSERT INTO Employee
(Name, EmpNo, DeptNo, DOB, Sex, EdLev)
VALUES ('Newhire A', 10025, 700, '49/10/17', 'M',) ;
```

**Example: TIME Column With a Default of the Current Time**

The following example shows a TIME column specified with a default of the current time:

```
Stage TIME(3) WITH DEFAULT
```

**Example: INTERVAL Column With a Default Interval**

The following example shows an INTERVAL column specified with a default interval:

```
Scale INTERVAL HOUR(2) WITH DEFAULT
```

# 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"> <li>If <i>delimiter</i> is semicolon: x; x; x</li> <li>If <i>delimiter</i> is { ,   OR }, you can do either of the following: <ul style="list-style-type: none"> <li>x, x, x</li> <li>x OR x OR x</li> </ul> </li> </ul> See note after table.

**Note:**

You can repeat only the immediately preceding item. For example, if the syntax is:

```
KEYWORD x [...]
```

You can repeat x. Do not repeat KEYWORD.

If there is no white space between x and the delimiter, the repeatable item is x and the delimiter. For example, if the syntax is:

```
[ x, [...] ] y
```

- You can omit x: y
- You can specify x once: x, y
- You can repeat x and the delimiter: x, x, x, y

## 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

# External Representations for UDTs

This section provides external representations for UDTs.

## Transforms Off and Period Data Type Support

The configuration response parcel (in the `PclCfgMultipartFeatType` struct (`pclmisc.h`)) includes byte flags that enable the server to inform the client of transforms off and Period data type support.

### UDTTransformsOff Byte Flag Values

Value	Description
0	Transforms are turned on. UDTs are transferred in transformed mode. Period metadata and data are transferred in non-struct mode (binary form).
1	UDTs are transferred in untransformed mode. Period metadata and data are transferred in non-struct mode (binary form). The client can set the <code>UDTTransformsOff</code> flag within the options portion of the DBCAREA. See <a href="#">UDTTransformsOff Flag</a> for more information on this flag.
2	UDTs are transferred in untransformed mode. Period metadata and data are transferred in struct mode. When the value is set to 2: <ul style="list-style-type: none"> <li>The client can set the <code>PeriodStructOn</code> flag within the options portion of the DBCAREA. See <a href="#">PeriodStructOn Flag</a> for more information on this flag.</li> <li>The client is permitted (but not required) to send PERIOD struct metadata and data values to Vantage in request messages.</li> </ul>

See *Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems*, B035-2417 and *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418 for more information on setting DBCAREA options.

### ArrayDataType Byte Flag Values

Value	Description
0	ARRAY data types are not supported.
1	ARRAY data types are supported. When the value is set to 1, the client can set the <code>ArrayTransformsOff</code> flag within the options portion of the DBCAREA. See <a href="#">ArrayTransformsOff Flag</a> for more information on this flag. See <i>Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems</i> , B035-2417 and <i>Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems</i> , B035-2418 for more information on setting DBCAREA options.

## UDTTransformsOff Flag

The UDTTransformsOff flag in the Options parcel indicates if UDTs are returned from the server in transformed or untransformed mode.

The UDTTransformsOff flag is stored in `pcloptlist_t` (`pc1toteq.h`).

UDTs must have transforms defined on them to be completely functional. Transforms are necessary for Teradata utilities to operate correctly.

### Values

Value	Description
Y	UDTs are returned from the server in untransformed mode.
N	Default. UDTs are returned from the server in transformed mode. If the flag is a binary zero, it is treated as if the flag was set to N. If no Options parcel is sent with an application request, the server assumes the flag was set to N.

See [Transforms Off and Period Data Type Support](#) for more information on setting these values.

## PeriodStructOn Flag

The PeriodStructOn flag in the Options parcel allows the client to request PERIOD struct metadata, null indicator bits, and data values from the database in response messages.

The PeriodStructOn flag is stored in `pcloptlist_t` (`pcltoteq.h`).

The client sending PERIOD struct values to Vantage in request messages is independent from the client setting the Option parcel flag to request that Vantage return PERIOD struct values in response messages.

### Values

Value	Description
Y	Default. Request PERIOD struct metadata, null indicator bits, and data values from Vantage. The Period data type value is exported as its date/time/timestamp attributes. Vantage accepts (but does not require) PERIOD struct metadata, null indicator bits, and PERIOD struct data values from the client.
N	Do not request PERIOD struct metadata, null indicator bits, and data values from Vantage.

See [Transforms Off and Period Data Type Support](#) for more information on setting these values.

When either the BEGIN or END PERIOD attributes is null, the PERIOD value is null. Therefore, null PERIOD values sent from Vantage to the client have all three null indicator bits set to indicate that the PERIOD value



is null, and that the BEGIN and END attributes are also null. If the client sends null BEGIN or END PERIOD attributes, Vantage considers the PERIOD value to be null.

## ArrayTransformsOff Flag

The ArrayTransformsOff flag in the Options parcel indicates if the ARRAY data type is sent in transformed or untransformed mode from the server.

The ArrayTransformsOff flag is stored in `pcloptlist_t` (`pcltoteq.h`).

### Values

Value	Description
Y	The ARRAY data type is sent in untransformed mode from the server. The ReturnStatementInfo flag must be set to Y before setting the ArrayTransformsOff flag to Y. Otherwise, SQL Engine returns an error.
N	Default. The ARRAY data type is sent in transformed mode from the server. If the flag is a binary zero, it is treated as if the flag was set to N. If no Options parcel is sent over with an application request, or the new field is not present in the Options parcel, the server assumes the flag was set to N.

See [Transforms Off and Period Data Type Support](#) for more information on setting these values.

## Flag Setting Combinations

Set the UDTTransformsOff flag to Y to turn on the PeriodStructOn flag.

SQL Engine returns an error if the UDTTransformsOff flag is set to N and the PeriodStructOn flag is set to Y.

The ArrayTransformsOff flag can be set independent of the UDTTransformsOff and PeriodStructOn flags.

The following table provides information on flag setting combinations.

UDTTransformsOff	PeriodStructOn	ArrayTransformsOff	Description	Applicable to
N	N	N	<ul style="list-style-type: none"> <li>Binary representation for primitive/native data types (for example, Decimal, Integer).</li> <li>UDTs are transformed.</li> <li>Binary representation for Period data types.</li> </ul>	CLI. Default mode.

UDTTransformsOff	PeriodStructOn	ArrayTransformsOff	Description	Applicable to
			<ul style="list-style-type: none"> <li>Arrays are transformed to VARCHAR and the wire-protocol representations in the above bullets are no longer applied to the array elements.</li> </ul>	
N	N	Y	<ul style="list-style-type: none"> <li>Binary representation for primitive/native data types (for example, Decimal, Integer).</li> <li>UDTs are transformed.</li> <li>Binary representation for Period data types.</li> <li>Arrays are untransformed and the wire-protocol representations in the above bullets are applied to the array elements.</li> </ul>	.NET, ODBC, and third-party or in-house CLI applications.
N	Y	N or Y	SQL Engine returns an error.	
Y	N	Y	<ul style="list-style-type: none"> <li>Binary representation for primitive/native data types (for example, Decimal, Integer).</li> <li>UDTs are untransformed.</li> <li>Binary representation for Period data types.</li> <li>Arrays are untransformed and the wire-protocol representations in the above bullets</li> </ul>	.NET Data Provider uses this combination to add provider-specific support for UDT and UDT to .NET-Object mappings. The application chooses UDT-Transform On and Off.

UDTTransformsOff	PeriodStructOn	ArrayTransformsOff	Description	Applicable to
			are applied to the array elements.	
Y	N	N	<ul style="list-style-type: none"> <li>• Binary representation for primitive/native data types (for example, Decimal, Integer).</li> <li>• UDTs are untransformed.</li> <li>• Binary representation for Period data types.</li> <li>• Arrays are transformed to VARCHAR and the wire-protocol representations in the above bullets are no longer applied to the array elements.</li> </ul>	Third-party CLI-based applications. Some existing applications support pass-through SQL. That is, these applications are not limited to a specific physical database model.
Y	Y	Y	<ul style="list-style-type: none"> <li>• Binary representation for primitive/native data types (for example, Decimal, Integer).</li> <li>• UDTs are untransformed.</li> <li>• Period data types are transmitted as structure with BEGIN and END period (for example, Date, Time).</li> <li>• Arrays are untransformed and the wire-protocol representations in the above bullets are applied to the array elements.</li> </ul>	JDBC.
Y	Y	N	<ul style="list-style-type: none"> <li>• Binary representation for primitive/native</li> </ul>	Third-party CLI based applications.

UDTTransformsOff	PeriodStructOn	ArrayTransformsOff	Description	Applicable to
			<p>data types (for example, Decimal, Integer).</p> <ul style="list-style-type: none"> <li>UDTs are untransformed.</li> <li>Period data types are transmitted as structure with BEGIN and END period (for example, Date, Time).</li> <li>Arrays are transformed to VARCHAR and the wire-protocol representations in the above bullets are no longer applied to the array elements.</li> </ul>	Some existing applications support pass-through SQL. That is, these applications are not limited to a specific physical database model.

### ARRAY Data Values Format with Transforms Off

The one-dimensional and multidimensional ARRAY data values with the ArrayTransformsOff flag set to Y can appear in various data parcels, including:

- IndicData (parcel flavor number 68)
- MultipartIndicData parcel (flavor 142)
- Record (in Indicator mode) (parcel flavor number 10)
- MultipartRecord (parcel flavor number 144)

One null indicator bit for the ARRAY data type is inside the null indicator bit array.

The following table describes wire-protocol representations of the array value.

Item	Description
2-byte (16-bit SMALLINT) length field	The length excluding itself in bytes.
4-byte (32-bit unsigned integer)	Current Cardinality, which indicates the total number of array elements returned.
Elements-Null-Indicator Bits	<p>A variable-length byte array, where:</p> $\text{length} = (\text{Current-Cardinality} * (\text{number-bits-per-element} + 7) / 8)$ <p>If the array element is a UDT and the UDTTransformsOff flag is set to Y, the Elements-Null-Indicator Bits for this element is expanded to 1 null indicator bit for the UDT</p>

Item	Description
	<p>followed by 1 null indicator bit for each attribute of the UDT from left to right in depth-first order.</p> <p>If the array element is a Period data type and the PeriodStructOn flag is set to Y, the Elements-Null-Indicator Bits for this element is expanded to 1 null indicator bit for the Period data type followed by 1 null indicator bit for BEGIN and another null indicator bit for END.</p>
Array Elements Data	<p>Actual data values for each element in row-major order.</p> <p>If the array element is a UDT and the UDTransformsOff flag is set to Y, each element value is expanded to data values of each attribute of the UDT from left to right in depth-first order.</p> <p>If the array element is a Period data type and the PeriodStructOn flag is set to Y, each element value is expanded to BEGIN value followed by END value.</p>

For a null array with the ArrayTransformsOff flag set to Y, the null indicator bit for the array is set and the 2-byte length field is set to zero. No other value are sent.

## Examples: Flag Setting Combinations

Following are examples of ARRAY data type values when the ArrayTransformsOff flag is set to Y. These examples focus on the combinations most commonly used by ODBC/.NET and JDBC

- UDTransformsOff set to Y, PeriodStructOn set to Y, and ArrayTransformsOff set to Y (YYY in the examples)
- UDTransformsOff set to N, PeriodStructOn set to N, and ArrayTransformsOff set to Y (NNY in the examples)

### Example 1: One-Dimensional Array With One Null Element

```
/*Oracle-compatible and TD syntax respectively: */
Create Type intarray as VARRAY(3) OF Int;
Create Type intarray as Int Array[3];

Create Table tab1 (id int, phonenum intarray);
Ins into tab1(10, NEW intarray(111222333, NULL, 123456789));
Sel phonenum from tab1;
```

For both YYY and NNY, the parcel body fields of the data parcels for the select query contain:

1. The null indicator bit array is  $(n+7)/8 = (1+7)/8 = 1$  byte and appears as "00000000".
2. The length field is a 2-byte SMALLINT of 17.
3. The Current Cardinality field is a 4-byte unsigned integer of 3.
4. The Elements-Null-Indicator Bits is  $(3+7)/8 = 1$  byte and appears as "01000000", because the second element is NULL.

5. The Array Elements Data is the binary representation of integer 111222333, followed by the dummy null value of integer zero, followed by integer 123456789.

### Example 2: One-Dimensional Array With UDT Element Type

This example is a one-dimensional array with a UDT element type with a null attribute.

```
CREATE TYPE employee AS (name VARCHAR(10),
                        employee_id INTEGER)
...;
/*Oracle-compatible and TD syntax respectively: */
CREATE TYPE emparray AS VARRAY(20) OF employee;
CREATE TYPE emparray AS employee ARRAY[20];
Create Table tab2 (dept_no int, dept_emps emparray);
Ins into tab2(10, NEW emparray (NEW employee('Mike', NULL), NEW
employee('Mark', 101)));
Sel dept_emps from tab2;
```

For YYY, the parcel body fields of the data parcels for the select query contain:

1. The null indicator bit array is  $(n+7)/8 = (1+7)/8 = 1$  byte and appears as "00000000".
2. The length field is a 2-byte SMALLINT of 25.
3. The Current Cardinality field is a 4-byte unsigned integer of 2.
4. The Elements-Null-Indicator Bits is  $(3+3+7)/8 = 1$  byte and appears as "00100000", because the second attribute of the first element is NULL.
5. The Array Elements Data is the binary representation of string 'Mike', followed by the dummy NULL value of integer zero, followed by string 'Mark' which is followed by integer 101.

For NNY, assuming NEW employee ('Mike', NULL) is transformed to 'Mike-NULL' and NEW employee ('Mark', 101) is transformed to 'Mark-101', the parcel body fields of the data parcels for the select query contain:

1. The null indicator bit array is  $(n+7)/8 = (1+7)/8 = 1$  byte and appears as "00000000".
2. The length field is a 2-byte SMALLINT of 26.
3. The Current Cardinality field is a 4-byte unsigned integer of 2.
4. The Elements-Null-Indicator Bits is  $(2+7)/8 = 1$  byte and appears as "00000000".
5. The Array Elements Data is the binary representation of string 'Mike-NULL', followed by string 'Mark-101'.

### Example 3: Multidimensional Array With One Null Element

```
/*Oracle-compatible and TD syntax respectively: */
Create Type TwoD_intarray as VARRAY(2)(2) OF Integer;
Create Type TwoD_intarray as Integer Array[2][2];

Create Table tab3 (id int, simulation TwoD_intarray);
```

```
Ins into tab3(10, NEW TwoD_intarray(111222333, 123456789, NULL, 777777777));
Sel simulation from tab3;
```

For both YYY and NNY, the parcel body fields of the data parcels for the above query contain:

1. The null indicator bit array is  $(n+7)/8 = (1+7)/8 = 1$  byte and appears as "00000000".
2. The length field is a 2-byte SMALLINT of 21.
3. The Current Cardinality field is a 4-byte unsigned integer of 4.
4. The Elements-Null-Indicator Bits is  $(4+7)/8 = 1$  byte and appears as "00100000".
5. The Array Elements Data is the binary representation of integer 111222333, followed by integer 123456789, followed by the dummy null value of integer zero which is followed by integer 777777777.

## USING Clause

UDTs with transforms off does not support the USING clause, and is available only with parameter markers.

## Transform Input/Output Strings for ARRAY/VARRAY UDTs

The transform group functionality for one-dimensional and multidimensional ARRAY/VARRAY UDTs is generated automatically by Vantage.

Vantage uses the same string format for one-dimensional and multidimensional ARRAY/VARRAY UDTs for:

- The tosql transform input parameter
- The fromsql transform output value

The format of the transformed output in a VARCHAR string is a string of each array element value, referred to as the transformed value string, separated by a comma and delimited by parentheses as indicated below. Assuming the array has  $n$  elements:

```
(element_1, ..., element_n)
```

Call this string the transformed value string. The format for each element of the array according to its SQL data type is listed in the following table.

Data Type	Format	Size (bytes)
CHARACTER( $n$ ) CHARACTER SET server_character_set	'string' While the string itself is a character string of length $k$ , the value for $k$ is the length of the characters and is encoded in the declared server character set.	$k + 2$
VARCHAR( $n$ CHARACTER SET server_character_set		
<ul style="list-style-type: none"> <li>• NUMERIC(<math>n,m</math>)</li> <li>• DECIMAL(<math>n,m</math>)</li> <li>• NUMBER(<math>n</math>) (exact type) For this type, <math>m = 0</math>.</li> </ul>	The array tosql should have the following syntax. <ul style="list-style-type: none"> <li>• <math>\pm n</math></li> <li>• <math>\pm .n</math></li> <li>• <math>\pm n.n</math></li> </ul>	$\leq n+2$

Data Type	Format	Size (bytes)
	<p>where:</p> <ul style="list-style-type: none"> <li>• <math>\pm</math> is an optional sign. The default is +.</li> <li>• <math>n</math> is any valid integer number.</li> </ul> <p>The array transformed fromsql has the following format.</p> <pre>--(I).9(F)</pre> <p>where:</p> <ul style="list-style-type: none"> <li>• <math>I = n-m</math></li> <li>• <math>F = m</math></li> </ul> <p>When <math>m = 0</math> for <code>NUMBER(<math>n</math>)</code> and <code>NUMBER(<math>n</math>, <math>m</math>)</code>, the array transformed fromsql must have the following format, indicating that the decimal point is not shown.</p> <pre>--(I)</pre>	
BYTEINT	$\pm n$ where: <ul style="list-style-type: none"> <li>• <math>\pm</math> is an optional sign. The default is +.</li> <li>• <math>n</math> is any valid integer number between -128 and 127.</li> </ul>	$\leq 4$
SMALLINT	$\pm n$ where: <ul style="list-style-type: none"> <li>• <math>\pm</math> is an optional sign. The default is +.</li> <li>• <math>n</math> is any valid integer number between -32768 and 32767.</li> </ul>	$\leq 6$
INTEGER	$\pm n$ where: <ul style="list-style-type: none"> <li>• <math>\pm</math> is an optional sign. The default is +.</li> <li>• <math>n</math> is any valid integer number between -2147483648 and 2147483647.</li> </ul>	$\leq 11$
BIGINT	$\pm n$ where: <ul style="list-style-type: none"> <li>• <math>\pm</math> is an optional sign. The default is +.</li> <li>• <math>n</math> is any valid integer number between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807.</li> </ul>	$\leq 20$



Data Type	Format	Size (bytes)
<ul style="list-style-type: none"> <li>• REAL</li> <li>• FLOAT</li> <li>• DOUBLE PRECISION</li> </ul>	<p>The array transformed tosql must have one of the following formats.</p> <ul style="list-style-type: none"> <li>• <math>\pm nE\pm m</math></li> <li>• <math>\pm n.E\pm m</math></li> <li>• <math>\pm .nE\pm m</math></li> <li>• <math>\pm n.nE\pm m</math></li> </ul> <p>where:</p> <ul style="list-style-type: none"> <li>• <math>\pm</math> is an optional sign. The default is +.</li> <li>• <math>n</math> is any valid integer number representing the whole and, optionally, fractional component of the mantissa. For the FLOAT, REAL, and DOUBLE PRECISION types the number of digits cannot exceed 15, <i>excluding</i> leading zeros in the whole component of the mantissa. For the NUMBER, NUMBER(*), and NUMBER(*, <math>m</math>) types, the number of digits cannot exceed 40, <i>excluding</i> leading zeros in the whole component of the mantissa.</li> <li>• <math>E</math> is a symbol indicating that the digits that follow in <math>m</math> are the exponent of the number.</li> <li>• <math>m</math> is any valid integer number representing the exponent for the number. The number of digits that <math>m</math> contains cannot exceed 3, including leading zeros.</li> </ul> <p>For the REAL, FLOAT, and DOUBLE PRECISION types the transformed fromsql has the following format.</p> <pre>-9.999999999999999E-999</pre> <p>For the NUMBER, NUMBER(*), and NUMBER(*, <math>m</math>) types the transformed fromsql has the following format.</p> <pre>FN9</pre> <p>The the definition of the FN9 format is as follows.</p> <ul style="list-style-type: none"> <li>• If the data fits in 64 characters, then display it as is in character format, discarding any leading zeros.</li> <li>• If the data does not fit in 64 characters, use the exponent notation to display the data. The format for exponent notation is FNE.</li> </ul>	<p><math>\leq 22</math> for these types.</p> <ul style="list-style-type: none"> <li>• REAL</li> <li>• FLOAT</li> <li>• DOUBLE PRECISION</li> </ul> <p><math>\leq 40</math> for these types.</p> <ul style="list-style-type: none"> <li>• NUMBER</li> <li>• NUMBER(*)</li> <li>• NUMBER(*, <math>m</math>)</li> </ul>
BYTE( $n$ )	<p>X(<math>2n</math>)</p> <p>This is a string of hexadecimal digits of length <math>2n</math>.</p>	$2n$
VARBYTE( $n$ )	X( $2k$ )	$2k$

Data Type	Format	Size (bytes)
	This is a string of hexadecimal digits of length $2k$ , where $k$ is the number of bytes in the string.	
DATE	YYYY-MM-DD	10
TIME( $n$ )	HH:MI:SS.S( $F$ )	<ul style="list-style-type: none"> <li>If <math>n=0</math>, size = 8</li> <li>If <math>n&gt;0</math>, size = <math>9+n</math></li> </ul>
TIME( $n$ ) WITH TIME ZONE	HH:MI:SS.S( $F$ )Z	<ul style="list-style-type: none"> <li>If <math>n=0</math>, size = 14</li> <li>If <math>n&gt;0</math>, size = <math>15+n</math></li> </ul>
TIMESTAMP( $n$ )	YYYY-MM-DDBHH:MI:SS.S( $F$ )	If $n=0$ , size = 19
		If $n>0$ , size = $20+n$
TIMESTAMP( $n$ ) WITH TIME ZONE	YYYY-MM-DDBHH:MI:SS.S( $F$ )Z	If $n=0$ , size = 25
		If $n>0$ , size = $26+n$
INTERVAL SECOND( $n, m$ )	$-s(n)$ if no fractional precision is defined	If no fractional precision defined, size = $n + 1$
	$-s(n).s(m)$ if a fractional precision is defined as $m$	If a fractional precision is defined, size = $n + m + 2$
INTERVAL MINUTE( $n$ )	$-m(n)$	$n + 1$
INTERVAL MINUTE( $n$ ) TO SECOND( $m$ )	$-m(n):ss$ if no fractional precision is defined.	If no fractional precision defined, size = $n + 4$
	$-m(n):ss.s(m)$ if a fractional precision is defined as $m$ .	If a fractional precision is defined as $m$ , size = $n + m + 5$
INTERVAL HOUR( $n$ )	$-h(n)$	$n + 1$
INTERVAL HOUR( $n$ ) TO MINUTE	$-h(n):mm$	$n + 4$
INTERVAL HOUR( $n$ ) TO SECOND( $m$ )	$-h(n):mm:ss$ if no fractional precision is defined.	If no fractional precision defined, size = $n + 7$
	$-h(n):mm:ss.s(m)$ if a fractional precision is defined as $m$ .	If a fractional precision is defined

Data Type	Format	Size (bytes)
		as $m$ , size = $n + m + 8$
INTERVAL DAY( $n$ )	$-d(n)$	$n + 1$
INTERVAL DAY( $n$ ) TO SECOND( $m$ )	$-d(n)$ hh:mm:ss if no fractional precision is defined.	If no fractional precision defined, size = $n + 10$
	$-d(n)$ hh:mm:ss.s( $m$ ) if a fractional precision is defined as $m$ .	If a fractional precision is defined as $m$ , size = $n+m+11$
INTERVAL DAY( $n$ ) TO MINUTE	$-d(n)$ hh:mm	$n + 7$
INTERVAL DAY( $n$ ) TO HOUR	$-d(n)$ hh	$n + 4$
INTERVAL MONTH( $n$ )	$-m(n)$	$n + 1$
INTERVAL YEAR( $n$ )	$-y(n)$	$n + 1$
INTERVAL YEAR( $n$ ) TO MONTH	$-y(n)$ -mm	$n + 4$
UDT (distinct form)	Same as the underlying predefined data type.	
UDT (structured form)	<p>Assume that the structured UDT has <math>k</math> attributes. Its format is as follows.</p> <pre>(attribute1, attribute2, ..., attributek)</pre> <p>where the attributes are separated by a comma character, are in their defined order, and are delimited by parentheses.</p> <p>If an array element is a nested structured UDT, it is returned in row order, depth first. A nested attribute must be delimited by parenthesis characters.</p> <p>For example, assume that attribute2 is a UDT with two attributes, attribute2_1 and attribute2_2. Its format is as follows.</p> <pre>(attribute1,(attribute2_1, attribute2_2), ... , attributek)</pre>	
PERIOD(DATE)	(YYYY-MM-DD, YYYY-MM-DD)	24

Data Type	Format	Size (bytes)
PERIOD(TIME(n))	(HH:MI:SS.S(F), HH:MI:SS.S(F))	<ul style="list-style-type: none"> <li>If <math>n=0</math>, size=20</li> <li>If <math>n&gt;0</math>, size=<math>2n+22</math></li> </ul>
PERIOD(TIME(n) WITH TIME ZONE)	(HH:MI:SS.S(F)Z, HH:MI:SS.S(F)Z)	<ul style="list-style-type: none"> <li>If <math>n=0</math>, size=32</li> <li>If <math>n&gt;0</math>, size=<math>2n+34</math></li> </ul>
PERIOD(TIMESTAMP(n))	(YYYY-MM-DDBHH:MI:SS.S(F), YYYY-MM-DDBHH:MI:SS.S(F))	<ul style="list-style-type: none"> <li>If <math>n=0</math>, size=42</li> <li>If <math>n&gt;0</math>, size=<math>2n+44</math></li> </ul>
PERIOD(TIMESTAMP(n) WITH TIME ZONE)	(YYYY-MM-DDBHH:MI:SS.S(F)Z, YYYY-MM-DDBHH:MI:SS.S(F)Z)	<ul style="list-style-type: none"> <li>If <math>n=0</math>, size=54</li> <li>If <math>n&gt;0</math>, size=<math>2n+56</math></li> </ul>

Note the following about the information in the preceding table:

- There is no support for transforms for the following data elements for either one-dimensional or multidimensional ARRAY/VARRAY data types. You cannot use these element types to create ARRAY/VARRAY data types:

- BLOB
- CLOB
- Geospatial

- If the element type is a primitive data type other than CHARACTER or VARCHAR, or is a PERIOD or UDT with no CHARACTER or VARCHAR attributes, the definition of the array constructor transformed value string is VARCHAR(64000) CHARACTER SET LATIN.

If the size of the transformed value string is larger than 64K when the ARRAY/VARRAY type is being created, the request aborts and SQL Engine returns an error to the requestor.

- If the element type is CHARACTER or VARCHAR CHARACTER SET LATIN, or a UDT with CHARACTER or VARCHAR attributes whose server character set is exclusively Teradata Latin, the definition of the array transformed value string is VARCHAR(64000) CHARACTER SET LATIN.

If the element type is CHARACTER or VARCHAR whose server character set is not Teradata LATIN, or if it is a UDT with any CHARACTER or VARCHAR attribute is not Teradata LATIN, the definition of the array transformed value string is VARCHAR(32000) CHARACTER SET UNICODE.

If the maximum size of the transformed value string is larger than the defined length when the ARRAY/VARRAY type is being created, the request aborts and SQL Engine returns an error to the requestor.

- If the element type is CHARACTER or VARCHAR or a UDT with CHARACTER or VARCHAR attributes, and its array elements contain many embedded apostrophe characters, you must specify an extra quote to distinguish the embedded apostrophe characters.

This limits the total size of the transform string that can be output when selecting the array column because SQL Engine counts the embedded quote characters against the size of the transform string.

If your array elements contain any embedded apostrophe characters, SQL Engine outputs those characters when the `ArrayTransformsOff` flag is set to `N`, using the `fromsql` transform.

In the worst case, which is a string with all apostrophe characters embedded, this reduces the size of the maximum transform string by half. Therefore, if the total size of the transform string that could be generated for an `ARRAY/VARRAY` type exceeds the maximum row size, SQL Engine aborts the `CREATE TYPE` request for that `ARRAY/VARRAY` type and returns an error to the requestor.

- Vantage does not display uninitialized elements.
- If an element value is initialized, all of the elements preceding it must have also been initialized either to a value or to null.
- You must indicate a null element using a null literal, which could be `'NULL'`, `'Null'`, `'null'`. The case is insensitive.

If the defined server character set is something other than Teradata LATIN, the null element is indicated by the corresponding encoding of a `NULL` literal in the defined server character set. This also applies for a structured UDT null and for a structured UDT with a null attribute.

- Overflow avoidance:

The size of an array transformed value string needs to be within the size limit of a `VARCHAR`, which is 64K for the Teradata LATIN server character set. The limit is different if the server character set is something other than Teradata LATIN. See [VARCHAR Data Type](#) for specific details of the size limit of `VARCHAR` data with a server character set other than Teradata LATIN.

If the size of an array transformed value string is greater than the size limit for a `VARCHAR` type when the `ARRAY/VARRAY` type is being created, the request aborts and SQL Engine returns an error to the requestor.

- SQL Engine ignores any space, new line, or tab character, either before or after the comma character, or before the last apostrophe character or after the first apostrophe character.

## Additional Information

### Teradata Links

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