

# Teradata Vantage™ SQL Data Manipulation Language

---

Release 16.20




March 2019

# Copyright and Trademarks

Copyright © 2000 - 2019 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
 <b>NOTICE</b>	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.
 <b>CAUTION</b>	Indicates a hazardous situation which, if not avoided, could result in minor or moderate personal injury.
 <b>WARNING</b>	Indicates a hazardous situation which, if not avoided, could result in death or serious personal injury.

## 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.

## 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: [teradata-books@lists.teradata.com](mailto:teradata-books@lists.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 Manipulation Language</b>	<b>6</b>
Overview	6
Changes and Additions	6
<b>Chapter 2: Select Statements</b>	<b>7</b>
Overview	7
SELECT	7
SELECT AND CONSUME	41
SELECT ... INTO	45
Request Modifier	45
WITH Modifier	45
WITH DELETED ROWS	73
AS JSON	74
Distinct	75
ALL	79
.ALL Operator	79
NORMALIZE	82
TOP <i>n</i>	86
FROM Clause	90
Table Function	96
Table Operator	103
Derived Tables	128
WHERE Clause	132
Specifying Subqueries in Search Conditions	144
Correlated Subqueries	149
Scalar Subqueries	158
GROUP BY Clause	161
CUBE Grouping Set Option	171
GROUPING SETS Option	174
ROLLUP Grouping Set Option	176
HAVING Clause	180
QUALIFY Clause	185
SAMPLE Clause	191
SAMPLEID Expression	200
EXPAND ON Clause	202
ORDER BY Clause	239
WITH Clause	256
<b>Chapter 3: Set Operators</b>	<b>263</b>

Overview . . . . .	263
Rules for Set Operators . . . . .	265
Precedence of Set Operators . . . . .	267
Retaining Duplicate Rows Using the ALL Option . . . . .	268
Attributes of a Set Result . . . . .	269
Set Operators With Derived Tables . . . . .	271
Set Operators in Subqueries . . . . .	273
Set Operators in INSERT ... SELECT Statements . . . . .	275
Set Operators in View Definitions . . . . .	276
Queries Connected by Set Operators . . . . .	278
INTERSECT Operator . . . . .	282
MINUS/EXCEPT Operator . . . . .	284
UNION Operator . . . . .	286
<b>Chapter 4: Join Expressions . . . . .</b>	<b>297</b>
Overview . . . . .	297
Joins . . . . .	297
Inner Joins . . . . .	299
Ordinary Inner Join . . . . .	299
Cross Join . . . . .	302
Self-Join . . . . .	302
Outer Joins . . . . .	303
Definition of the Outer Join . . . . .	304
Outer Join Relational Algebra . . . . .	312
Left Outer Join . . . . .	313
Right Outer Join . . . . .	315
Full Outer Join . . . . .	316
Multitable Joins . . . . .	318
Coding ON Clauses for Outer Joins . . . . .	321
Coding ON Clauses With WHERE Clauses for Outer Joins . . . . .	323
Outer Join Case Study . . . . .	327
Case Study Examples . . . . .	327
Heuristics for Determining a Reasonable Answer Set . . . . .	328
Guidelines for Using Outer Joins . . . . .	334
<b>Chapter 5: Statement Syntax . . . . .</b>	<b>335</b>
Overview . . . . .	335
Statement Independence Support . . . . .	335
Null . . . . .	336
ABORT . . . . .	337
BEGIN TRANSACTION . . . . .	344
CALL . . . . .	349
CHECKPOINT . . . . .	372
COMMENT (Comment-Retrieving Form) . . . . .	374
COMMIT . . . . .	378

DELETE .....	381
ECHO .....	403
END TRANSACTION .....	405
EXECUTE (Macro Form) .....	406
INSERT/INSERT ... SELECT .....	409
LOCKING Request Modifier .....	462
MERGE .....	476
ROLLBACK .....	540
UPDATE .....	546
UPDATE (Upsert Form) .....	575
USING Request Modifier .....	590
<b>Chapter 6: Query and Workload Analysis Statements .....</b>	<b>615</b>
Overview .....	615
COLLECT DEMOGRAPHICS .....	615
COLLECT STATISTICS (QCD Form) .....	618
DROP STATISTICS (QCD Form) .....	625
DUMP EXPLAIN .....	628
EXPLAIN Request Modifier .....	632
INITIATE INDEX ANALYSIS .....	641
INITIATE PARTITION ANALYSIS .....	653
INSERT EXPLAIN .....	657
RESTART INDEX ANALYSIS .....	670
<b>Appendix A: Notation Conventions .....</b>	<b>674</b>
<b>Appendix B: Performance Considerations .....</b>	<b>681</b>
<b>Appendix C: Additional Information .....</b>	<b>688</b>

# Introduction to SQL Data Manipulation Language

## Overview

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

This document describes the Teradata SQL Data Manipulation Language (DML) statements used to retrieve and manipulate data in a Teradata database. Teradata SQL is an ANSI compliant product. Teradata has its own extensions to the language.

## Changes and Additions

Date	Description
March 2019	<p><b>JSON Auto Composition and Shredding</b>  Updated <a href="#">SELECT</a> syntax to add AS JSON option. Added <a href="#">AS JSON</a>, <a href="#">Rules for Using SELECT AS JSON</a>, and <a href="#">Examples: SELECT AS JSON</a>.  Updated <a href="#">INSERT/INSERT ... SELECT</a> syntax to add JSON option. Added <a href="#">JSON</a> and <a href="#">Rules for Using the JSON Option</a>.</p> <p><b>Function Mapping Variable Substitution</b>  Updated <a href="#">Table Operator</a>.</p> <p><b>Function Mapping for Native Functions</b>  Updated <a href="#">Table Operator</a>.</p>
April 2018	<p><b>Function Mapping</b>  Updated <a href="#">SELECT</a> Table Operator syntax to add the function mapping option.  Updated <a href="#">ON Clause</a> to add ON <i>mapping_name</i>.  Added <a href="#">Output Table Clause</a>.  Added <a href="#">Executing a Function Mapping</a>, <a href="#">Function Mapping and the ON Clause</a>, and <a href="#">Function Mapping and the USING Clause</a>.  Added <a href="#">Examples: Table Operator Function Mapping</a> and <a href="#">Example: SELECT and the Function Mapping Definition</a>.</p> <p><b>DBQL Counts DML Rows in Requests</b>  Added <a href="#">Logging Row Counts for DML Statements</a>.</p>
November 2017	<p><b>CSV Format for DATASET Data Type</b>  <a href="#">Example: Retrieving Data from a DATASET Column in CSV Format</a></p>

# Select Statements

## Overview

These topics describe SELECT statement syntax, options, clauses, and examples of use, in addition to, the request and statement modifiers you can include.

## SELECT

### Purpose

Returns specific row data in the form of a result table.

For information about syntax that is compatible with temporal tables, see *Teradata Vantage™ ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ Temporal Table Support*, B035-1182.

### Required Privileges

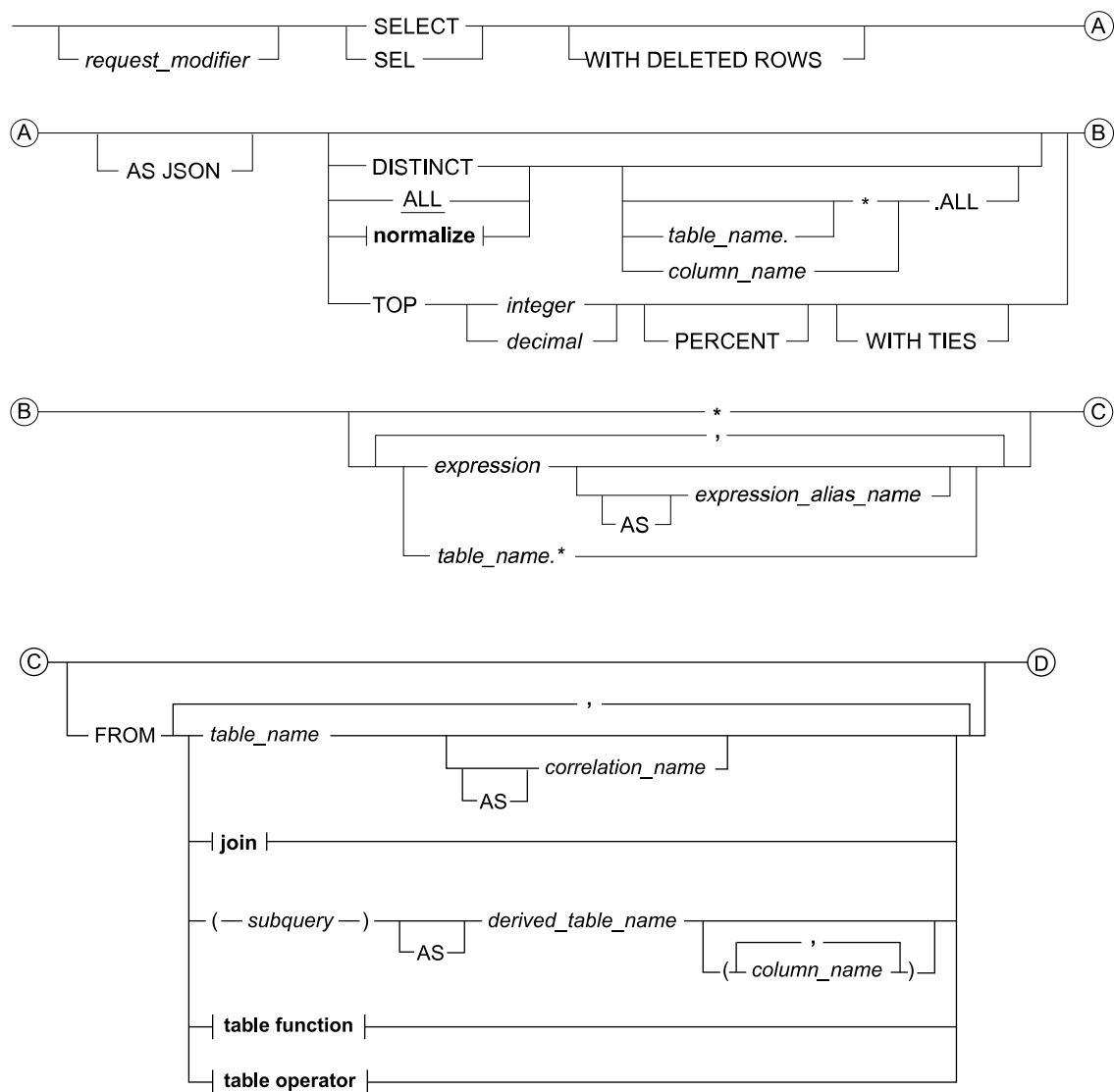
To select data from a table, you must have the SELECT privilege on the table or column set being retrieved.

To select data through a view, you must have the SELECT privilege on that view. Also, the immediate owner of the view (that is, the database or user in which the view resides) must have SELECT WITH GRANT OPTION privileges on all tables or views referenced in the view.

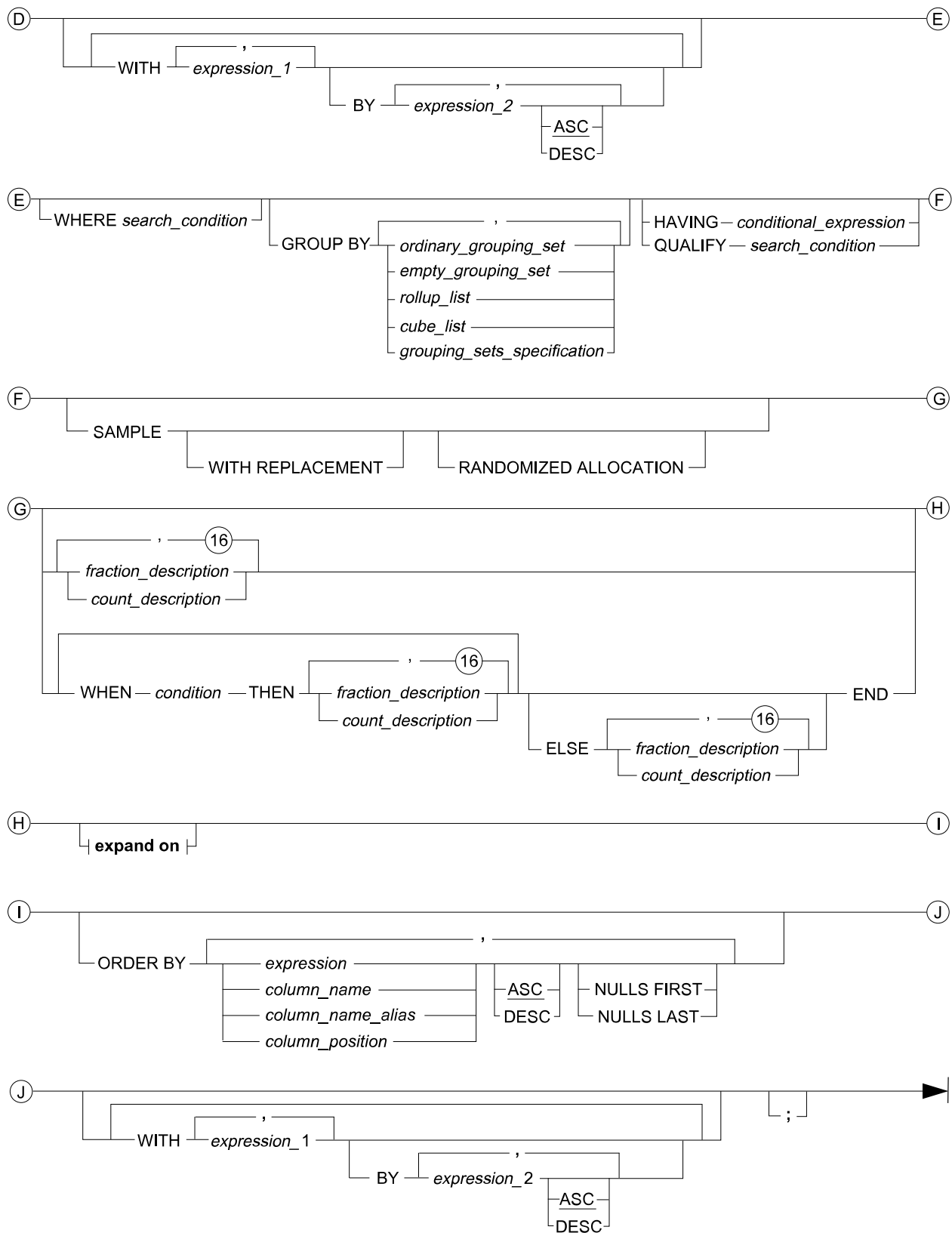
If indirect references are made to a table or view, the privileges must be held by the immediate owner of the object being accessed rather than the user executing the query.

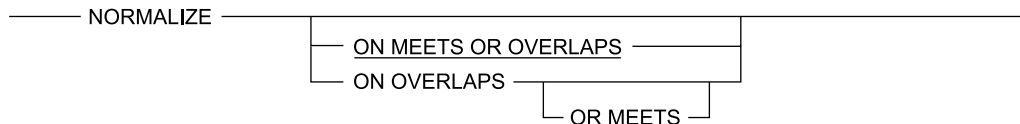
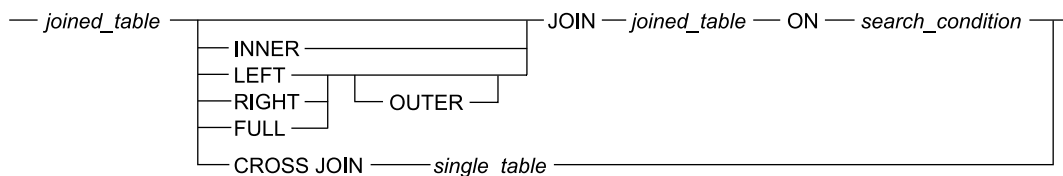
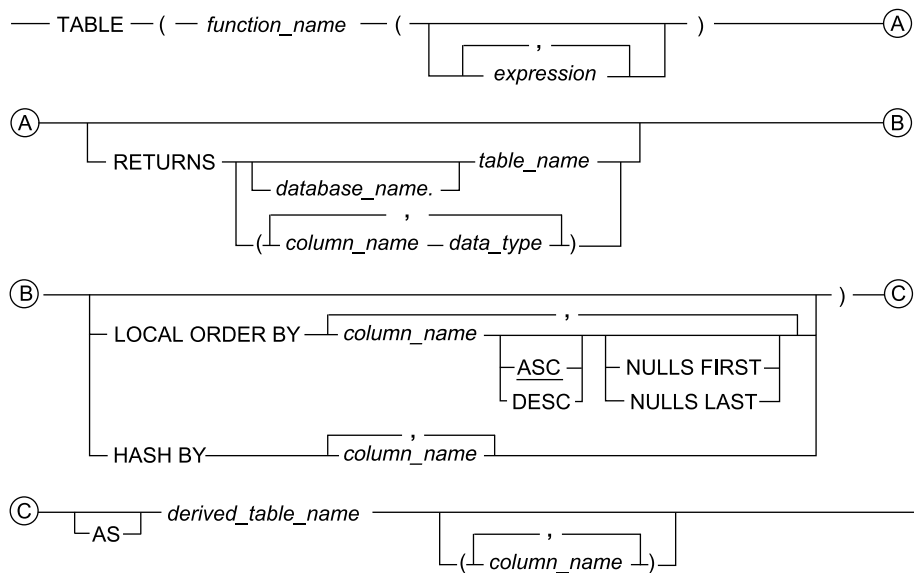
For more information, see *Teradata Vantage™ SQL Data Control Language*, B035-1149 and *Teradata Vantage™ - Database Administration*, B035-1093.

## Syntax

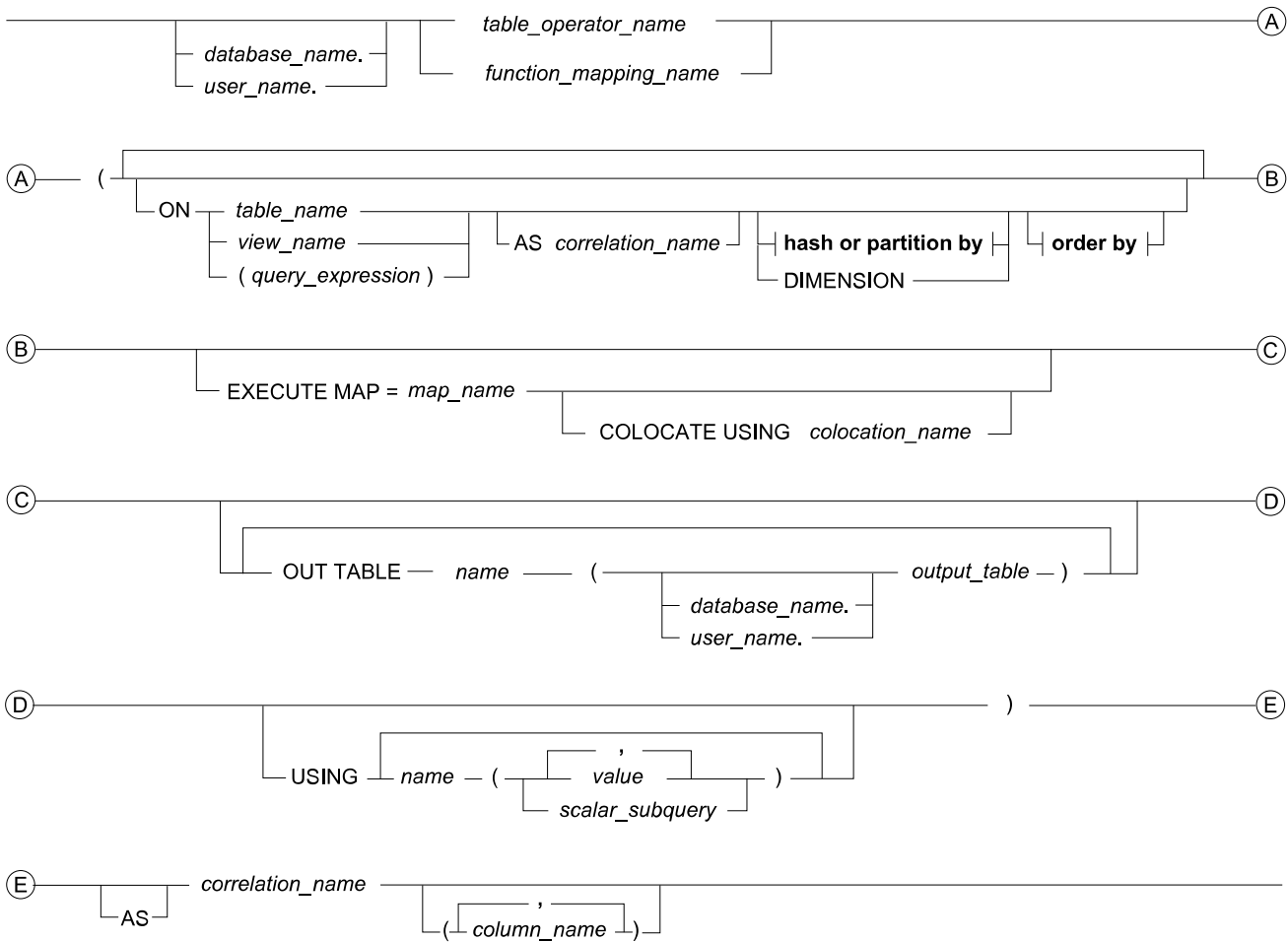




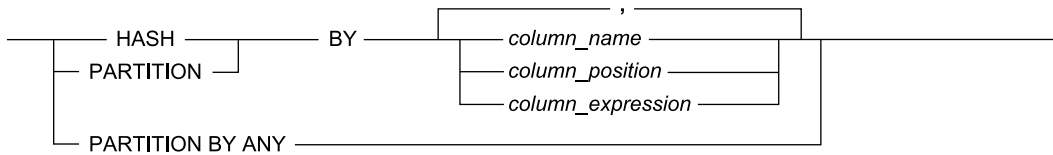


**normalize****join****table function**

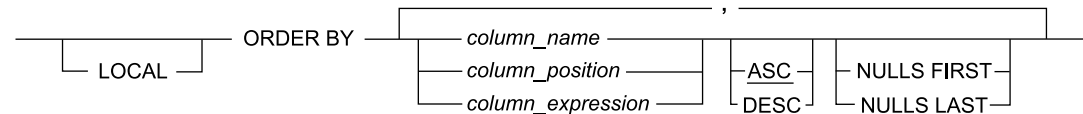
**table operator**

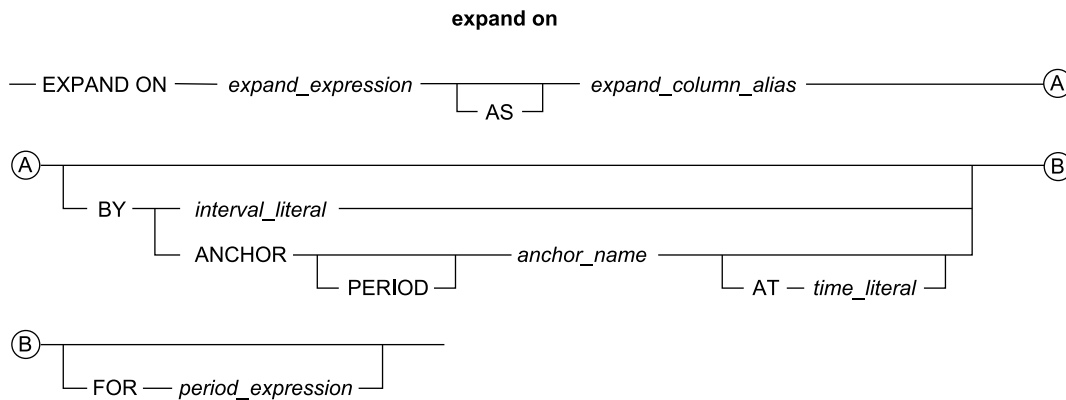


**hash or partition by**



**order by**





## Syntax Elements

### Select List

\*

All columns of all tables referenced in the FROM clause be returned.

When qualified by *table\_name*, specifies that all columns of *table\_name* only are to be returned.

View columns are explicitly enumerated when views are defined. If a table is changed after a view is defined, those changes will not appear if you perform a SELECT \* query.

SELECT \* ... is a column operation, projecting all the columns of a table. SELECT COUNT(\*)... is a row operation, restricting and counting all the rows of a table, then reporting the cardinality of the table in question. The function of the asterisk differs in the two cases.

Since these derived columns are not actual columns of a table, you must explicitly specify PARTITION or PARTITION#L *n*, where *n* ranges from 1 to 62, inclusive, to project the PARTITION or PARTITION#L *n* columns for the table.

#### **expression**

SQL expression, including scalar subqueries and scalar UDFs.

If you specify a scalar UDF, it must return a value expression.

You can specify an expression that returns a UDT in a column list only if its transform group has a fromsql routine. The system automatically converts the expression from its UDT value to the external type via the fromsql routine before returning it to a client application. See [Specifying UDTs in an SQL Request](#).

#### **AS**

Optional introduction to *derived\_column\_name*.

#### **expression\_alias\_name**

An alias for the column expression that is derived from *expression*. You must specify a *derived\_column\_name* for self-join operations.

***table\_name***

Name of a table, queue table, derived table, or view.

*table\_name.\** in the select list can define the table from which rows are to be returned when two or more tables are referenced in the FROM clause.

## ANSI Compliance

SELECT is ANSI SQL:2011-compliant with extensions.

In Teradata SQL, the FROM clause is optional. This is a Teradata extension to the ANSI SQL:2011 standard, in which a FROM clause is mandatory.

The WITH, SAMPLE, QUALIFY, and TOP clauses are Teradata extensions to the ANSI SQL:2011 standard.

The documented syntax specifies the ANSI SQL:2011 ordering of these clauses: FROM, WHERE, GROUP BY, HAVING, and ORDER BY. Teradata Database does not enforce this ordering, but you should observe it when you write applications to maintain ANSI compliance.

## Usage Notes

### Uses of the SELECT Statement

SELECT is the most frequently used SQL statement. Use SELECT statements to specify any of the following:

- The set of result data that is returned
- The order in which result sets are returned
- How the result sets should be grouped
- Whether the result sets should be precisely determined or randomly sampled
- The format in which the results set is reported

Specify fully qualified names in SELECT statements that reference objects in multiple databases and users. Name resolution problems can occur when databases and users referenced in a query contain tables or views with identical names, and those objects are not fully qualified. Name resolution problems can occur even if the identically named objects are not referenced in the query.

### SELECT Subqueries

A subquery is a SELECT expression that is nested within another SQL statement or expression. You can specify SELECT expressions as subqueries in a main query, an outer query, or another subquery for these DML statements:

- [ABORT](#)
- [DELETE](#)

- [INSERT/INSERT ... SELECT](#)
- [MERGE](#)
- [ROLLBACK](#)
- SELECT
- [UPDATE](#)

The upsert form of UPDATE does not support subqueries as WHERE clause conditions. For more information about the upsert form of the UPDATE statement, see [UPDATE \(Upsert Form\)](#).

The syntax rules require that if there are any subqueries in the statement, then any tables referenced in the SELECT must be specified in a FROM clause, either of the subquery itself or of some outer query that contains the subquery.

Correlated subqueries are another category of subquery. A subquery is correlated when it references columns of outer tables in an enclosing or containing, outer query. Correlated subqueries provide an implicit loop function within any standard SQL DML statement.

For details on correlated subqueries and their use, see [Correlated Subqueries](#).

For simple examples of correlated subqueries, see [Examples: SELECT Statements with a Correlated Subquery](#).

Scalar subqueries are another category of subquery that can be coded either as a correlated subquery or as a noncorrelated subquery. A correlated scalar subquery returns a single value for each row of its correlated outer table set. A noncorrelated scalar subquery returns a single value to its containing query. For information about scalar subqueries, see [Scalar Subqueries](#).

## Locks and Concurrency

A DML SELECT sets a default READ lock on the tables or rows referenced in the SELECT statement, depending on session isolation level (refer to the table, below) and the locking level the Lock Manager imposes or the locking level you specify using the LOCKING modifier. See [LOCKING Request Modifier](#). If the SELECT statement references a view, then the system places a READ lock on each of its underlying tables.

---

### Note:

This topic does not apply to DDL statements such as CREATE JOIN INDEX and CREATE VIEW that nest SELECT subqueries in their object definition statements.

---

For outer SELECT statements and SELECT subqueries that are not nested within a DML statement that manipulates data using a DELETE, INSERT, MERGE, or UPDATE statement, the default locking severity is always READ whether the session transaction isolation level is SERIALIZABLE.

If the session transaction isolation level is READ UNCOMMITTED and the DBS Control field AccessLockForUncomRead is set to TRUE, then the default locking level for SELECT operations depends on the factors explained in the following paragraph and table.

For SELECT subqueries that are nested within a DML statement that manipulates data using a DELETE, INSERT, MERGE, or UPDATE statement, the default locking severity is ACCESS when the DBS Control field AccessLockForUncomRead is set TRUE and the transaction isolation level for the session is READ UNCOMMITTED, as the following table indicates.

Transaction Isolation Level	DBS Control AccessLockForUncomRead Field Setting	Default Locking Severity for Outer SELECT and Ordinary SELECT Subquery Operations	Default Locking Severity for SELECT Operations Embedded Within DELETE, INSERT, MERGE, or UPDATE Statements
SERIALIZABLE	FALSE	READ	READ
	TRUE		READ
READ UNCOMMITTED	FALSE		READ
	TRUE		ACCESS

When a SELECT statement does not specify a LOCKING modifier, but a view it uses to access tables does, Teradata Database places the lock specified by the view and does not comply with the defaults described in the preceding table.

For example, suppose you create this view.

```
CREATE VIEW access_view (a, b) AS
LOCKING TABLE acctrec FOR ACCESS
SELECT acctno, qualifyacct
FROM acctrec
WHERE qualifyacct = 1587;
```

This SELECT statement places an ACCESS lock on table *acctrec* regardless of the specified session isolation level and setting of the AccessLockForUncomRead field.

```
SELECT a, b
FROM access_view;
```

For details, see “SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## Derived Tables

A derived table is a transient table that is created dynamically from one or more tables by evaluating a query expression or table expression specified as a subquery within a SELECT statement. You must specify a correlation name for each derived table you create.

The semantics of derived tables and views are identical. See “CREATE VIEW” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## NoPI Tables and SELECT Statements

Because NoPI tables do not have a primary index, single-AMP primary index access is not available. The table must be accessed using a full-table scan, unless one of the following conditions is true:

- Table has a secondary or join index that the Optimizer can use for the access plan, or
- Column or row partition elimination can be applied for a column-partitioned table.

Secondary and join index access paths work the same for NoPI tables as for primary index tables.

## SELECT Statements in Embedded SQL

The rules and restrictions are:

- SELECT must be performed from within a Selection DECLARE CURSOR construct.
- For updatable cursors opened for SELECT statements only, no ORDER BY, GROUP BY, HAVING, or WITH... BY clause is allowed.
- The SELECT privilege is required on all tables specified in the FROM clause and in any subquery contained in the query specification, or on the database set containing the tables.
- The number of columns in the select list must match the number of host variables.
- Teradata Database assigns values to the host variables specified in the INTO clause in the order in which the host variables are specified. The system assigns values to the status parameters SQLSTATE and SQLCODE last.
- The main host variable and the corresponding column in the returned data set must be of the same data type group, except that if the main host variable has an approximate type, then the temporary table column data type can have either an approximate numeric or exact numeric type.
- If the temporary table is empty, then Teradata Database assigns these values to the status parameters:

Status Parameter	Assigned Value
SQLSTATE	'02000'
SQLCODE	+100

Values are not assigned to the host variables specified in the INTO clause.

- If exactly one row of data is returned, the values from the row are assigned to the corresponding host variables specified in the INTO clause.
- If more than one row of data is returned, the system assigns these values to the status parameters:



Status Parameter	Assigned Value
SQLSTATE	'21000'
SQLCODE	-811

Values are not assigned to the host variables specified in the INTO clause.

- If an error occurs in assigning a value to a host variable, the system assigns one of these value sets to the status parameters:

Status Parameter	Assigned Value
SQLSTATE	'22509'
SQLCODE	-303
SQLSTATE	'22003'
SQLCODE	-304
SQLSTATE	'22003'
SQLCODE	-413

Values are not assigned to the host variables specified in the INTO clause.

- If a column value in the returned data is NULL and a corresponding indicator host variable is specified, the value -1 is assigned to the indicator host variable and no value is assigned to the main host variable.

If no corresponding indicator host variable is specified, then Teradata Database assigns these values to the status parameters:

Status Parameter	Assigned Value
SQLSTATE	'22002'
SQLCODE	-305

Values are not assigned to the host variables specified in the INTO clause.

- If a column value in the returned data is NOT NULL and a corresponding indicator host variable is specified, the indicator host variable is set as follows:
  - If the column and main host variable are of CHARACTER data type and the column value is longer than the main host variable, the indicator host variable is set to the length of the column value.
  - In all other cases, the indicator variable is set to zero.

Status parameters have the following default values:

Status Parameter	Assigned Value
SQLSTATE	'00000'
SQLCODE	0

- Column values are set in the corresponding main host variables according to the rules for host variables.
- SELECT ... INTO cannot be performed as a dynamic request.

## DEFAULT Function in SELECT Statements

The rules and restrictions are:

- The DEFAULT function accepts a single expression that identifies a relation column by name. The function evaluates to a value equal to the current default value for the column. For cases where the default value of the column is specified as a current built-in system function, the DEFAULT function evaluates to the current value of system variables at the time the request is executed.

The resulting data type of the DEFAULT function is the data type of the constant or built-in function specified as the default unless the default is NULL. If the default is NULL, the resulting data type of the DEFAULT function is the same as the data type of the column or expression for which the default is being requested.

- The DEFAULT function can be specified as DEFAULT or DEFAULT(*column\_name*). When column name is not specified, the system derives the column based on context. If the column context cannot be derived, the system returns an error.
- You can specify a DEFAULT function with a column name in the select list of a SELECT statement. The DEFAULT function evaluates to the default value of the specified column.
- The DEFAULT function must include a column name in the expression list.
- You can determine the default value for a column by selecting it without specifying a FROM clause for the statement.

When you specify a SELECT statement that does not also specify a FROM clause, the system always returns only a single row with the default value of the column, regardless of how many rows are in the table. This is similar to the behavior of the TYPE function.

- When the column passed as an input argument to the DEFAULT function does not have an explicit default value associated with it, the DEFAULT function evaluates to null.

For more information about using DEFAULT within the SELECT statement, see:

- [WHERE Clause](#)
- [HAVING Clause](#)
- [Rules and Restrictions for the QUALIFY Clause](#)
- [Rules for Using the DEFAULT Function as a Search Condition for an Outer Join ON Clause](#)
- [Inserting When Using a DEFAULT Function](#)

- [Rules for Using the DEFAULT Function With MERGE Requests](#)
- [Rules for Using the DEFAULT Function With Update](#)

## Specifying UDTs in an SQL Request

The rules and restrictions are:

- A UDT must have its tosql and fromsql transform functionality defined prior to its use as a column data type for any table. See “CREATE TRANSFORM” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.
- You can specify an expression that returns a UDT in a column list only if its transform group has a fromsql routine. The system automatically converts the expression from its UDT value to the external type via the fromsql routine before returning it to a client application.
- The only difference between distinct and structured types in this regard is that Teradata Database generates a transform group with both fromsql and tosql routines by default for distinct types, and its external data type is the source data type, while you must explicitly create the transform group for a structured type. For details on how to create a transform group for a UDT, see “CREATE TRANSFORM” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.
- If you submit a SELECT statement with no table references that contains a UDT expression in its select list, and you have not declared transform functionality for it using a UDT literal declared using a NEW expression, then the request aborts and the system returns an error. For information about the NEW expression, see *Teradata Vantage™ SQL Operators and User-Defined Functions*, B035-1210.

## Invoking a Scalar UDF From a SELECT Statement

You must use the alias to reference the result of a scalar UDF invocation that has an alias.

## Returning a Varying Column Table External UDF Result

The rules and restrictions are:

- You can only include table functions in the FROM clause. See [FROM Clause](#).  
The system processes table functions like derived tables.
- The SELECT statement syntax supports the implicit specification of the number of return columns at runtime in its TABLE (*function\_name* RETURNS ... *table\_name*) clause.
- When you reference a table function that does not have a variable number of output columns, the system obtains its explicit output column definitions from DBC.TVFields.

However, when you reference a table function that outputs a dynamic number of columns, you must specify either an output column list or the name of an existing table as part of the function invocation using the RETURNS ... *table\_name* option in the FROM clause of the SELECT request.

SELECT Specification	Columns Returned
Outcome column name list	Specified in the list as the function output columns.
Name of an existing table	From the definition for that table as the function output columns.

In either case, the number of columns specified cannot exceed the defined maximum number of output columns specified by the VARYING COLUMNS *maximum\_output\_columns* specification of its CREATE FUNCTION (Table Form) definition.

See “CREATE FUNCTION (Table Form)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

- To invoke a varying column table function, you can:
  - Specify a list of return column name-data type pairs.
  - Specify the return columns from an existing table.

For examples of both forms returning the same result, see [Example: Dynamic Row Results Returned by Specifying Table Functions](#).

- You cannot specify LOB columns using the BLOB AS LOCATOR or CLOB AS LOCATOR forms.

## SELECT Statements and Derived Period Columns

A SELECT \* on a table with a derived period column returns the two DATE or TIMESTAMP columns used to define the derived period column, not the derived period column. The two DATE or TIMESTAMP columns appear as regular columns in a nontemporal table.

You can only specify derived period columns with period predicate operators, begin and end bound functions, duration functions, comparison operators, logical predicate operators, and in the EXPAND clause. For example, you cannot specify derived period columns in WITH or HAVING clauses. You cannot specify derived period columns in functions, macros, or stored procedures such as P\_INTERSECT, P\_NORMALIZE, LDIFF, or RDIFF, except in a WHERE clause.

You cannot specify derived period columns in the select list.

## SELECT and INSERT-SELECT Statements with Set Operators

Only the SELECT and INSERT-SELECT SQL statements can use the set operators UNION, INTERSECT, and MINUS/EXCEPT. The set operators enable you to manipulate the answers to two or more queries by combining the results of each query into a single result set.

The set operators can also be used within these operations:

- View definitions
- Derived tables
- Subqueries

- INSERT ... SELECT statements

You cannot use the set operators with LOB columns.

## Joins

A SELECT statement can reference data in one or more sets of tables and views, including a mix of tables and views. The SELECT statement can define a join of tables or views.

You can join tables and views that have row-level security constraints, if the tables and views being joined have identical row-level security constraints. Otherwise, the system returns an error.

For more information about join operations, see [Join Expressions](#).

For information about how the Optimizer processes joins, see *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

## Activity Count Indicates the Number of Rows Returned

When the result of a SELECT statement is returned, the activity count success response indicates the number of rows returned. If no rows are returned, then the activity count is 0.

## SELECT and Queue Tables

A queue table is similar to an ordinary base table, with the additional unique property of being similar to an asynchronous first-in-first-out (FIFO) queue.

The first column of a queue table always contains Queue Insertion TimeStamp (QITS) values. The QITS value of a row indicates the time the row was inserted into the queue table, unless a different, user-supplied value is inserted.

The CREATE TABLE statement for the table must define the first column with the following data type and attributes:

```
TIMESTAMP(6) NOT NULL DEFAULT CURRENT_TIMESTAMP(6)
```

You can use SELECT statements to perform a FIFO peek on a queue table. In this case, data is returned from the queue table without deleting any of its rows.

If no rows are available, no rows are returned. The transaction does not enter a delay state.

To return the queue in FIFO order, specify the first column of the queue table in the ORDER BY clause of the SELECT statement.

To perform a FIFO pop on a queue table, use a SELECT AND CONSUME statement. For more information, see [SELECT AND CONSUME](#).

**Note:**

A SELECT AND CONSUME statement cannot specify a scalar subquery.

For more information about how to define a queue table, see “CREATE TABLE (Queue Table Form)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

## Logging Problematic Queries

The recommended best practice for logging problematic DML requests (queries) is to specify XMLPLAN logging whenever you submit the queries. The XMLPLAN option captures (logs) the query plan generated by the Optimizer as an XML document that can be used by Teradata support tools to diagnose performance issues with the query. The query plan is stored in the DBC.DBQLXMLTbl system table. For information on how to capture query plans for DML requests using the XMLPLAN option, see “Using BEGIN QUERY LOGGING to Log Query Plan Information” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184. For information about the BEGIN QUERY LOGGING statement syntax and descriptions of the statement options, see *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## Examples

### Examples: SELECT Statements

The following SELECT statements are syntactically correct in Teradata SQL. The first statement does not reference any tables, so a FROM clause is unnecessary. This syntax is a Teradata extension to the ANSI SQL:2011 standard.

```
SELECT DATE, TIME;
```

```
SELECT x1,x2
FROM t1,t2
WHERE t1.x1=t2.x2;
```

```
SELECT x1
FROM t1
WHERE x1 IN (SELECT x2
             FROM t2);
```

### Examples: SELECT Statements with a Correlated Subquery

Use a correlated subquery to return the names of the employees with the highest salary in each department.

```

SELECT name
FROM personnel p
WHERE salary = (SELECT MAX(salary)
                FROM personnel sp
                WHERE p.department=sp.department);

```

Use a correlated subquery to return the names of publishers without books in the library.

```

SELECT pubname
FROM publisher
WHERE 0 = (SELECT COUNT(*)
           FROM book
           WHERE book.pubnum=publisher.pubnum);

```

## Example: SELECT Statements With Scalar Subqueries in Expressions and as Arguments to Built-In Functions

You can specify a scalar subquery in the same way that you specify a column or constant in an expression or as an argument to a system function. You can specify an expression that is composed of a scalar subquery wherever an expression is allowed in a DML statement.

Following are examples of the types of expressions that you can code using scalar subqueries.

### Arithmetic expressions

```

SELECT (fix_cost + (SELECT SUM(part_cost)
                    FROM parts)) AS total_cost, ...

```

### String expressions

```

SELECT (SELECT prod_name
        FROM prod_table AS p
        WHERE p.pno = s.pno) || store_no ...

```

### CASE expressions

```

SELECT CASE WHEN (SELECT count(*)
                  FROM inventory
                  WHERE inventory.pno = orders.pno) > 0
        THEN 1
        ELSE 0
END, ...

```

**Aggregate expressions**

```
SELECT SUM(SELECT count(*)
           FROM sales
           WHERE sales.txn_no = receipts.txn_no), ...
```

**Value list expressions**

```
... WHERE txn_no IN (1,2, (SELECT MAX(txn_no)
                          FROM sales
                          WHERE sale_date = CURRENT_DATE));
```

**Examples: SELECT and PARTITION****Example: Partition Lock for a Row-Partitioned Table**

In this example, the query retrieves rows from table slppit1 with a WHERE condition on the partitioning column that qualifies a single partition. An all-AMPs partition lock is used to lock the single partition being accessed and a proxy lock is used to serialize the placement of the all-AMPs partition range lock.

The table definition for this example is as follows:

```
CREATE TABLE HLSDS.SLPPIT1 (PI INT, PC INT, X INT, Y INT)
PRIMARY INDEX (PI)
PARTITION BY (RANGE_N(PC BETWEEN 1 AND 10 EACH 1));
```

An EXPLAIN of the SELECT statement shows the partition lock:

```
Explain SELECT * FROM HLSDS.SLPPIT1 WHERE PC = 5;
1) First, we lock HLSDS.slppit1 for read on a reserved RowHash in a
   single partition to prevent global deadlock.
2) Next, we lock HLSDS.slppit1 for read on a single partition.
3) We do an all-AMPs RETRIEVE step from a single partition of HLSDS.slppit1
   with a condition of ("HLSDS.slppit1.Pc = 5") with a residual condition
   of ("HLSDS.slppit1.Pc = 5") into Spool 1 (group_amps), which is built
   locally on the AMPs. The size of Spool 1 is estimated with no confidence
   to be 1 row (65 bytes). The estimated time for this step is 0.07 seconds.
4) Finally, we send out an END TRANSACTION step to all AMPs involved in
   processing the request.
```

**Example: SELECT Statements That Specify a System-Derived PARTITION Column In Their Select Lists**

You can specify a system-derived PARTITION column in the select list of a statement. This example specifies an unqualified PARTITION column because its use is unambiguous:

```
SELECT orders.*, PARTITION
FROM orders
WHERE orders.PARTITION = 10
AND orders.o_totalprice > 100;
```



PARTITION must be qualified in this SELECT statement in the select list and in the WHERE clause because it joins two tables and specifies the PARTITION columns of both in the WHERE clause:

```
SELECT orders.*, lineitem.*, orders.PARTITION
FROM orders, lineitem
WHERE orders.PARTITION = 3
AND   lineitem.PARTITION = 5
AND   orders.o_orderkey = lineitem.l_orderkey;
```

You must specify PARTITION explicitly in the select list. If you specify \* only, then PARTITION or PARTITION#L *n* column information is not returned.

Usually, the table would be a row-partitioned table because:

- For a nonpartitioned table, PARTITION is always 0.
- For a column-partitioned table without row partitioning, PARTITION is always 1.

### **Example: PARTITION Values Not Returned Because PARTITION Not Specified in the Select List**

In this example, the value of PARTITION is not returned as one of the column values, even though it is specified in the WHERE clause, because it was not explicitly specified in the select list for the query:

```
SELECT *
FROM orders
WHERE orders.PARTITION = 10
AND   orders.o_totalprice > 19.99;
```

### **Example: Qualification of PARTITION Not Necessary Because Specification Is Unambiguous**

PARTITION does not have to be qualified in this example because its use is unambiguous:

```
SELECT orders.*, PARTITION
FROM orders
WHERE orders.PARTITION = 10
AND   orders.o_totalprice > 100;
```

### **Example: Qualification of PARTITION Necessary to Avoid Ambiguity**

PARTITION must be qualified in the two following examples to distinguish between PARTITION values in the *orders* table and PARTITION values in the *lineitem* table:

```
SELECT *
FROM orders, lineitem
WHERE orders.PARTITION = 3
AND   lineitem.PARTITION = 5
AND   orders.o_orderkey = lineitem.l_orderkey;
```

```
SELECT orders.*, lineitem.*, orders.PARTITION
FROM orders, lineitem
WHERE orders.PARTITION = 3
AND lineitem.PARTITION = 5
AND orders.o_orderkey = lineitem.l_orderkey;
```

### Example: Selecting All Active Partitions From a Table

The following two examples provide a list of the populated row partitions in the *orders* table (assuming the maximum combined partition number for a populated row partition is 999 or 127, respectively, for the *orders* table):

```
SELECT DISTINCT PARTITION (FORMAT '999')
FROM orders
ORDER BY PARTITION;

SELECT DISTINCT CAST (PARTITION AS BYTEINT)
FROM orders
ORDER BY PARTITION;
```

### Example: Using PARTITION With An Aggregate Function

The following example counts the number of rows in each populated row partition:

```
SELECT PARTITION, COUNT(*)
FROM orders
GROUP BY PARTITION
ORDER BY PARTITION;
```

### Example: SELECT With Queue Tables

The following statement measures the queue depth of a queue table named *shopping\_cart*:

```
SELECT COUNT(*)
FROM shopping_cart;
```

Assuming the column named *cart\_qits* contains QITS values, this statement returns the *shopping\_cart* queue table in FIFO order:

```
SELECT *
FROM shopping_cart
ORDER BY cart_qits;
```

## Example: Specifying UDTs in the SELECT List and WHERE Clause of SELECT Statements

The following set of simple examples shows the valid use of UDT expressions in a SELECT statement:

```
SELECT euro_column
FROM t1;

SELECT euro_column.roundup(0)
FROM t1;

SELECT address_column.street(), address_column.zip()
FROM t2;

SELECT t.address_column.ALL
FROM t2 AS t;

SELECT address_column
FROM t2
WHERE address_column.zip() = '92127';

SELECT *
FROM t3
WHERE circle_column = NEW circle(1.34, 4.56, 10.3);

SELECT circle_column.area()
FROM t3;
```

The following example shows the need to cast the distinct UDT column named myDollar to compare it with a DECIMAL value.

```
SELECT *
FROM t4
WHERE myDollar < (CAST 3.20 AS DollarUDT);
```

## Example: SELECT Statements Specifying the DEFAULT Function

You can specify the DEFAULT function with a column name in the select projection list. The DEFAULT function evaluates to the default value of the specified column.

Assume this table definition for the examples below:

```
CREATE TABLE table14 (
  col1 INTEGER,
  col2 INTEGER DEFAULT 10,
  col3 INTEGER DEFAULT 20,
  col4 CHARACTER(60)
  col5 TIMESTAMP DEFAULT NULL,
  col6 TIMESTAMP DEFAULT CURRENT_TIMESTAMP);
```

The following query returns the default value of col2 and col3 of table14 in all the resulting rows. This is an inefficient form to get the default value of a column, because the system does a full table scan to perform the query. A more efficient method to get the default value of a column is to specify the query without a FROM clause.

```
SELECT DEFAULT(col2), DEFAULT(col3)
FROM table14;
```

Assuming there are four rows in table14, the returned row set is as follows:

DEFAULT(col2)	DEFAULT(col3)
-----	-----
10	20
10	20
10	20
10	20

If there are no rows in the table, then no rows are returned.

The following example returns the default value of col2 from table14. This helps discover the default value of a particular column. This is an inefficient form to get the default value of a column, because the system performs a full table scan to perform the query. A more efficient method to get the default value of a column is to specify the query without a FROM clause.

```
SELECT DISTINCT DEFAULT(col2)
FROM Table14;
```

The returned row set is as follows:

DEFAULT(col2)	DEFAULT(col3)
-----	-----
10	20

If there are no rows in the table, then no rows are returned.

When this SELECT statement does not specify a FROM clause, the system always returns a single row with the default value of the column, regardless of how many rows are in the table. This behavior is similar to the TYPE function.

```
SELECT DEFAULT(table14.col2);
```

The resulting row set is as follows:

```
DEFAULT(col2)
-----
          10
```

When the column passed as an input argument to the DEFAULT function does not have an explicit default value associated with it, the DEFAULT function evaluates to null.

Assume this table definition for the examples below:

```
CREATE TABLE table15 (
  col1 INTEGER ,
  col2 INTEGER NOT NULL,
  col3 INTEGER NOT NULL DEFAULT NULL,
  col4 INTEGER CHECK (col4 > 10) DEFAULT 9);
```

Because col1 does not have an explicit default value, the DEFAULT function evaluates to null.

```
SELECT DEFAULT(Table15.Col1);
```

The resulting row set is as follows:

```
DEFAULT(col1)
-----
          ?
```

Because col2 does not have an explicit default value, the DEFAULT function evaluates to null. The function returns null even though the NOT NULL constraint on the column does not permit nulls in the column.

```
SELECT DEFAULT(table15.col2);
```

The resulting row set is as follows:

```
DEFAULT(col2)
-----
          ?
```

Because col3 has an explicit default value of null, the DEFAULT function evaluates to its explicit default value. The function returns null even though the NOT NULL constraint on the column does not permit nulls in the column.

```
SELECT DEFAULT(table15.col3);
```

The resulting row set is as follows:

```

DEFAULT(col3)
-----
?
```

Even though col4 has a default value 9, which violates the CHECK constraint col4>10, the DEFAULT function returns the value 9 even though check constraint does not permit this value in the column.

```
SELECT DEFAULT(Table15.Col4);
```

The resulting row set is as follows:

```

DEFAULT(Col4)
-----
9
```

## Example: Dynamic Row Results Returned by Specifying Table Functions

The following example uses the extract\_store\_data table function to return dynamic row results in the store\_data table:

```

SELECT *
FROM (TABLE(extract_store_data('...', 1000)
RETURNS store_data) AS store_sales;
```

The following equivalent examples use the sales\_retrieve table function to return dynamic row results either by specifying the maximum number of output columns and their individual column names and data types or by specifying only the name of the table into which the converted rows are to be written.

```

SELECT *
FROM TABLE (sales_retrieve(9005)
RETURNS (store    INTEGER,
         item     INTEGER,
         quantity INTEGER)) AS s;

SELECT *
FROM TABLE (sales_retrieve(9005)
RETURNS sales_table) AS s;
```

## Example: Scalar Subquery in the Select List of a SELECT Statement

You can specify a scalar subquery as a column expression or parameterized value in the select list of a query. You can assign an alias to a scalar subquery defined in the select list, thus enabling the rest of the query to reference the subquery by that alias.

The following example specifies a scalar subquery (SELECT AVG(price)...) in its select list with an alias (avgprice) and then refers to it in the WHERE clause predicate (t2.price < avgprice).

```
SELECT category, title, (SELECT AVG(price)
                        FROM movie_titles AS t1
                        WHERE t1.category=t2.category) AS avgprice
FROM movie_titles AS t2
WHERE t2.price < avgprice;
```

For additional examples of specifying scalar subqueries in SELECT statements, see:

- [Example: Scalar Subquery in the WHERE Clause of a SELECT Statement](#)
- [Example: SELECT Statement With a Scalar Subquery in Its GROUP BY Clause](#)
- [Example: Scalar Subquery in the WITH ... BY Clause of a SELECT Statement](#)
- [Example: Scalar Subquery in the ON Clause of a Left Outer Join](#)

## Example: SQL UDF in the Select List of a SELECT Statement

You can specify an SQL UDF in the select list of an SQL request. As if true of other expressions in a select list, you can also alias an SQL UDF. In this request, the aliased value expression cve is specified as cve in the select list and in the WHERE clause.

```
SELECT test.value_expression(t1.a1, t2.as) AS cve, cve+1
FROM t1, t2
WHERE t1.b1 = t2.b2
AND    cve = 5;
```

## Example: Creating a Time Series Using Expansion By an Interval Constant Value

For additional examples of using the EXPAND ON clause in SELECT statements, see [EXPAND ON Clause](#).

Suppose you create a table named employee with this definition.

```
CREATE SET TABLE employee (
  emp_id      INTEGER,
  emp_name    CHARACTER(20) CHARACTER SET LATIN NOT CASESPECIFIC,
  job_duration PERIOD(DATE))
PRIMARY INDEX (emp_id);
```

You insert 3 rows into the employee table:

employee		
emp_id	emp_name	job_duration
1001	Xavier	2002-01-10, 9999-12-31
1002	Ricci	2007-07-01, 9999-12-31
1003	Charles	2006-02-10, 2008-06-01

When you specify an EXPAND ON clause using an interval constant, Teradata Database expands each row based on the specified interval value, and the duration in the expanded period is the interval value.

So you now use an EXPAND ON clause to retrieve the employee details specifying an interval constant period.

```
SELECT emp_id, emp_name, job_duration AS tsp
FROM employee
EXPAND ON job_duration AS tsp BY INTERVAL '1' YEAR
FOR PERIOD(DATE '2006-01-01', DATE '2008-01-01');
```

Teradata Database returns employee details for each year of a given period, as you specified in the FOR clause of the SELECT statement.

Teradata Database returns a warning to the requestor that some rows in the expanded result might have an expanded period duration that is less than the duration of the specified interval.

emp_id	emp_name	job_duration	tsp
-----	-----	-----	---
1003	Charles	2006-02-10, 2008-06-01	2006-02-10, 2007-02-10
1003	Charles	2006-02-10, 2008-06-01	2007-02-10, 2008-01-01
1002	Ricci	2007-07-01, 9999-12-31	2007-07-01, 2008-01-01
1001	Xavier	2002-01-10, 9999-12-31	2006-01-01, 2007-01-01
1001	Xavier	2002-01-10, 9999-12-31	2007-01-01, 2008-01-01

## Example: Invoking an SQL UDF From a Derived Table

This example invokes the SQL UDF value\_expression in the select list of the derived table dt.

```
SELECT *
FROM (SELECT a1, test.value_expression(3,4), c1
      FROM t1
      WHERE a1>b1) AS dt (a,b,c);
```



This example invokes the SQL UDF `value_expression` in the WHERE clause of the derived table `dt`.

```
SELECT *
FROM (SELECT a1, b1, c1
      FROM t1
      WHERE test.value_expression(b1,c1)>10) AS dt (a,b,c);
```

## Example: Invoking a UDF or Method Using a RETURNS Specification

The table definitions used in this example are:

```
CREATE TABLE t1 (
  int_col      INTEGER,
  var_char_col VARCHAR(40) CHARACTER SET UNICODE);
```

```
CREATE TABLE t2 (
  int_col      INTEGER,
  decimal_col  DECIMAL (10, 6));
```

These function definitions are used in this example to identify the `TD_ANYTYPE` parameters:

```
CREATE FUNCTION udf_1(
  A    INTEGER,
  B    TD_ANYTYPE)
RETURNS TD_ANYTYPE;
```

```
CREATE FUNCTION udf_3(
  A    INTEGER,
  B    TD_ANYTYPE)
RETURNS TD_ANYTYPE;
```

The following example invokes `udf_1` using `t1.int_col` and `t2.decimal_col` as parameters and `DECIMAL(10,6)` as the explicitly specified return data type.

```
SELECT (udf_1 (t1.int_col, t2.decimal_col)
RETURNS DECIMAL(10,6));
```

The following example invokes `udf_3` using `t1.var_char_col` and `t2.decimal_col` as parameters and `VARCHAR(40)` as the explicitly specified return data type.

```
SELECT (udf_3 (t1.var_char_col, t2.decimal_col)
RETURNS VARCHAR(40) CHARACTER SET LATIN);
```

The final example invokes `method_2` using `t2.decimal_col` as a parameter and `DECIMAL(10,6)` as the explicitly specified return data type.

```
SELECT udt_col.(method_2(t2.decimal_col)
RETURNS DECIMAL(10,6));
```

## Example: Invoking a UDF or Method Using a RETURNS STYLE Specification

The table definitions used in this example are:

```
CREATE TABLE t1 (
  int_col      INTEGER,
  var_char_col VARCHAR(40) CHARACTER SET UNICODE);
```

```
CREATE TABLE t2 (
  int_col      INTEGER,
  decimal_col  DECIMAL (10, 6));
```

These function definitions are used in this example to identify the TD\_ANYTYPE parameters:

```
CREATE FUNCTION udf_1(
  A    INTEGER,
  B    TD_ANYTYPE)
  RETURNS TD_ANYTYPE;
```

```
CREATE FUNCTION udf_3(
  A    INTEGER,
  B    TD_ANYTYPE)
  RETURNS TD_ANYTYPE;
```

The following example invokes `udf_2` using `t1.var_char_col` and `t2.decimal_col` as parameters and `DECIMAL(10,6)` as the implicitly specified return data type because the data type for column `t2.decimal_col` is `DECIMAL(10,6)`.

```
SELECT (udf_2 (t1.var_char_col, t2.decimal_col)
  RETURNS STYLE t2.decimal_col);
```

The following example invokes `udf_3` using `t1.var_char_col` and `t2.decimal_col` as parameters and `VARCHAR(40) CHARACTER SET UNICODE` as the implicitly specified return data type because the data type for column `t1.var_char_col` is `VARCHAR(40) CHARACTER SET UNICODE`.

```
SELECT (udf_3 (t1.var_char_col, t2.decimal_col)
  RETURNS STYLE t1.var_char_col);
```

The final example invokes `method_2` using `t2.decimal_col` as a parameter and `DECIMAL(10,6)` as the implicitly specified return data type because the data type for column `t2.decimal_col` is `DECIMAL(10,6)`.

```
SELECT udt_col.(method_2(t2.decimal_col)
  RETURNS STYLE t2.decimal_col);
```

## Example: Selecting Rows From a Table With Row-Level Security Protection

This example shows how you can specify a row-level security constraint column name in the select list or WHERE clause of a SELECT statement.

First, define the row-level security constraint. The create text for the group\_membership constraint object looks like this.

```
CREATE CONSTRAINT group_membership SMALLINT, NOT NULL,
VALUES (exec:100, manager:90, clerk:50, peon:25),
INSERT SYSLIB.ins_grp_member,
SELECT SYSLIB.rd_grp_member;
```

In this table definition, Teradata Database implicitly adds a row-level security constraint column named group\_membership to the emp\_record table when it creates that table. The group\_membership column contains the data for the row-level security constraint.

```
CREATE TABLE emp_record (
  emp_name  VARCHAR(30),
  emp_number INTEGER,
  salary    INTEGER,
  group_membership CONSTRAINT)
UNIQUE PRIMARY INDEX(emp_name);
```

After you create emp\_record, you can use a SELECT statement that retrieves the specified data from the group\_membership column by specifying the name of that column in the select list for the query.

Following is an example of a SELECT statement on table emp\_record that includes group\_membership in the select list.

```
SELECT emp_name, group_membership
FROM emp_record
WHERE group_membership=90;
```

If you have the required security credentials, this query returns the emp\_name and group\_membership value name and value code for all managers. If you do not have the required credentials, no rows are returned. You cannot specify a value name as the search condition. You must specify a value code. In this example, the value code 90 represents the value name manager.

Suppose you then create this row-level security constraints and inventory table.

```
CREATE CONSTRAINT classification_level SMALLINT, NOT NULL,
VALUES (top_secret:4, secret:3, confidential:2, unclassified:1),
INSERT SYSLIB.InsertLevel,
UPDATE SYSLIB.UpdateLevel,
```

```

DELETE SYSLIB.DeleteLevel,
SELECT SYSLIB.ReadLevel;

CREATE CONSTRAINT classification_category BYTE(8)
VALUES (nato:1, united_states:2, canada:3, united_kingdom:4,
       france:5, norway:6, russia:7),
INSERT SYSLIB.insert_category,
UPDATE SYSLIB.update_category,
DELETE SYSLIB.delete_category,
SELECT SYSLIB.read_category;

CREATE TABLE inventory (
  column_1 INTEGER,
  column_2 INTEGER,
  column_3 VARCHAR(30),
  classification_level CONSTRAINT,
  classification_category CONSTRAINT)
PRIMARY INDEX(column_1);

```

User joe then logs onto a Teradata Database session. The create text for joe looks like this.

```

CREATE USER joe AS
PERMANENT=1e6,
PASSWORD=Joe1234,
CONSTRAINT = classification_level (top_secret),
CONSTRAINT = classification_category (united_states);

```

Because user joe is defined with the classification\_level and classification\_category row-level security constraints, he can execute this SELECT statement on inventory.

```

SELECT *
FROM inventory
WHERE column_1 = 1212;

```

The result set looks similar to this, returning not only the data from the three columns explicitly defined for inventory, but also the data from the two row-level security columns.

column_1	column_2	column_3	classification_level	classification_category
1212	90505	Widgets	3	'4000000000000000'xb

## Example: Row-Level Security Constraint and SELECT Statement When User Lacks Required Privileges

This example shows how the SELECT constraint is applied when a user that does not have the required OVERRIDE privileges attempts to execute a SELECT statement on a table that has the row-level security SELECT constraint:

```
SELECT * FROM rls_tbl;
```

SELECT statement processing includes a step to RETRIEVE from RS.rls\_tbl with a condition of SYSLIB.SELECTCATEGORIES ( '90000000'XB, RS.rls\_tbl.categories ) AND SYSLIB.SELECTLEVEL (2, RS.rls\_tbl.levels).

The table definition used in this example is:

```
CREATE TABLE rls_tbl(
  col1 INT,
  col2 INT,
  classification_levels CONSTRAINT,
  classification_categories CONSTRAINT);
```

The user's session constraint values are:

```
Constraint1Name LEVELS
Constraint1Value 2
Constraint3Name CATEGORIES
Constraint3Value '90000000'xb
```

## Example: Using a Table Operator with Multiple PARTITION BY Inputs

The following example shows one way that you can use table operator functionality with multiple PARTITION BY inputs.

Suppose that you have the following tables:

WebLog		
cookie	cart_amount	page
AAAA	\$60	Thankyou
AAAA	\$140	Thankyou
BBBB	\$100	Thankyou
CCCC		Intro

WebLog		
cookie	cart_amount	page
CCCC	\$200	Thankyou
DDDD	\$100	Thankyou

AdLog		
cookie	ad_name	action
AAAA	Champs	Impression
AAAA	Puppies	Click
BBBB	Apples	Click
CCCC	Baseball	Impression
CCCC	Apples	Click

In this example, the operator `attribute_sales` returns the amount of sales revenue that is attributed to online ads. The inputs to the operator are sales information from the store's web logs and logs from the ad server. Both the input tables are partitioned on the user's browser cookie. The operator also accepts two custom clauses, `Clicks` and `Impressions`, which supply the percentages of sales to attribute to ad clicks and the views that lead to a purchase, respectively.

The syntax is as follows:

```
SELECT adname, cart_amt
FROM attribute_sales (
  ON (SELECT cookie, cart_amt FROM weblog WHERE page = 'thankyou' )
      as W PARTITION BY cookie
  ON adlog as S PARTITION BY cookie
  USING clicks(.8) impressions(.2)) AS D1(adname,attr_revenue) ;
```

The output looks similar to the following:

ad_name -----	attr_revenue -----
Champs	\$40
Puppies	\$160
Apples	\$240

	Baseball		\$40
--	----------	--	------

### Example: Using a Multiple Input Table Operator with DIMENSION Input

In this example, the table operator `closest_store` finds the closest retail store when a purchase is made over a cell phone. The operator takes the following input data sets:

- The `phone_purchases` table, which contains entries for purchases made over a cell phone, along with normalized spatial coordinates of the phone when the online purchase was made.
- The `stores` table, which contains the location of all the retail stores and their associated normalized spatial coordinates. The `stores` table is a smaller fact table that is provided as DIMENSION input.

pid	x_coordinate	y_coordinate
P0	2	1
P1	1	5
P2	3	2
P3	0	4

sid	x_coordinate	y_coordinate
S0	1	4
S1	2	3

The SELECT syntax is as follows:

```
SELECT pid, sid
  FROM closest_store (
    ON phone_purchases PARTITION BY pid,
    ON stores DIMENSION) AS D;
```

The output looks similar to the following:

	pid ---		sid ---
	P0		S1
	P1		S0

	P2		\$S1
	P3		\$S0

You can also make the table operator call using `PARTITION BY ANY`, which would keep the existing distribution of the rows on the AMPs. The `SELECT` syntax is as follows:

```
SELECT pid, sid
FROM closest_store (
  ON phone_purchases PARTITION BY ANY,
  ON stores DIMENSION) AS D;
```

## Example: Retrieving Data from a DATASET Column in CSV Format

Here is the table definition for this example.

```
CREATE TABLE myDatasetTable01(
  id INTEGER,
  csvFile DATASET STORAGE FORMAT CSV);
```

Then, we insert the following CSV data into this column, where the first line contains the header, columns are delimited by the ampersand character (&) and rows are delimited by the number sign character (#). An ampersand not followed by characters indicates a NULL value.

```
Item ID&Item Name&Item Color&Item Style&Quantity Purchased&Item Price&Total
Price#55&bicycle&red&boys&1&100.00&100.00#88&toy
boat&pink&&1&15.10&15.10#105&soap&&&1&0.99&0.99
```

This statement retrieves the column data:

```
SELECT csvFile FROM myDatasetTable01;
csvFile
-----
> Item ID&Item Name&Item Color&Item Style&Quantity Purchased&Item Price&Total
Price#55&bicycle&red&boys&1&100.00&100.00#88&toy
boat&pink&&1&15.10&15.10#105&soap&&&1&0.99&0.99
```

## Related Topics

For more information on using `SELECT` statements, see:

- [SELECT AND CONSUME](#)
- [WITH Modifier](#)
- [Distinct](#)



- [ALL](#)
- [NORMALIZE](#)
- [.ALL Operator](#)
- [TOP n](#)
- [FROM Clause](#)
- [Derived Tables](#)
- [WHERE Clause](#)
- [Specifying Subqueries in Search Conditions](#)
- [Correlated Subqueries](#)
- [Scalar Subqueries](#)
- [GROUP BY Clause](#)
- [HAVING Clause](#)
- [QUALIFY Clause](#)
- [SAMPLE Clause](#)
- [SAMPLEID Expression](#)
- [EXPAND ON Clause](#)
- [ORDER BY Clause](#)
- [WITH Clause](#)
- “SELECT ... INTO” in *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148.
- “SELECT” (Temporal Form)” in *Teradata Vantage™ Temporal Table Support*, B035-1182.
- SQL operators and functions in *Teradata Vantage™ SQL Date and Time Functions and Expressions*, B035-1211.
- JSON data types and methods in *Teradata Vantage™ JSON Data Type*, B035-1150.

## SELECT AND CONSUME

### Purpose

Returns data from the row with the oldest insertion timestamp in the specified queue table and deletes the row from the queue table.

### Syntax

```
SELECT AND CONSUME TOP 1 — select_list — FROM — queue_table_name —————▶
                                     └──┬──┘
                                     ;
```

### Required Privileges

To perform a SELECT AND CONSUME from a queue table, you must have the SELECT and DELETE privileges on that table.

## Syntax Elements

### *select\_list*

An asterisk ( \* ) or a comma-separated list of valid SQL expressions.

If *select\_list* specifies \*, all columns from the queue table specified in the FROM clause are returned.

The select list must not contain aggregate or ordered analytical functions.

### *queue\_table\_name*

Name of a queue table that was created with the QUEUE option in the CREATE TABLE statement.

## ANSI Compliance

SELECT AND CONSUME is a Teradata extension to the ANSI SQL:2011 standard.

## Usage Notes

### Locking and Concurrency

A SELECT AND CONSUME operation sets a rowhash-level WRITE lock. To upgrade to an EXCLUSIVE lock, use the LOCKING request modifier. See [LOCKING Request Modifier](#).

If the request enters a delay state, the lock is not granted until after a row is inserted into the queue table.

### Transaction Processing Semantics

Multiple SELECT AND CONSUME statements can appear in a transaction when:

- The session mode is ANSI.
- The session mode is Teradata and the SELECT AND CONSUME statements appear between BEGIN TRANSACTION (BT) and END TRANSACTION (ET) statements.

A SELECT AND CONSUME statement can consume any rows inserted into a queue table within the same transaction.

When performing SELECT AND CONSUME statements:

- Perform SELECT AND CONSUME statements early in a transaction to avoid conflicts with other database resources.
- Perform one SELECT AND CONSUME statement per transaction, if possible.

## FIFO Operations and SELECT AND CONSUME

### Attributes of a Queue Table

A queue table is similar to an ordinary base table, with the additional unique property of behaving like an asynchronous first-in-first-out (FIFO) queue.

The first column of a queue table contains Queue Insertion TimeStamp (QITS) values. The CREATE TABLE request must define the first column with these data type and attributes:

```
TIMESTAMP(6) NOT NULL DEFAULT CURRENT_TIMESTAMP(6)
```

The QITS value of a row indicates the time the row was inserted into the queue table, unless a different, user-supplied value is inserted.

### FIFO Ordering of Queue Table Rows Is Approximate

A queue table provides only provides an approximate FIFO ordering of its rows for these reasons:

- System clocks on the nodes of an MPP system are not synchronized.
- A user-supplied value that is different from the current timestamp can be inserted into the QITS column.
- The rollback of a transaction or a request may restore rows with an earlier QITS value than rows already consumed.
- Multiple rows use the same value for the QITS column when the INSERT statements in a multistatement request use the default value.
- Teradata workload analysis management software rules can defer the SELECT AND CONSUME statement. For example, if a statement references a user, account, or other object that has a rule defined on it, the statement is subject to that rule. Deferment might be because of a Teradata dynamic workload management software throttle against an attribute of the containing request that causes it to be delayed until some currently running requests finish.

The best practice is not to apply Teradata workload analyzer restrictions to sessions that execute SELECT AND CONSUME statements.

For information about creating workload management rules, see *Teradata® Workload Analyzer User Guide*, B035-2514.

### SELECT AND CONSUME Statements Operate Like a FIFO Pop Operation

SELECT AND CONSUME statements operate like a FIFO pop in these ways.

- Data is returned from the row that has the oldest value in the QITS column of the specified queue table.
- The row is deleted from the queue table, guaranteeing that the row is processed only once.

## Rules and Restrictions for SELECT AND CONSUME Statements

The rules and restrictions are:

- Only one SELECT AND CONSUME statement can appear in a multistatement request.
- A multistatement request that includes a SELECT AND CONSUME statement cannot also include a DELETE, UPDATE, UPSERT, or MERGE statement that references the same queue table.
- SELECT AND CONSUME statements cannot be specified in any of these SQL constructs:
  - Subqueries with the exception INSERT ... SELECT statements
  - Search conditions
  - Logical predicates
  - Operations using the SQL UNION, MINUS, or EXCEPT/MINUS SET operators
- A limit of 20 percent of the total possible PE sessions (24 per PE by default) can enter a delay state as a result of submitting a SELECT AND CONSUME statement.
- For embedded SQL, the selection form of a DECLARE CURSOR statement cannot specify a SELECT AND CONSUME statement if the preprocessor TRANSACT or -tr option is set to ANSI.

## Performance Characteristics of SELECT AND CONSUME Statements

The performance of SELECT AND CONSUME statements is:

- Similar to that of selecting a single row from a base table using a USI.
- More expensive than selecting a single row from a base table using a primary index.

If a queue table does not contain rows when you perform a SELECT AND CONSUME statement, its containing transaction enters a delay state until one of the following occurs:

- A row is inserted into the queue table.
- The transaction aborts as a result of one of these reasons.
  - Direct user intervention, such as an ABORT statement
  - Indirect user intervention, such as a DROP TABLE statement on the queue table

## Examples

### Example: Returning the Row Having the Oldest QITS Value in a Queue Table

This CREATE TABLE statement creates a queue table named shopping\_cart:

```
CREATE SET TABLE shopping_cart, QUEUE (
  cart_qits  TIMESTAMP(6) NOT NULL DEFAULT CURRENT_TIMESTAMP(6),
```

```
product_id INTEGER,
quantity    INTEGER )
PRIMARY INDEX (product_id);
```

This SELECT AND CONSUME statement returns all the columns of the row with the oldest value in the cart\_qits column of the shopping\_cart table:

```
SELECT AND CONSUME TOP 1 * FROM shopping_cart;
```

## Related Topics

For more information about:

- Using SELECT requests to browse through queue table data without deleting rows, see [SELECT and Queue Tables](#)
- Inserting data into queue tables, see [Inserting Rows into Queue Tables](#).
- Using cursors with queue tables, see *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148.
- Using SELECT AND CONSUME ... INTO statements with embedded SQL, see *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148.
- Creating queue tables, see “CREATE TABLE (Queue Table Form)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

## SELECT ... INTO

For information on this statement, see *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148.

## Request Modifier

For information on this statement, see *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148.

### ***request\_modifier***

Optional clauses that modify the SELECT statement using one or more of these request or statement modifiers:

- [EXPLAIN Request Modifier](#)
- [LOCKING Request Modifier](#)
- [USING Request Modifier](#)
- [WITH Modifier](#)

Request and statement modifiers apply to all statements in a request.

# WITH Modifier

## Purpose

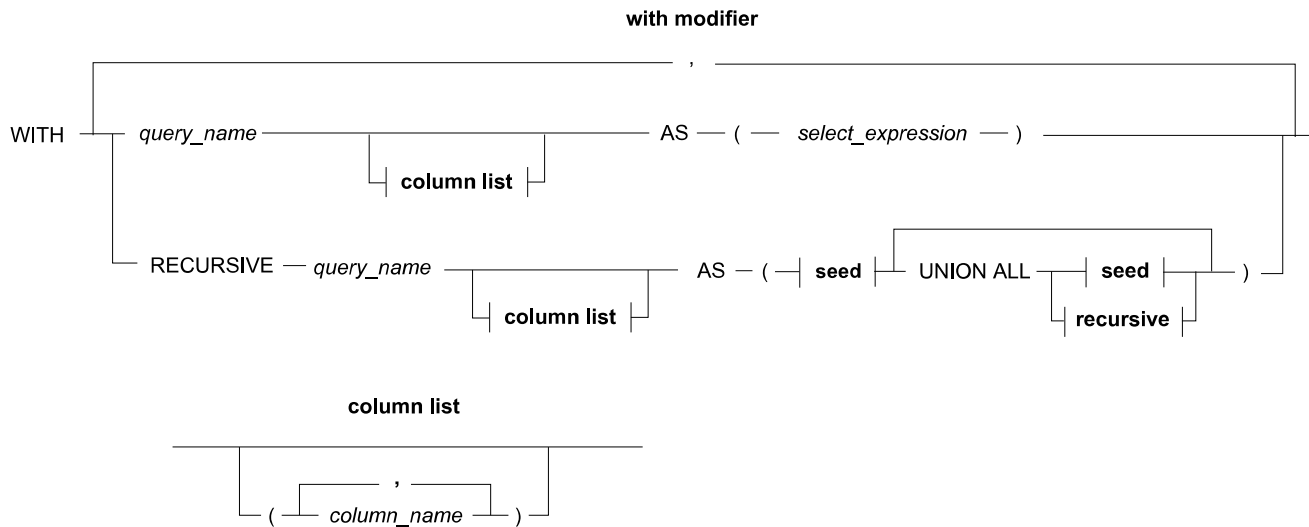
You can define multiple queries with a single SELECT statement by specifying the queries by name.

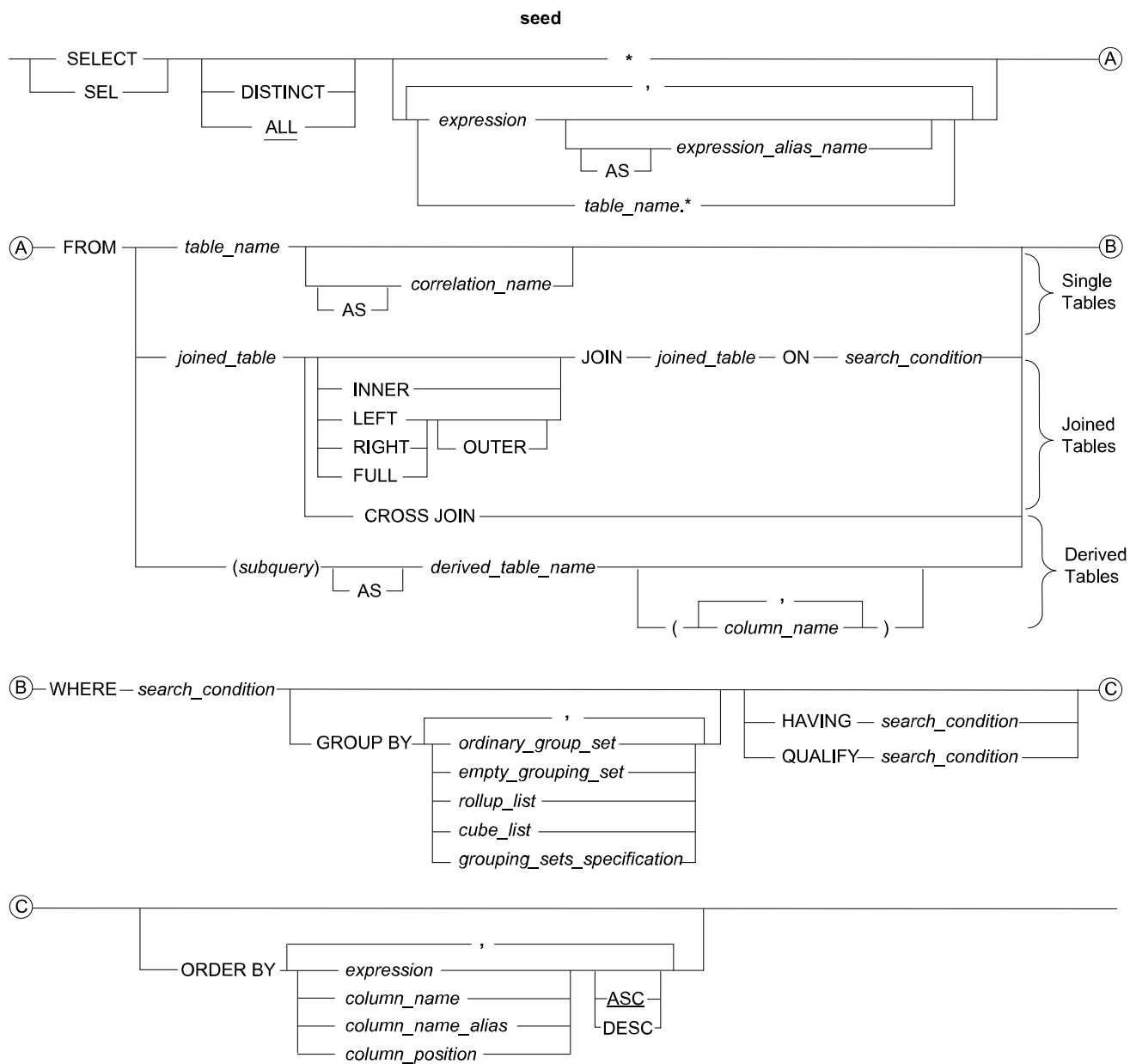
The WITH modifier defines a named query list from which the SELECT statement can select data. In this way, a query in the WITH modifier is similar to a derived table. Named queries are also referred to as common table expressions (CTEs).

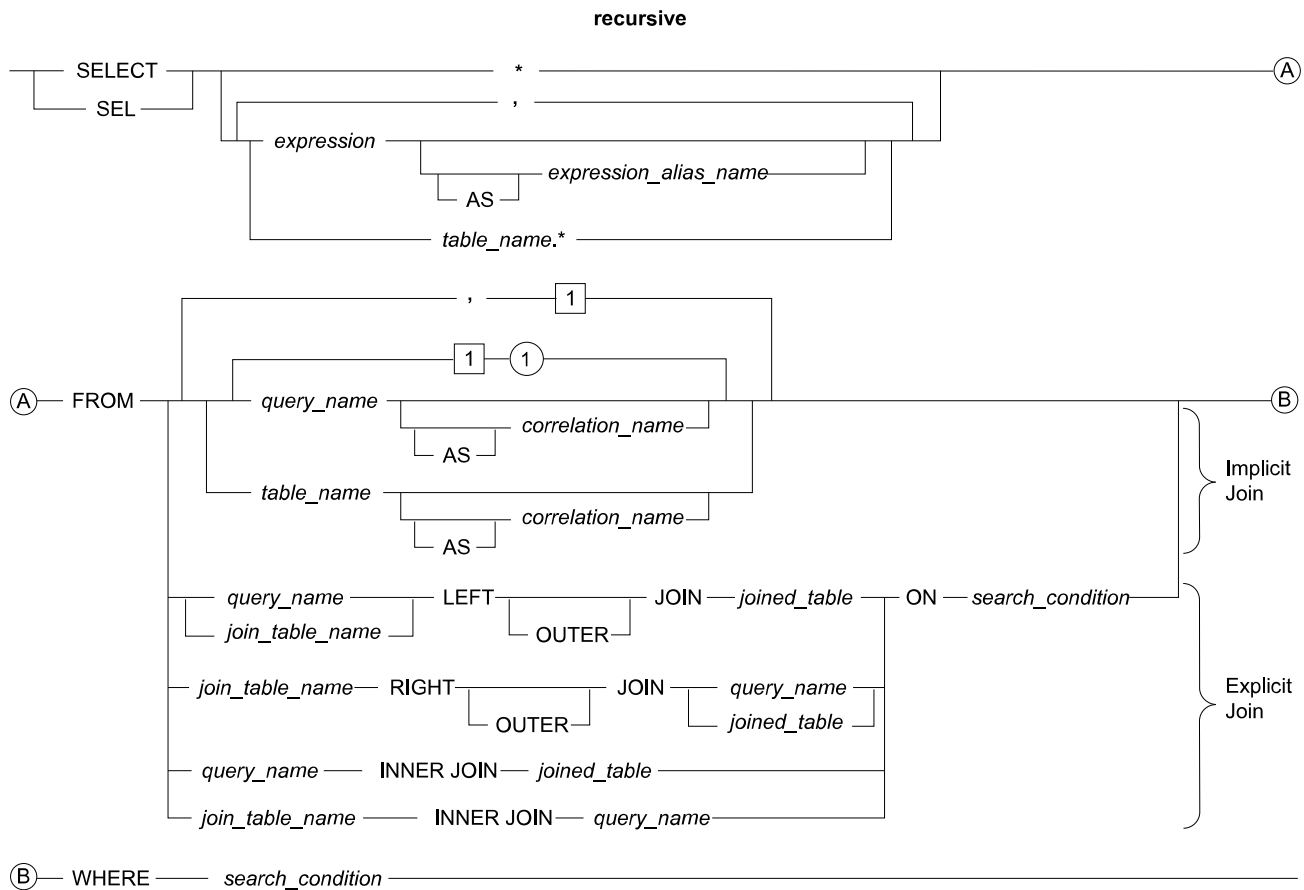
## Note:

You can only use the WITH modifier with SELECT statements. You cannot use the WITH modifier with other DML statements.

## Syntax







## Syntax Elements

### Nonrecursive Form

#### **query\_name**

Name of the query.

#### **column\_name**

Name of a column in the named query definition.

#### **select\_expression**

Nonrecursive SELECT statement that retrieves the row data to store in the named query.

You can specify an expression that returns a UDT in a column list only if the transform group has a fromsql routine. The system automatically converts the expression from its UDT value to the external type using the fromsql routine before returning the result to a client application.

You can specify a row-level security constraint column in the select list of a SELECT request. However, the column cannot be specified as part of an arithmetic expression in the select list.



The value returned for the column is the coded value for the row-level security constraint from the row.

## Recursive Form

A RECURSIVE named query that can refer to itself in the query definition. The named query list consists of at least one nonrecursive, or seed, statement and at least one recursive statement.

### **RECURSIVE** *query\_name*

Recursive query name.

### *column\_name*

Name of a column in the named query definition.

### **UNION ALL**

Operator that adds results of iterative operations to the named query.

## Seed Statement

The seed statement is a nonrecursive SELECT statement that retrieves row data from other tables to store in the named query.

### **DISTINCT**

Only one row is returned from any set of duplicates that result from a given expression list.

Two rows are considered duplicates only if each value in one is equal to the corresponding value in the other.

### **ALL**

Return all rows, including duplicates, in the results of the expression list. This is the default value.

\*

Return all columns of all tables referenced in the FROM clause of the seed statement.

When qualified by *table\_name*, return all columns of *table\_name* only.

### *expression*

Any valid SQL expression, including scalar UDFs.

### **AS**

Optional introduction to *expression\_alias\_name*.

### *expression\_alias\_name*

Alias for the expression.

### *table\_name*

Name of a table, derived table, or view.

*table\_name.\** in the select list can define the table from which rows are to be returned when two or more tables are referenced in the FROM clause.

**FROM**

Introduction to the names of one or more tables, views, or derived tables from which *expression* is to be derived.

The FROM clause in a seed statement cannot specify the TABLE option.

**Single Table**

This option enables the FROM clause of the seed statement to specify single tables.

A FROM clause can include a sequence of single table references, which creates an implicit inner join.

A FROM clause in a seed statement cannot specify the name of the WITH RECURSIVE query.

***table\_name***

Name of a single table, derived table, table UDF, or view referred to in the FROM clause.

**AS**

Optional introduction to *correlation\_name*.

***correlation\_name***

Alias for the table referenced in the FROM clause.

**Joined Tables**

Options for joined tables enable the FROM clause of the seed statement to specify that multiple tables be joined in explicit ways, as described below.

***joined\_table***

Name of a joined table.

**INNER**

Join in which qualifying rows from one table are combined with qualifying rows from another table according to a join condition.

Inner join is the default join type.

**LEFT OUTER**

Outer join with the table that was listed first in the FROM clause.

In a LEFT OUTER JOIN, the rows from the left table that are not returned in the result of the inner join of the two tables are returned in the outer join result and extended with nulls.

**RIGHT OUTER**

Outer join with the table that was listed second in the FROM clause.

In a RIGHT OUTER JOIN, the rows from the right table that are not returned in the result of the inner join of the two tables are returned in the outer join result and extended with nulls.

**FULL OUTER**

Rows are returned from both tables.

In a FULL OUTER JOIN, rows from both tables that have not been returned in the result of the inner join are returned in the outer join result and extended with nulls.

**JOIN**

Introduction to the name of the second table to participate in the join.

**ON *search\_condition***

One or more conditional expressions that must be satisfied by the result rows.

An ON condition clause is required if the FROM clause specifies an outer join.

A value you specify for a row-level security constraint in a search condition must be in encoded form.

**CROSS JOIN**

Unconstrained or Cartesian join, returns all rows from all tables specified in the FROM clause.

In a FULL OUTER JOIN, rows from both tables that have not been returned in the result of the inner join are returned in the outer join result and extended with nulls.

**Derived Tables**

The derived table option enables the FROM clause of a seed statement to specify a spool made up of selected data from an underlying table set. The derived table acts like a viewed table.

**(*subquery*)**

Nested SELECT statements.

SELECT AND CONSUME statements cannot be used in a subquery.

You can specify NORMALIZE in a subquery.

**AS**

Optional introduction to *derived\_table\_name*.

***derived\_table\_name***

Name of the derived table.

***column\_name***

Column name. This field is for the column name only. Do not use forms such as `Tablename.Columnname` or `Databasename.Tablename.Columnname`.

Columns with a UDT type are valid with exceptions.

**WHERE**

Introduction to the search condition in the seed statement.

***search\_condition***

Conditional search expression that must be satisfied by the row or rows returned by the seed statement.

If you specify the value for a row-level security constraint in a search condition, that value must be expressed in encoded form.

**GROUP BY**

Introduction to the specification of how to group result rows.

***ordinary\_group\_set***

Column expression by which the rows returned by the seed statement are grouped.

The expression cannot group result rows that have a LOB, ARRAY, or VARRAY type.

*ordinary\_grouping\_set* falls into three general categories:

- *column\_name*
- *column\_position*
- *column\_expression*

***empty\_grouping\_set***

Contiguous LEFT PARENTHESIS, RIGHT PARENTHESIS pair with no argument. You use this syntax to request a grand total of the computed group totals.

***rollup\_list***

ROLLUP expression that reports result rows in a single dimension with one or more levels of detail.

The expression cannot group result rows that have a LOB, ARRAY, or VARRAY type.

***cube\_list***

CUBE expression that reports result rows in multiple dimensions with one or more levels of detail.

The expression cannot group result rows that have a LOB, ARRAY, or VARRAY type.

***grouping\_sets\_specification***

GROUPING SETS expression that reports result rows in one of two ways:

- As a single dimension, but without a full ROLLUP.
- As multiple dimensions, but without a full CUBE.

**HAVING**

Introduction to the conditional clause in the SELECT statement.

***search\_condition***

One or more conditional expressions that must be satisfied by the result rows.

If you specify the value for a row-level security constraint in a search condition, that value must be expressed in encoded form.

**QUALIFY**

Introduction to a conditional clause that filters rows from a WHERE clause. The difference between QUALIFY and HAVING is that QUALIFY filtering is based on the result of performing various ordered analytical functions on the data.

***search\_condition***

One or more conditional expressions that must be satisfied by the result rows.

If you specify the value for a row-level security constraint in a search condition, that value must be expressed in encoded form.

**ORDER BY**

Order in which result rows are to be sorted.

***expression***

Expression in the SELECT expression list of the select statement, either by name or by means of a constant that specifies the numeric position of the expression in the expression list.

***column\_name***

Names of columns used in the ORDER BY clause in the SELECT statement. These can be ascending or descending.

***column\_name\_alias***

Column name alias specified in the select expression list of the query for the column on which the result rows are to be sorted.

If you specify a *column\_name\_alias* to sort by, then that alias cannot match the name of any column that is defined in the table definition for any table referenced in the FROM clause of the query whether that column is specified in the select list or not. The system always references the underlying physical column having the name rather than the column that you attempt to reference using that same name as its alias.

You can specify the sort column by *column\_position* value within the select list for the query.

***column\_position***

Numeric position of the columns specified by the ORDER BY clause. These can be ascending or descending.

**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.

**Recursive Statement**

The recursive statement is a SELECT statement that retrieves row data from a join of the named query and other tables.

You cannot include NORMALIZE in a recursive statement of a recursive query.

\*

All columns of all tables referenced in the FROM clause of the recursive statement be returned.

When qualified by *table\_name*, specifies that all columns of *table\_name* only are to be returned.

***expression***

Any valid SQL expression, with these exceptions:

- Aggregate functions
- Ordered analytical functions

**AS**

Optional introduction to *expression\_alias\_name*.

***expression\_alias\_name***

Alias for the *expression*.

***table\_name***

Name of the named query or the name of a table or view.

*table\_name.\** in the select list can define the table from which rows are to be returned when two or more tables are referenced in the FROM clause.

**FROM**

Introduction to the named query and one or more tables or views from which *expression* is to be derived.

The FROM clause in a recursive statement cannot specify the TABLE option.

**Implicit Join**

This option enables the FROM clause of the recursive statement to specify the name of the RECURSIVE query and one or more single table references, creating an implicit inner join.

***query\_name***

Named query referred to in the FROM clause.

**AS**

Optional introduction to *correlation\_name*.

***correlation\_name***

Alias for the query name referenced in the FROM clause.

***table\_name***

Name of a single table or view referred to in the FROM clause.

**AS**

Optional introduction to *correlation\_name*.

***correlation\_name***

Alias for the table name referenced in the FROM clause.

**Explicit Join**

Options for joined tables enable the FROM clause of the seed statement to specify that multiple tables be joined in explicit ways, described as follows.

***query\_name***

Named query referred to in the FROM clause.

***join\_table\_name***

Name of a joined table.

**INNER JOIN**

Join in which qualifying rows from one table are combined with qualifying rows from another table according to a join condition.

Inner join is the default join type.

**LEFT OUTER**

Outer join with the table that was listed first in the FROM clause.

In a LEFT OUTER JOIN, the rows from the left table that are not returned in the result of the inner join of the two tables are returned in the outer join result and extended with nulls.

**RIGHT OUTER**

Outer join with the table that was listed second in the FROM clause.

In a RIGHT OUTER JOIN, the rows from the right table that are not returned in the result of the inner join of the two tables are returned in the outer join result and extended with nulls.

**JOIN**

Introduction to the name of the second table to participate in the join.

***joined\_table***

Name of the joined table.

**ON *search\_condition***

One or more conditional expressions that must be satisfied by the result rows.

A value you specify for a row-level security constraint in a search condition must be in encoded form.

An ON condition clause is required if the FROM clause specifies an outer join.

**WHERE Clause****WHERE**

Keyword that introduces the search condition in the recursive statement.

***search\_condition***

A conditional search expression that must be satisfied by the row or rows returned by the recursive statement.

A value you specify for a row-level security constraint in a search condition must be in encoded form.

**ANSI Compliance**

The WITH modifier is ANSI SQL:2011 compliant.

Other SQL dialects support similar non-ANSI standard statements with names such as:

- CONNECT BY PRIOR

## Usage Notes

### Nonrecursive Named Query in a WITH Modifier

Consider these table definitions:

```
CREATE TABLE product (
  product_id INTEGER,
  on_hand    INTEGER);

CREATE TABLE stocked (
  store_id   INTEGER,
  product_id INTEGER,
  quantity   INTEGER);
```

The following statement uses a nonrecursive WITH statement modifier to define a temporary named result set called `orderable_items` that is built from the select expression that follows the AS keyword:

```
WITH orderable_items (product_id, quantity) AS
( SELECT stocked.product_id, stocked.quantity
  FROM stocked, product
  WHERE stocked.product_id = product.product_id
    AND product.on_hand > 5
)

SELECT product_id, quantity
  FROM orderable_items
  WHERE quantity < 10;
```

The same results are produced by this statement using a derived table:

```
SELECT product_id, quantity
  FROM (SELECT stocked.product_id, stocked.quantity
        FROM stocked, product
        WHERE stocked.product_id = product.product_id
          AND product.on_hand > 5) AS orderable_items
  WHERE quantity < 10;
```

### RECURSIVE Named Query in a WITH Modifier

A recursive named query provides a way to iteratively query hierarchies of data, such as an organizational structure, bill-of-materials, and document hierarchy.

A recursive query cannot reference another recursive query, either directly or indirectly.



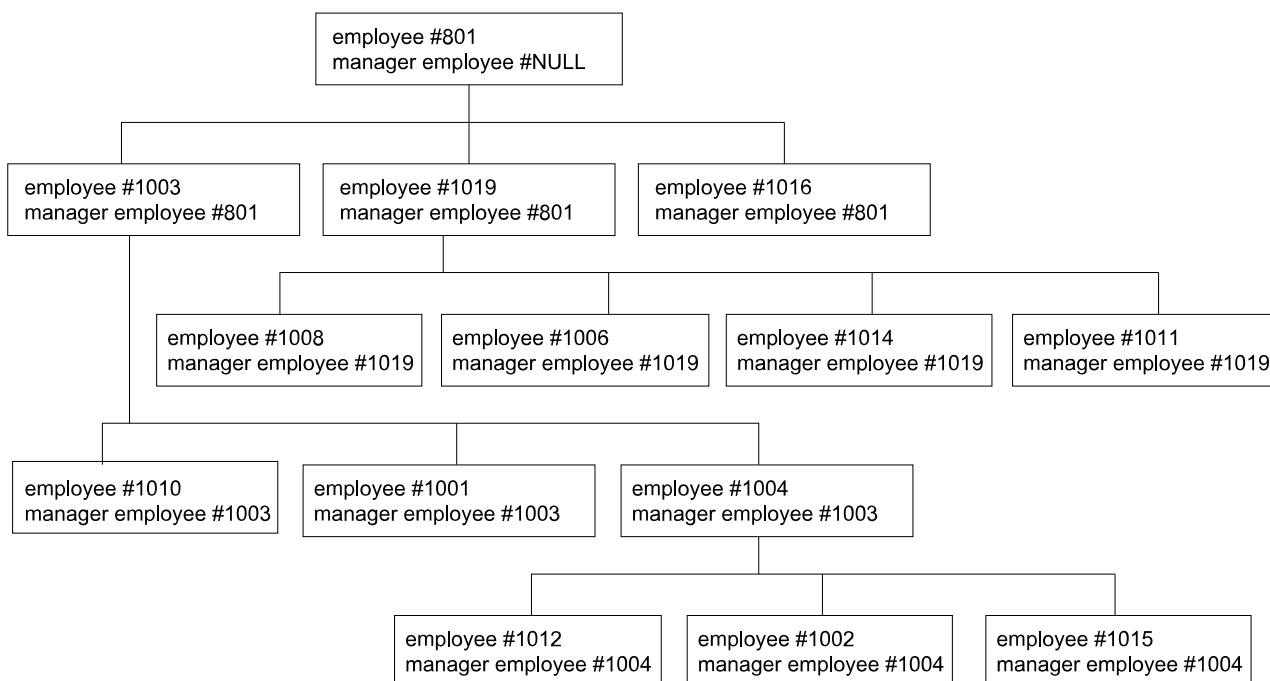
A recursive named query has three execution phases:

1. Create an initial nonrecursive, or seed, result set.
2. Recurse the intermediate result sets based on the seed result set until no new rows are added to the temporary named result set.
3. Execute a last query on the temporary named result set to return the final result set.

Consider this employee table:

```
CREATE TABLE employee (
  employee_number      INTEGER,
  manager_employee_number INTEGER,
  last_name            CHARACTER(20),
  first_name           VARCHAR(30));
```

The table represents an organizational structure of employee-manager relationships. The employee table is similar to this organization chart.



The following recursive query retrieves the employee numbers of all employees who directly or indirectly report to the manager who has an employee\_number value of 801:

```
WITH RECURSIVE temp_table (employee_number) AS
  (SELECT root.employee_number
   FROM employee AS root
   WHERE root.manager_employee_number = 801
  UNION ALL
```

```

        SELECT indirect.employee_number
        FROM temp_table AS direct, employee AS indirect
        WHERE direct.employee_number = indirect.manager_employee_number
    )
    SELECT *
    FROM temp_table
    ORDER BY employee_number;

```

In the example, `temp_table` is a temporary named result set that can be referred to in the `FROM` clause of the recursive statement.

The initial result set is established in `temp_table` by the nonrecursive, or seed, statement and contains the employees that report directly to the manager with an `employee_number` of 801:

```

    SELECT root.employee_number
    FROM employee AS root
    WHERE root.manager_employee_number = 801

```

The recursion takes place by joining each employee in `temp_table` with employees who report to the employees in `temp_table`. The `UNION ALL` adds the results to `temp_table`.

```

    SELECT indirect.employee_number
    FROM temp_table AS direct, employee AS indirect
    WHERE direct.employee_number = indirect.manager_employee_number

```

Recursion stops when no new rows are added to `temp_table`.

The final query is not part of the recursive `WITH` request modifier and extracts the employee information from `temp_table`:

```

    SELECT *
    FROM temp_table
    ORDER BY employee_number;

```

The results of the recursive query are as follows:

```

employee_number
-----
          1001
          1002
          1003
          1004
          1006
          1008
          1010
          1011
          1012

```

```

1014
1015
1016
1019

```

## WITH Modifiers

The rules and restrictions are:

- The only set operator that can appear in a recursive named query within a WITH modifier is UNION ALL.
- The following elements cannot appear within a WITH or WITH RECURSIVE modifier:
  - WITH or WITH RECURSIVE modifier
  - TOP *n* operator
  - User-defined functions
- The following elements cannot appear within a recursive statement in a WITH RECURSIVE modifier:
  - NOT IN or NOT EXISTS logical predicate
  - Aggregate functions
  - Ordered analytical functions
  - GROUP BY clause
  - HAVING clause
  - DISTINCT clause
  - Subqueries
  - Derived tables
- You cannot specify a WITH modifier in the definitions of any of these database objects:
  - Triggers
  - Stored procedures
- A recursive named query that does not have a recursive statement works like a nonrecursive named query.

This request produces the same results as the request that specifies a nonrecursive named query in the WITH modifier:

```

WITH RECURSIVE orderable_items (product_id, quantity) AS (
  SELECT stocked.product_id, stocked.quantity
  FROM stocked, product
  WHERE stocked.product_id = product.product_id
  AND   product.on_hand > 5)
SELECT product_id, quantity
FROM orderable_items
WHERE quantity < 10;

```

## RECURSIVE Named Query in a WITH Modifier

A recursive named query provides a way to iteratively query hierarchies of data, such as an organizational structure, bill-of-materials, and document hierarchy.

A recursive query cannot reference another recursive query, either directly or indirectly.

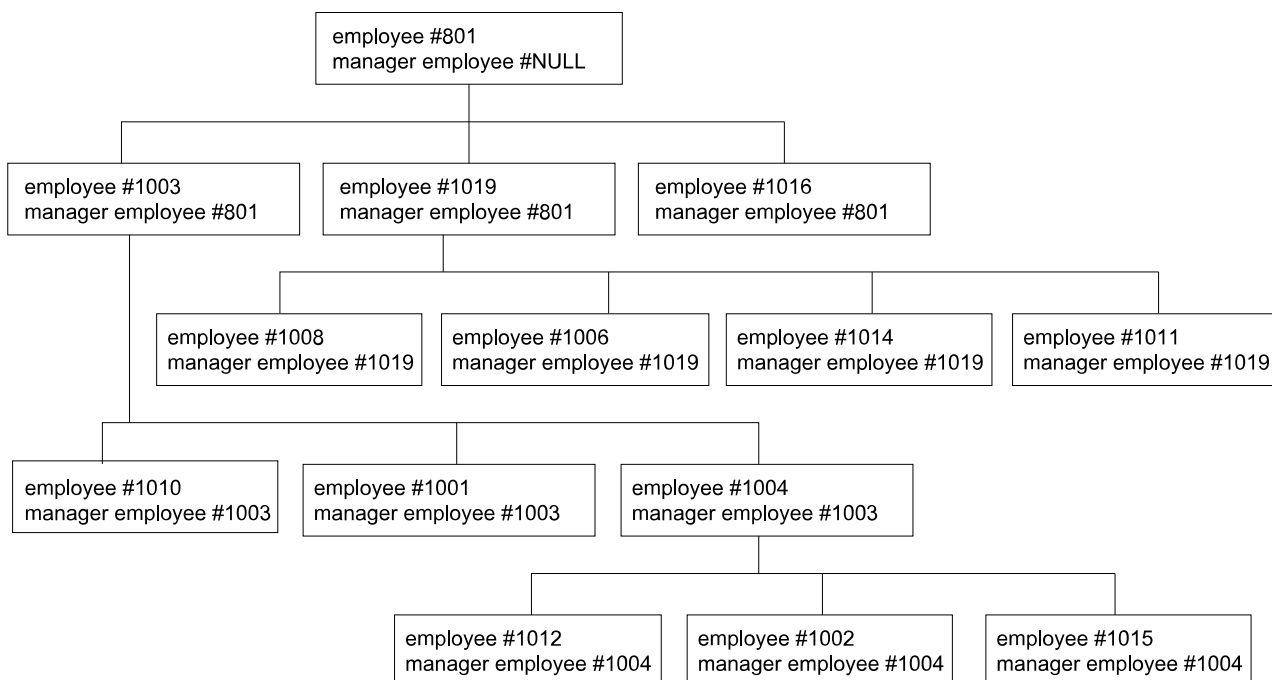
A recursive named query has three execution phases:

1. Create an initial nonrecursive, or seed, result set.
2. Recurse the intermediate result sets based on the seed result set until no new rows are added to the temporary named result set.
3. Execute a last query on the temporary named result set to return the final result set.

Consider this employee table:

```
CREATE TABLE employee (
  employee_number      INTEGER,
  manager_employee_number INTEGER,
  last_name            CHARACTER(20),
  first_name           VARCHAR(30));
```

The table represents an organizational structure of employee-manager relationships. The employee table is similar to this organization chart.



The following recursive query retrieves the employee numbers of all employees who directly or indirectly report to the manager who has an employee\_number value of 801:

```
WITH RECURSIVE temp_table (employee_number) AS
  (SELECT root.employee_number
   FROM employee AS root
   WHERE root.manager_employee_number = 801
  UNION ALL
   SELECT indirect.employee_number
   FROM temp_table AS direct, employee AS indirect
   WHERE direct.employee_number = indirect.manager_employee_number
  )
SELECT *
FROM temp_table
ORDER BY employee_number;
```

In the example, temp\_table is a temporary named result set that can be referred to in the FROM clause of the recursive statement.

The initial result set is established in temp\_table by the nonrecursive, or seed, statement and contains the employees that report directly to the manager with an employee\_number of 801:

```
SELECT root.employee_number
FROM employee AS root
WHERE root.manager_employee_number = 801
```

The recursion takes place by joining each employee in temp\_table with employees who report to the employees in temp\_table. The UNION ALL adds the results to temp\_table.

```
SELECT indirect.employee_number
FROM temp_table AS direct, employee AS indirect
WHERE direct.employee_number = indirect.manager_employee_number
```

Recursion stops when no new rows are added to temp\_table.

The final query is not part of the recursive WITH request modifier and extracts the employee information from temp\_table:

```
SELECT *
FROM temp_table
ORDER BY employee_number;
```

The results of the recursive query are as follows:

```
employee_number
-----
          1001
```

```

1002
1003
1004
1006
1008
1010
1011
1012
1014
1015
1016
1019

```

## WITH Modifiers

The rules and restrictions are:

- The only set operator that can appear in a recursive named query within a WITH modifier is UNION ALL.
- The following elements cannot appear within a WITH or WITH RECURSIVE modifier:
  - WITH or WITH RECURSIVE modifier
  - TOP *n* operator
  - User-defined functions
- The following elements cannot appear within a recursive statement in a WITH RECURSIVE modifier:
  - NOT IN or NOT EXISTS logical predicate
  - Aggregate functions
  - Ordered analytical functions
  - GROUP BY clause
  - HAVING clause
  - DISTINCT clause
  - Subqueries
  - Derived tables
- You cannot specify a WITH modifier in the definitions of any of these database objects:
  - Triggers
  - Stored procedures
- A recursive named query that does not have a recursive statement works like a nonrecursive named query.

This request produces the same results as the request that specifies a nonrecursive named query in the WITH modifier:

```
WITH RECURSIVE orderable_items (product_id, quantity) AS (
  SELECT stocked.product_id, stocked.quantity
  FROM stocked, product
  WHERE stocked.product_id = product.product_id
  AND   product.on_hand > 5)
  SELECT product_id, quantity
  FROM orderable_items
  WHERE quantity < 10;
```

## Rules and Restrictions for Embedded SQL

Teradata Database does not support recursive queries for these forms of embedded SQL:

- Static embedded SQL
- Dynamic embedded SQL

You cannot precede these statements with a WITH modifier:

- SELECT ... INTO
- DECLARE CURSOR

## Using a WITH Statement Result as Input to a Table Function in the FROM Clause

You can specify the temporary result set created by a WITH statement modifier subquery as input to a FROM clause table function. See [Example: CTE Result Set as Input to a Table Function](#).

## Depth Control to Avoid Infinite Recursion

If the data hierarchy is cyclic, or if the recursive query specifies an improper join condition, a recursive query can produce a request that never completes with a finite result.

In this context, a bad join is defined as a join that contains one or more of these errors.

- Joining incorrect columns.
- Selecting the wrong columns from the join.
- Specifying an OR operator instead of an AND operator with multiple join conditions.
- Specifying a join condition that is always true.

The following statement specifies an incorrect join condition in the recursive query. The join condition WHERE indirect.employee\_number IN (1003, 1004) is not correct because the result is always true.

```
WITH RECURSIVE temp_table (employee_id, level) AS (
  SELECT root.employee_number, 0 AS level
  FROM employee AS root
  WHERE root.employee_number = 1003
  UNION ALL
```

```

SELECT direct.employee_id, direct.level + 1 /* <--recursive statement*/
FROM temp_table AS direct, employee AS indirect
WHERE indirect.employee_number IN (1003,1004)
)
SELECT *
FROM temp_table
ORDER BY level;

```

The result set returned by this query is as follows:

employee_id	level
-----	-----
1003	0
1003	1
1003	1
1003	2
1003	2
1003	2
1003	2
1003	3
1003	3
1003	3
1003	3
1003	3
1003	3
1003	3
1003	3
...	...

and so on infinitely.

The best practice is to control the depth of the recursion as follows:

- Specify a depth control column in the column list of the recursive named query.
- Initialize the column value to 0 in the seed statement.
- Increment the column value by 1 in the recursive statement.
- Specify a limit for the value of the depth control column in the join condition of the recursive statements.

The following example adds a join condition (AND direct.level < 2) to the recursive named query in the WITH modifier of the previous query to limit the number of levels of recursion.

```

WITH RECURSIVE temp_table (employee_id, level) AS (
  SELECT root.employee_number, 0 AS level
  FROM employee AS root
  WHERE root.employee_number = 1003
  UNION ALL
  SELECT direct.employee_id, direct.level+1

```



```

        FROM temp_table AS direct, employee AS indir
        WHERE indir.employee_number IN (1003,1004)
        AND   direct.level < 2
    )
    SELECT *
    FROM temp_table
    ORDER BY level;

```

The data type of the numeric literal you specify for the initial value of the depth control column is the smallest data type that can contain the value. In the preceding query, the data type of the numeric literal 0 is BYTEINT because it is the smallest type that can fit the value 0.

The data type of the initial value of the depth control column limits the number of levels of recursion to the maximum value that the data type can represent.

For example, the maximum value of a BYTEINT is 127. If you need more than 127 levels of recursion, you must cast the numeric literal that you specify for the initial value of the depth control column to a larger type.

## Examples

### Example: Common Table Expression

A common table expression (CTE) in a WITH modifier can reference either preceding or subsequent CTEs defined in the WITH modifier, provided that the CTE does not indirectly reference itself. That is, circular references are not allowed. CTEs are also referred to as named queries.

Following is the table definition for this example.

```
CREATE TABLE orders (customer_id INTEGER, total_cost FLOAT);
```

These statements insert rows of data into the table.

```

INSERT INTO orders (43563, 734.12);
INSERT INTO orders (65758, 211.15);
INSERT INTO orders (23235, 1264.98);
INSERT INTO orders (43563, 583.23);
INSERT INTO orders (89786, 278.66);
INSERT INTO orders (13253, 401.97);
INSERT INTO orders (98765, 1042.23);
INSERT INTO orders (23235, 699.23);
INSERT INTO orders (43563, 935.35);
INSERT INTO orders (88354, 375.09);

```

This example of a WITH modifier includes a nonrecursive common table expression (CTE), specified as `multiple_order_totals`, that references the table `multiple_orders`, which is previously defined in the WITH clause.

```
WITH multiple_orders AS (
  SELECT customer_id, COUNT(*) AS order_count
  FROM orders
  GROUP BY customer_id
  HAVING COUNT(*) > 1
),
multiple_order_totals AS (
  SELECT customer_id, SUM(total_cost) AS total_spend
  FROM orders
  WHERE customer_id IN (SELECT customer_id FROM multiple_orders)
  GROUP BY customer_id
)
SELECT * FROM multiple_order_totals
ORDER BY total_spend DESC;
```

The query returns this answer set:

customer_id	total_spend
43563	2.25270000000000E 003
23235	1.96421000000000E 003

This example of a WITH modifier includes a nonrecursive common table expression (CTE), specified as `multiple_order_totals`, that references the table `multiple_orders`, which is subsequently defined in the WITH clause.

```
WITH multiple_order_totals AS (
  SELECT customer_id, SUM(total_cost) AS total_spend
  FROM orders
  WHERE customer_id IN (SELECT customer_id FROM multiple_orders)
  GROUP BY customer_id
),
multiple_orders AS (
  SELECT customer_id, COUNT(*) AS order_count
  FROM orders
  GROUP BY customer_id
  HAVING COUNT(*) > 1
)
SELECT * FROM multiple_order_totals
ORDER BY total_spend DESC;
```

The query returns this answer set:

customer_id	total_spend
43563	2.25270000000000E 003
23235	1.96421000000000E 003

## Example: Recursive Common Table Expressions in a WITH Modifier

If you mix nonrecursive queries with recursive queries in a WITH modifier, you must specify all forward CTE references or all backward CTE references. You cannot mix forward CTE references and backward CTE references.

Following is the table definition for this example.

```
CREATE TABLE t1(a1 INT, b1 INT);
```

These statements insert rows of data into the table.

```
INS t1(1,2);
INS t1(1,4);
INS t1(2,3);
INS t1(3,4);
```

This statement includes the recursive queries s3 and s4.

```
WITH
RECURSIVE s3 (MinVersion) AS (SELECT a1 FROM t1 WHERE a1 > 1
                                UNION ALL
                                SEL MinVersion FROM s3 WHERE MinVersion > 3),
RECURSIVE s4(MinVersion) AS (SELECT a1 FROM t1 WHERE a1 = 2
                                UNION ALL
                                SEL MinVersion FROM S4 WHERE MinVersion > 2)
SEL * FROM s3,s4;
```

The query returns this answer set:

MinVersion	MinVersion
3	2
2	2

## Example: Multiple Seed and Recursive Statements

This example shows a recursive query that uses multiple seed and recursive statements.

Consider these two tables:

```
CREATE TABLE planes (
  depart VARCHAR(40),
  arrive  VARCHAR(40),
  carrier VARCHAR(40),
  cost    DECIMAL(5,0));

CREATE TABLE trains (
  depart VARCHAR(40),
  arrive  VARCHAR(40),
  cost    DECIMAL(5,0));
```

The data in the planes table is as follows:

Depart	Arrive	Carrier	Cost
Paris	New York	AA	199
Paris	London	AA	99
London	New York	AA	199
New York	Mexico City	UA	99
Mexico City	New York	UA	99
Paris	Mexico City	AF	299
New York	London	AA	199
New York	Tokyo	JAL	999
Mexico City	Tokyo	JAL	999
Tokyo	New York	JAL	999

The data in the trains table is as follows:

Depart	Arrive	Cost
Paris	London	99
London	Paris	99
Paris	Milan	199

Depart	Arrive	Cost
London	Milan	199
Milan	Paris	199
Milan	Rome	49
Rome	Florence	49

The following query uses two seed statements and two recursive statements to return all cities reachable from Paris by train or plane.

```
WITH RECURSIVE temp_table (depart, arrive, carrier, depth) AS (
  SELECT p_root.depart, p_root.arrive, p_root.carrier, 0 AS depth
  FROM planes p_root
  WHERE p_root.depart = 'Paris'
  UNION ALL
  SELECT t_root.depart, t_root.arrive, 'EuroRail', 0 AS depth
  FROM trains t_root
  WHERE t_root.depart = 'Paris'
  UNION ALL
  SELECT direct.depart, indirect.arrive, indirect.carrier, direct.depth+1
  FROM temp_table AS direct, planes AS indirect
  WHERE direct.arrive = indirect.depart
  AND indirect.arrive <> 'Paris'
  AND direct.depth <= 4
  UNION ALL
  SELECT direct.depart, indirect.arrive, 'EuroRail', direct.depth+1
  FROM temp_table AS direct, trains AS indirect
  WHERE direct.arrive = indirect.depart
  AND indirect.arrive <> 'Paris'
  AND direct.depth <= 4)
SELECT DISTINCT arrive (TITLE 'Destinations Reachable From Paris')
FROM temp_table;
```

The result set for this recursive query is as follows:

```
Destinations Reachable From Paris
```

```
-----
```

```
Florence
London
Mexico City
Milan
New York
Rome
Tokyo
```

## Example: CTE Result Set as Input to a Table Function

Suppose you have created these tables and table function.

```

CREATE TABLE t1 (
    a1 INTEGER,
    b1 INTEGER);

CREATE TABLE t2 (
    a2 INTEGER,
    b2 INTEGER);

CREATE FUNCTION add2int (
    a INTEGER,
    b INTEGER)
RETURNS TABLE (addend1 INTEGER,
                addend2 INTEGER,
                mysum   INTEGER)
SPECIFIC add2int
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
NOT DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'CS!add3int!add2int.c';

```

Use the temporary result set derived from the subquery in the WITH statement modifier as input to table function add2int in the FROM clause.

```

WITH dt(a,b) AS (
    SELECT a1, b1
    FROM t1)
SELECT addend1, addend2, mysum
FROM dt, TABLE (add2int(dt.a, dt.b)) AS tf
ORDER BY 1,2,3;

```

## Example: Specifying a Dynamic UDT in a Recursive Query

The following example shows how you can use dynamic UDTs in a recursive query.

```

WITH MyDerived(u_sal) AS (
    SELECT NEW MP_STRUCTURED_INT(salary, '1', '1') AS u_sal
    FROM employee)
SELECT udf_aggr_avg_mp_struct(NEW VARIANT_TYPE(1 AS dummy,
                                                u_sal AS x) )
FROM MyDerived;
*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.

```

```
udf_aggr_avg_mp_struct(NEW VARIANT_TYPE (dummy, x))
-----
33438
```

## Examples: External UDFs

The following examples show ways to specify scalar or table external UDFs in a WITH modifier.

In this request, the input to the external UDF is recursive, and the UDF is joined with a recursive query.

```
WITH RECURSIVE dt(a,b,c,d) AS (
  SELECT a1, b1,a1-b1,0
  FROM t1
  UNION ALL
  SELECT addend1, addend2, mysum,d+1
  FROM dt,table (add2int(dt.a,dt.b)) AS tf
  WHERE d < 2
)
SELECT *
FROM dt;
```

In this statement, the input to the external UDF is not recursive, and the UDF is joined with a recursive query.

```
WITH RECURSIVE dt(a,b,c,d) AS (
  SELECT a1, b1,a1-b1,0
  FROM t1
  UNION ALL
  SELECT addend1, addend2, mysum,d+1
  FROM dt,table (add2int(t1.a1,t1.b1)) AS tf
  WHERE d < 2
)
SELECT *
FROM dt;
```

In this statement, the input to the external UDF is recursive, and the UDF is not joined with a recursive query.

```
WITH RECURSIVE dt(a,b,c,d) AS (
  SELECT a1, b1,a1-b1,0
  FROM t1
  UNION ALL
  SELECT addend1, r.b1, mysum, 1 AS d
  FROM table (add2int(dt.a,dt.b)) tf, t1 r
  WHERE d < 1
```

```

    AND    tf.addend1=t1.a1
  )
  SELECT *
  FROM dt;

```

In this statement, the input to the external UDF is not recursive, and the UDF is not joined with a recursive query.

```

WITH dt(a,b,c) AS (
  SELECT a1, b1 ,a1-b1
  FROM t1
  UNION ALL
  SELECT addend1, addend2, mysum
  FROM table (add2int(t1.a1, t1.b1)) tf
)
  SELECT *
  FROM dt;

```

### Example: Invoking an SQL UDF in a Recursive Query

This example invokes the SQL UDF *value\_expression* in the WHERE clause of the recursive query.

```

WITH RECURSIVE temp_table (employee_number) AS (
  SELECT root.employee_number
  FROM employee AS root
  WHERE root.manager_employee_number = 801
  AND    test.value_expression(dept_no, 0) = 25;
  UNION ALL
  SELECT indirect.employee_number
  FROM temp_table AS direct, employee AS indirect
  WHERE direct.employee_number = indirect.manager_employee_number
  AND    test.value_expression(2,3) = 5
)
  SELECT *
  FROM temp_table
  ORDER BY employee_number;

```

### Example: Invoking an SQL UDF in a Recursive Query

This example invokes the SQL UDF *value\_expression* in the WHERE clause of the recursive query.

```

WITH RECURSIVE temp_table (employee_number) AS (
  SELECT root.employee_number
  FROM employee AS root
  WHERE root.manager_employee_number = 801

```



```

    AND    test.value_expression(dept_no, 0) = 25;
UNION ALL
  SELECT indirect.employee_number
  FROM temp_table AS direct, employee AS indirect
  WHERE direct.employee_number = indirect.manager_employee_number
  AND    test.value_expression(2,3) = 5
  )
SELECT *
FROM temp_table
ORDER BY employee_number;

```

## Related Topics

For more information on recursive queries and views, see:

- “Recursive Queries” in *Teradata Vantage™ SQL Fundamentals*, B035-1141.
- “CREATE RECURSIVE VIEW” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## WITH DELETED ROWS

Include deleted rows in the query processing for a single load isolated table.

### Required Privileges

In addition to the SELECT privilege, you must have the DROP TABLE privilege to execute this special read operation on the load isolated table.

If the table is associated with row-level security (RLS) constraints, you must have appropriate row-level security. RLS checks are applied on all rows, including logically deleted rows.

### ANSI Compliance

The WITH DELETED ROWS clause is a Teradata extension to the ANSI SQL:2011 standard.

### Locking Modifiers and the WITH DELETED ROWS Option

A locking modifier, if specified, is ignored when you specify the WITH DELETED ROWS option.

### Join Indexes and the WITH DELETED ROWS Option

You cannot include join indexes in the SELECT statement when you specify the WITH DELETED ROWS option.

### Example: Obtain the Total Number of Rows in the Table

```
SELECT WITH DELETED ROWS COUNT(*) FROM ldi_table;
```

**Example: Obtain the Number of Logically Deleted Rows**

For information on the TD\_ROWLOADID expression, see [Inserting into Load Isolated Tables](#).

```
SELECT WITH DELETED ROWS COUNT(*) FROM ldi_table
WHERE TD_ROWLOADID > '100000000'xi8;
```

**Example: Obtain the Number of Logically Deleted Rows From a Load Operation**

For information on the TD\_ROWLOADID expression, see [Inserting into Load Isolated Tables](#).

```
SELECT WITH DELETED ROWS COUNT(*) FROM ldi_table
WHERE (TD_ROWLOADID/'100000000'xi8)(int) = load_id_value;
```

## AS JSON

Composes data from table columns into a text format JSON document.

The SELECT statement returns a single JSON column named "JSON".

## Usage Notes

### Rules for Using SELECT AS JSON

- The SELECT AS JSON statement composes column values into a text format JSON document. Binary formats such as BSON or UBJSON are not supported.
- You cannot specify SELECT AND CONSUME together with SELECT AS JSON to compose JSON documents.
- If you specify TOP n or ORDER BY in the SELECT statement, you must explicitly specify the sorted fields because you cannot sort by the JSON column.

## Examples

### Examples: SELECT AS JSON

Following is the table definition for these examples.

```
CREATE TABLE MyTable (
a INTEGER,
b INTEGER,
j JSON AUTO COLUMN);
```

These statements insert data into the table.

```
INSERT INTO MyTable JSON '{"a":10,"b":1234,"extra":"1234"}';
```

```
INSERT INTO MyTable JSON '{"a":2345,"b":11,"extra":"2222"}';
```

This SELECT statement returns all of the columns for MyTable, ordered by the first column.

```
SELECT * FROM MyTable ORDER BY 1;
```

a	b	j
10	1234	Snippet:{"extra":"1234"}
2345	11	Snippet:{"extra":"2222"}

This statement includes the AS JSON option to return the data from columns a, b, and j in JSON format.

```
SELECT AS JSON a, b, j FROM MyTable;
```

The output is a single JSON column titled "JSON".

JSON
Snippet:{"a":2345,"b":11,"j":{"extra":"2222"}}
Snippet:{"a":10,"b":1234,"j":{"extra":"1234"}}

For SELECT AS JSON with the ORDER BY option, you must specify the column by name, not position.

```
SELECT AS JSON a, b FROM MyTable ORDER BY a ASC;
```

JSON
Snippet:{"a":2345,"b":11,"j":{"extra":"2222"}}
Snippet:{"a":10,"b":1234,"j":{"extra":"1234"}}

For SELECT AS JSON with the TOP *n* option, you must specify the ORDER BY column by name, not position.

```
SELECT AS JSON TOP 2 a, b FROM MyTable ORDER BY a DESC;
```

JSON
Snippet:{"a":2345,"b":11}
Snippet:{"a":10,"b":1234}

## Distinct

DISTINCT specifies that duplicate values are not to be returned when an expression is processed.

### Syntax Elements

#### DISTINCT

Only one row is to be returned from any set of duplicates that might result from a given expression list. Rows are duplicates only if each value in one is equal to the corresponding value in the other.

If you specify DISTINCT and an *expand\_alias* in the select list, Teradata Database performs the DISTINCT operation after the EXPAND operation to ensure that duplicate rows are removed from the expanded result.

If you specify DISTINCT but do not specify an *expand\_alias* in the select list, the system ignores the EXPAND ON clause (even if you specify one), and does not expand rows.

You cannot specify DISTINCT if any member of the select column list is a LOB column.

### ANSI Compliance

DISTINCT is ANSI SQL:2011-compliant.

## Usage Notes

### DISTINCT and GROUP BY

For cases when the DISTINCT is semantically equivalent to GROUP BY, the optimizer makes a cost-based decision to eliminate duplicates either by way of a combined sort and duplicate elimination or an aggregation step.

### DISTINCT Operator and UDTs

If you specify the DISTINCT operator and the select list specifies a UDT column, Teradata Database applies the ordering functionality of that UDT to achieve distinctness.

#### NOTICE

Depending on the ordering definition, the same query can return different results, similar to how the results of performing case-insensitive DISTINCT processing on character strings can vary. Issuing such a query twice can return different results.

For a UDT example, consider a structured UDT named *CircleUdt* composed of these attributes:

- *x* INTEGER
- *y* INTEGER
- *r* INTEGER

Suppose that the external type for *CircleUdt* is a character string containing the ASCII form of the *x*, *y*, and *r* components. The ordering functionality is defined to map and compare the *r* components of two instance of *CircleUdt*.

These circles are considered to be equal because their *r* values are both 9:

- NEW CircleUdt('1,1,9'), where *x*=1, *y*=1, and *r*=9
- NEW CircleUdt('6,10,9')

The system sometimes returns the '1,1,9' result and at other times returns the '6,10,9' result as part of the DISTINCT result set.

## DISTINCT Operator and Large Objects

You cannot include LOB columns in the select list of a SELECT request if you also specify DISTINCT.

You *can* specify LOB columns with DISTINCT if you CAST them to an appropriate data type, as documented in this table:

IF the LOB column has this data type ...	THEN you can reference it with DISTINCT if you CAST it to this data type ...
BLOB	<ul style="list-style-type: none"> <li>• BYTE</li> <li>• VARBYTE</li> </ul>
CLOB	<ul style="list-style-type: none"> <li>• CHARACTER</li> <li>• VARCHAR</li> </ul>

## SQL Elements That Cannot Be Used With a DISTINCT Operator

The following SQL elements cannot be specified with a request that also specifies a DISTINCT operator.

- WITH request modifier
- TOP *n* operator
- The recursive statement of a recursive query

However, you can specify DISTINCT within a nonrecursive seed statement in a recursive query.

## Unexpected Row Length Errors With the DISTINCT Operator

Before performing the sort operation required to eliminate duplicates, Teradata Database creates a sort key and appends it to the rows to be sorted. If the length of this temporary data structure exceeds the system row length limit of 64 KB, the operation fails and the system returns an error. Depending on the situation, the error message text states that:

- A data row is too long.
- Maximum row length exceeded in *database\_object\_name*.

For explanations of these messages, see *Teradata Vantage™ - Database Messages*, B035-1096.

The BYNET only looks at the first 4,096 bytes of the sort key created to sort the specified fields, so if the column the sort key is based on is greater than 4,096 bytes, the key is truncated and rows in the set can either be identified as duplicates when they are not or identified as unique when they are duplicates.

## Examples

### Example: Simple Uses of DISTINCT

The following statement returns the number of unique job titles.

```
SELECT COUNT(DISTINCT JobTitle)
FROM ...
```

The following statement lists the unique department numbers in the *employee* table.

```
SELECT DISTINCT dept_no
FROM employee;
```

The result returned is:

```
dept_no
-----
    100
    300
    500
    600
    700
```

### Example: Using DISTINCT With Aggregates

The following statement uses the DISTINCT operator to eliminate duplicate values before applying SUM and AVG aggregate operators, respectively, to column *x*:

```
SELECT SUM(DISTINCT x), AVG(DISTINCT x)
FROM test_table;
```

You can only perform DISTINCT aggregations at the first, or highest level of aggregation if you specify subqueries.

Note that the result data type for a SUM operation on a DECIMAL(*m*,*n*) column is DECIMAL(*p*,*n*), as described in this table:

IF DBS Control field MaxDecimal is set to ...	AND ...	THEN <i>p</i> is ...
<ul style="list-style-type: none"> <li>• 0</li> <li>• 15</li> </ul>	$m \leq 15$	15
	$15 < m \leq 18$	18
	$m > 18$	38

IF DBS Control field MaxDecimal is set to ...	AND ...	THEN <i>p</i> is ...
18	$m \leq 18$	18
	$m > 18$	38
31	$m \leq 31$	31
	$m > 31$	38
38	$m = \text{any value}$	38

## ALL

ALL specifies that duplicate values are to be returned when an expression is processed.

ALL is the default option, except in query expressions using set operators, where ALL is an option, but not the default. For more information, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

### Syntax Elements

#### ALL

All rows, including duplicates, are to be returned in the results of the expression list.

This is the default value.

### ANSI Compliance

ALL is ANSI SQL:2011-compliant.

### Example: Using ALL

In contrast to the second statement in [Example: Simple Uses of DISTINCT](#), this statement returns the department number for every employee:

```
SELECT ALL dept_no
FROM employee;
```

## .ALL Operator

For structured UDTs only, .ALL specifies that the individual attribute values for a specified table or column name are to be returned.

### Syntax Elements

#### *table\_name*

Name of a table for which all the attributes of all its structured UDT columns are to be returned.

**column\_name**

Name of a structured UDT column for which all its attributes are to be returned.

**.ALL**

All attributes for the specified UDT column or for all structured UDT columns in the specified table are to be returned in the results of the expression list.

**ANSI Compliance**

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

**.ALL Operator and Structured UDTs**

If a column has a structured UDT data type, then the values of its individual attributes are returned when you specify the .ALL option.

If you specify the .ALL operator using either of these syntaxes, but there is no structured data type column in the specified table set, Teradata Database disregards the specification:

- \*.ALL
- table\_name.\*.ALL

If you specify this syntax, but the specified column does not have a structured UDT type, the specification is disregarded:

```
column_name.ALL
```

**Example: Using the .ALL Operator With Structured UDTs**

The following UDT and table definitions are for the \*.ALL operator examples below.

```
CREATE TYPE school_record AS (
  school_name VARCHAR(20),
  GPA          FLOAT)
INSTANTIABLE
...
CREATE TYPE college_record AS (
  school school_record,
  major  VARCHAR(20),
  minor  VARCHAR(20))
INSTANTIABLE
...
CREATE TABLE student_record (
  student_id  INTEGER,
  Last_name   VARCHAR(20),
  First_name  VARCHAR(20),
  high_school school_record,
  college     college_record);
```



**Example 1a: Using \*.ALL to Retrieve All Table Columns, Expanding the Structured Types**

This example shows a pair of SELECT statements that return equivalent response sets. The only difference is that one statement uses the \*.ALL notation and the other does not.

Use \*.ALL to retrieve all columns of the table *student\_record* where the structured type columns *high\_school* and *college* are expanded:

```
SELECT *.ALL
FROM student_record;
```

The following equivalent SELECT statement does not use the \*.ALL notation:

```
SELECT student_id, last_name, first_name,
       high_school.school_name(), high_school.GPA(),
       college.school().school_name(), college.school().GPA(),
       college.major(), college.minor()
FROM student_record;
```

**Example 1b: Using \*.ALL to Retrieve the Expanded Structured Type for a Single UDT Column**

Use \*.ALL to retrieve all columns of the table *student\_record* where the structured type columns *high\_school* and *college* are expanded:

```
SELECT student_record.*.ALL;
```

The following equivalent SELECT statement does not use the \*.ALL notation:

```
SELECT student_record.student_id, student_record.last_name,
       student_record.first_name,
       student_record.high_school.school_name(),
       student_record.high_school.GPA(),
       student_record.college.school().school_name(),
       student_record.college.school().GPA(),
       student_record.college.major(),
       student_record.college.minor();
```

**Example 1c: Using \*.ALL to Retrieve the Expanded Structure Type for Multiple UDT Columns**

Retrieve high school names and grade point averages of all students:

```
SELECT high_school.ALL
FROM student_record;
```

The following equivalent SELECT statement does not use the .ALL notation:

```
SELECT high_school.school_name(), high_school.GPA()
FROM student_record;
```

**Example 1d: Selecting the Information For Only One Student From a UDT**

Retrieve the college name, GPA, major, and minor of student 'Steven Smith':

```
SELECT s.college.ALL
FROM student_record s
WHERE s.student.First_name() = 'Steven'
AND s.student.Last_name() = 'Smith';
```

The following equivalent SELECT statement does not use the .ALL notation:

```
SELECT s.college.school().school_name(), s.college.school().GPA(),
       s.college.major(), s.college.minor()
FROM student_record s
WHERE s.student.first_name() = 'Steven'
AND s.student.last_name() = 'Smith';
```

## NORMALIZE

Period values in the first period column that meet or overlap are combined to form a period that encompasses the individual period values.

### Syntax Elements

#### NORMALIZE

The result of the select is normalized on the first period column in the select list. Period values that meet or overlap are coalesced, that is, combined to form a period that encompasses the individual period values.

Any period columns that are specified after the first period column are treated as regular column values. Normalize is the last operation performed on the result of a SELECT statement. You can use a SELECT statement with the normalize clause on a normalized or non-normalized table.

A SELECT statement cannot normalize a derived period column.

#### ON MEETS OR OVERLAPS

Period values that meet or overlap are to be coalesced, that is, combined to form a period that encompasses the individual period values.

#### ON OVERLAPS

Period values that overlap are to be coalesced, that is, combined to form a period that encompasses the individual period values.

#### ON OVERLAPS OR MEETS

Period values that overlap or meet are to be coalesced, that is, combined to form a period that encompasses the individual period values.

#### *table\_name*

Name of a table for which all the attributes of all its structured UDT columns are to be returned.

#### *column\_name*

Name of a column in the named query definition.

You can specify a row-level table constraint column in the select list of a SELECT statement, as long as it is not part of an arithmetic expression. The value returned for the column is the coded value for the row-level security constraint from the row.

Columns with a UDT type are valid with some exceptions. See [Specifying UDTs in an SQL Request](#).

You cannot specify LOB columns with NORMALIZE.

## ANSI Compliance

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

## Usage Notes

### Using SELECT with NORMALIZE

Following are the rules and restrictions for using SELECT with NORMALIZE. For information about temporal tables, see *Teradata Vantage™ Temporal Table Support*, B035-1182.

The NORMALIZE operation is performed on the first period column in the select list.

For NORMALIZE, at least one column in the select list must be of period data type.

You can use NORMALIZE:

- on normalized or non-normalized tables
- in subqueries
- in the SELECT INTO statement

When the first period value specified in the SELECT list is a USING value, a constant value, or DEFAULT function on the period column, NORMALIZE operates like SELECT with DISTINCT.

When the NORMALIZE clause is specified with EXPAND, the rows are expanded and then the result is normalized.

When a query includes NORMALIZE and ORDER BY, the rows are normalized first and the result is ordered on the normalized result.

NORMALIZE is the last operation performed, except when the query includes ORDER BY or INTO. The other operations in a query are performed before NORMALIZE, except ORDER BY and INTO, and then the rows are normalized. If NORMALIZE is specified in a query with aggregation or OLAP, normalize is performed on the final result after the aggregate or OLAP operation.

When SELECT NORMALIZE includes a view list, views are not updatable.

A SELECT statement on a derived period column cannot include NORMALIZE.

You cannot include LOB columns in the select column list for NORMALIZE.

You cannot use NORMALIZE in a SELECT statement that includes the WITH clause or the TOP n operator.

You can use NORMALIZE in a non-recursive seed statement in a recursive query. However, NORMALIZE is not allowed in a recursive statement of a recursive query.

When NORMALIZE is specified in the CURSOR select statement, the CURSOR is not updatable.

## INSERT... SELECT and NORMALIZE

These restrictions also apply to an INSERT... SELECT statement included in a CREATE TRIGGER, REPLACE TRIGGER, CREATE MACRO, REPLACE MACRO, CREATE PROCEDURE, or REPLACE PROCEDURE statement.

## LOCAL ORDER BY and NORMALIZE

You cannot specify the NORMALIZE option as part of SELECT in an INSERT... SELECT statement that includes the LOCAL ORDER BY option.

## NOPI Target Table and NORMALIZE

You cannot specify the NORMALIZE option as part of SELECT in an INSERT... SELECT statement where the target table is defined with NO PRIMARY INDEX.

## NORMALIZE in Target Table Definition

You cannot specify the LOCAL ORDER BY option in an INSERT... SELECT statement where the target table definition includes the NORMALIZE option.

You cannot specify the HASH BY option in an INSERT... SELECT statement where target table definition includes the NORMALIZE option.

## Examples

### Example: Using NORMALIZE

Following is the table definition for the NORMALIZE examples:

```
CREATE TABLE project
(
  emp_id      INTEGER,
  project_name VARCHAR(20),
  dept_id     INTEGER,
  duration    PERIOD(DATE)
);
```

The table contains the following rows:

Emp_ID	Project_Name	Dept_ID	Duration
10	First Phase	1000	10 Jan 2010 - 20 Mar 2010

Emp_ID	Project_Name	Dept_ID	Duration
10	First Phase	2000	20 Mar 2010 - 15 July 2010
10	Second Phase	2000	15 June 2010 - 18 Aug 2010
20	First Phase	2000	10 Mar 2010 - 20 July 2010

The following select statement performs a normalize operation on *emp\_id*. Note that the select list contains only one period column.

```
SELECT NORMALIZE ON MEETS OR OVERLAPS emp_id, duration
FROM project;
```

The query returns the following result:

Emp_ID	Duration
10	10 Jan 2010 - 18 Aug 2010
20	10 Mar 2010 - 20 July 2010

The following select statement performs a normalize operation on *project\_name*. Note that the select list contains only one period column.

```
SELECT NORMALIZE project_name, duration
FROM project;
```

The query returns the following result:

Project_Name	Duration
First Phase	10 Jan 2010 - 20 July 2010
Second Phase	15 June 2010 - 18 Aug 2010

The following select statement performs a normalize operation on *project\_name* and *dept\_id*. Note that the select list contains only one period column.

```
SELECT NORMALIZE project_name, dept_id, duration
FROM project;
```

The query returns the following result:

Project_Name	Dept_ID	Duration
First Phase	1000	10 Jan 2010 - 20 Mar 2010
First Phase	2000	20 Mar 2010 - 20 July 2010

Project_Name	Dept_ID	Duration
Second Phase	2000	15 June 2010 - 18 Aug 2010

## TOP *n*

Specifies that either an explicit number of rows or an explicit percentage of rows is to be returned from the query result set.

### Syntax Elements

#### TOP *n*

The query returns a specified number of rows or a percentage of available rows.

You cannot specify NORMALIZE with TOP *n*.

#### *integer*

#### *integer* PERCENT

A nonzero, positive INTEGER literal indicating the number of rows to return.

If the PERCENT option is specified, the system returns *integer*% of the total rows available, where  $100 \geq \textit{integer} > 0$ .

#### *decimal*

#### *decimal* PERCENT

A nonzero, positive DECIMAL literal indicating the number of rows to return.

If the PERCENT option is specified, the system return *decimal*% of the total rows available, where  $100 \geq \textit{decimal} > 0$ .

If the PERCENT option is not specified, decimal cannot include a decimal point.

#### PERCENT

The *integer* or *decimal* value indicates a percentage of the rows to return.

The number of rows returned is  $(n/100 * \text{result set row count})$ , where *n* is the *integer* or *decimal* value.

If the number is not an integer, and the fractional part of the number  $\geq 0.0000000000000001$ , the next highest integer is used.

For fractions  $< 0.0000000000000001$ , the number is truncated.

The TOP *n* operator is referred to as TOP *m* when you specify the PERCENT option for clarification purposes.

#### WITH TIES

Rows returned by the query include the specified number or percentage of rows in the ordered set produced by the ORDER BY clause, plus any additional rows where the value of the sort key is the same as the value of the sort key in the last row that satisfies the specified number or percentage of rows.

The WITH TIES option is ignored if the SELECT statement does not also specify an ORDER BY clause.

## ANSI Compliance

TOP *n* is a Teradata extension to the ANSI SQL:2011 standard.

Other SQL dialects use similar operators with names such as:

- FIRST *n*
- LIMIT *n*
- SET ROWCOUNT *n*
- STOP AFTER *n*

## Usage Notes

### What the TOP *n* Operator Does

You can use the TOP *n* operator in these ways:

- The TOP *n* operator returns only a subset of the rows in the query result set. For example, this query returns 10 rows from the *orders* table:

```
SELECT TOP 10 *
FROM orders;
```

- To obtain a subset of the data from an ordered set, specify the TOP *n* operator with an ORDER BY clause. For example, this query returns the last five orders shipped:

```
SELECT TOP 5 *
FROM orders
ORDER BY ship_date DESC;
```

- If the TOP *n* operator specifies a number greater than the number of rows in the query result set, then the query returns all of the rows in the query result set without returning an error.
- As the size of *n* increases, the performance of TOP *n* gradually degrades.
- You can use parameters to pass values for *n* from:
  - CREATE MACRO, see *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.
  - CREATE PROCEDURE (SQL Form), see *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.
  - USING Request Modifier, see [USING Request Modifier](#).

### Rules and Restrictions for the TOP *n* Operator

You cannot specify the TOP *n* operator in any of these SQL statements or statement components:

- Correlated subquery
- Subquery in a search condition
- CREATE JOIN INDEX
- CREATE HASH INDEX
- Seed statement or recursive statement in a CREATE RECURSIVE VIEW statement or WITH RECURSIVE statement modifier
- Subselects of set operations.

You cannot specify these options in a SELECT statement that specifies the TOP *n* operator:

- DISTINCT option
- QUALIFY clause
- SAMPLE clause
- WITH clause

This restriction refers to the WITH clause you can specify for summary lines and breaks. See [WITH Clause](#). The nonrecursive WITH statement modifier that can precede the SELECT keyword can be included in statements that also specify the TOP *n* operator. See [WITH Modifier](#).

- ORDER BY clause where the sort expression is an ordered analytical function.

You cannot specify the *n* value of a TOP *n* specification as a USING parameter for iterated array processing. For an example, see [Example: Non-Support for Iterated Requests With TOP n](#).

## Evaluation Order of TOP *n* in a SELECT

The system evaluates the TOP *n* operator after all other clauses in the SELECT statement have been evaluated.

## TOP *n* Operator Out Performs QUALIFY RANK and QUALIFY ROW\_NUMBER

The QUALIFY clause with the RANK or ROW\_NUMBER ordered analytical functions returns the same results as the TOP *n* operator.

For best performance, use the TOP option instead of the QUALIFY clause with RANK or ROW\_NUMBER. In best case scenarios, the TOP *n* operator provides better performance; in worst case scenarios, the TOP *n* operator provides equivalent performance.

For example, these two statements have the same semantics, but the statement that specifies TOP *n* performs better than the statement that specifies QUALIFY ROW\_NUMBER.

```
SELECT TOP 10 *
FROM sales
ORDER BY county;
```



```
SELECT *
FROM sales
QUALIFY ROW_NUMBER() OVER (ORDER BY COUNTY) <= 10;
```

Similarly, these two statements have the same semantics, but the statement that specifies TOP  $n$  performs better than the statement that specifies QUALIFY RANK.

```
SELECT TOP 10 WITH TIES *
FROM sales ORDER BY county;

SELECT *
FROM sales
QUALIFY RANK() OVER (ORDER BY county) <= 10;
```

## Examples

### Example: Comparing the Results of Specifying TOP $n$ \* and TOP $n$ WITH TIES \*

Consider this data in the orders table:

```
SELECT *
FROM orders;
order_date  customer    product      quantity
-----
04/05/10    Bestway     JR-0101      10
04/04/28    Bestway     SW-0022      25
04/05/10    Bestway     QR-7737      10
04/04/28    Samstone    JR-0101      35
04/05/10    Bestway     SW-0023      10
04/04/28    Samstone    KB-6883      20
04/05/10    Finelity    JR-0101      12
04/04/28    Samstone    SW-0023      12
04/05/10    Finelity    SW-0021      24
04/05/10    Finelity    KB-8883      24
```

The following statement selects the top three orders with the largest quantities:

```
SELECT TOP 3 *
FROM orders
ORDER BY quantity DESC;
order_date  customer    product      quantity
-----
04/04/28    Samstone    JR-0101      35
```

04/04/28	Bestway	SW-0022	25
04/05/10	Finelity	SW-0021	24

To include any orders with the same quantity as the third-largest, use the WITH TIES option:

```
SELECT TOP 3 WITH TIES *
FROM orders
ORDER BY quantity DESC;
order_date  customer      product          quantity
-----
04/04/28    Samstone        JR-0101          35
04/04/28    Bestway         SW-0022          25
04/05/10    Finelity        SW-0021          24
04/05/10    Finelity        KB-8883          24
```

## Related Topics

For more information about options and functions related to the TOP *n* operator, see:

- [QUALIFY Clause](#)
- “RANK” and “ROW\_NUMBER” in *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145

## FROM Clause

### Purpose

Defines either of these items:

- The set of base tables, global temporary tables, volatile tables, derived tables, views, or table functions that are referenced by the SELECT statement or query.
- A correlation name for a base table, global temporary table, volatile table, derived table, or view for a self-join operation. See [Self-Join](#).

For information about syntax that is compatible with temporal tables, see *Teradata Vantage™ ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ Temporal Table Support*, B035-1182.

## Syntax Elements

### FROM Clause

#### FROM

Keyword preceding the names of one or more tables, queue tables, views, or derived tables from which *expression* is to be derived.

If the **TABLE** option is specified, the **FROM** clause invokes the specified table user-defined function that returns row results as a derived table.

## Single Table

A **FROM** clause can include a sequence of single table references.

This action creates an implicit inner join, instead of explicit joins where the keyword **JOIN** is part of the syntax.

### ***table\_name***

Name of a base table, queue table, global temporary table, volatile table, derived table, or view. If the database is omitted, the system infers it from context.

### **AS**

Optional introduction to *correlation\_name*.

### ***correlation\_name***

Optional alias for the table that is referenced by *table\_name*.

You must specify a *correlation\_name* for a self-join.

ANSI SQL refers to table aliases as correlation names. Correlation names are also referred to as range variables.

## Joined Tables

These options enable the **FROM** clause to specify that multiple tables be joined.

### ***joined\_table***

Name of a joined table, which can be either a single table name with optional alias name, or a joined table, indicating nested joins.

### **INNER**

Join in which qualifying rows from one table are combined with qualifying rows from another table according to some join condition.

Inner join is the default join type.

### **OUTER**

Join in which qualifying rows from one table that do not have matches in the other table, are included in the join result along with the matching rows from the inner join. The rows from the first table are extended with nulls.

### **LEFT OUTER**

Outer join with the table that was listed first in the **FROM** clause.

In a **LEFT OUTER JOIN**, the rows from the left table that are not returned in the result of the inner join of the two tables are returned in the outer join result and extended with nulls.

**RIGHT OUTER**

Outer join with the table that was listed second in the FROM clause.

In a RIGHT OUTER JOIN, the rows from the right table that are not returned in the result of the inner join of the two tables are returned in the outer join result and extended with nulls.

**FULL OUTER**

Rows are returned from both tables

In a FULL OUTER JOIN, rows from both tables that have not been returned in the result of the inner join will be returned in the outer join result and extended with nulls.

**JOIN**

Introduction to the name of the second table to include in the join.

**ON *search\_condition***

One or more conditional expressions, including scalar subqueries, that must be satisfied by the result rows.

You can only specify a scalar UDF for *search\_condition* if it is invoked within an expression and returns a value expression.

If you specify the value for a row-level security constraint in a search condition, it must be expressed in its encoded form.

An ON condition clause is required if the FROM clause specifies an outer join.

**CROSS JOIN**

Unconstrained, or Cartesian, join. Returns all rows from all tables specified in the FROM clause. Two joined tables can be cross joined.

***single\_table***

Name of a single base or derived table or view on a single table to be cross joined with *joined\_table*.

**ANSI Compliance**

The FROM clause is ANSI SQL:2011-compliant with extensions.

A FROM clause is mandatory for SELECT statements in the ANSI SQL:2011 standard. The optional FROM clause in Teradata SQL is a Teradata extension.

The ANSI SQL:2011 standard also requires that all of the columns referenced in an outer query be in tables that are specified in the FROM clause, but Teradata SQL does not.

## Usage Notes

### Fully-qualified Names and the FROM Clause

If the FROM clause refers to a table or view name that is duplicated in other, non-default, databases referenced in the query, you must use the fully-qualified form of the name, such as one of the following:

```
database_name.table_name
user_name.table_name
```

### Subqueries and the FROM Clause

If a column reference is made from an inner query to an outer query, then it must be fully qualified with the appropriate table name from the FROM clause of the outer query.

Any subquery must contain a FROM clause, and every column referenced in the SELECT statement must be in a table referenced either in the FROM clause for the subquery or some FROM clause in an outer query containing the subquery.

SQL Statement Main Query	FROM Clause Required
<ul style="list-style-type: none"> <li>• ABORT</li> <li>• DELETE</li> <li>• SELECT</li> <li>• UPDATE</li> </ul>	No.
<ul style="list-style-type: none"> <li>• ABORT</li> <li>• SELECT</li> <li>• UPDATE</li> </ul>	Optional.
DELETE	Optional, depending on the syntax used.

### Self-join Operations and the FROM Clause

During a self-join operation, related data selected from different rows of the same table is combined and returned as a single row. The temporary table names that are defined using the FROM clause qualify different references to the same table columns.

The following table lists the effects of column references on self-join processing. These rules apply only to cases where a single aliasing FROM clause is supplied.

Columns Referenced	Type of Reference	Self-join
All	Unqualified (the preceding table name or correlation name is omitted)	Not performed.

Columns Referenced	Type of Reference	Self-join
	Qualified (but the qualifiers reference only the correlation name)	
Some, but not all.	Qualified and the qualifiers reference only the correlation name	
	Qualified and the qualifiers reference only the permanent table name	Moot because no assumptions can be made about the owner of the unqualified columns. The self-join not performed.

When a FROM clause references both a permanent table name and a correlation name, a self-join can be performed depending on how the column references are qualified.

For a successful self-join operation, column references must be fully qualified (that is, specified in the form *table\_name.column\_name*) and the qualifiers should involve both the permanent and the correlation table names.

After a correlation name is declared, any subsequent reference to the base table name causes a new instance of the table to be used. The result of the SELECT can be a Cartesian self-join.

See [Example: FROM Clause Used for a Self-Join](#) where *table\_1* is given the correlation name *t*. The subsequent specification of *table\_1* in the WHERE clause then causes the statement to perform a Cartesian product.

```
SELECT *
FROM table_1 AS t
WHERE table_1.column_1 = 2;
```

## Examples

### Example: FROM Clause Used for a Self-Join

The following statement returns employees who have more years of work experience than their department managers:

```
SELECT workers.name, workers.yrs_exp, workers.dept_no,
       managers.name, managers.yrs_exp
FROM employee AS workers, employee AS managers
WHERE managers.dept_no = workers.dept_no
AND    UPPER (managers.jobtitle) IN ('MANAGER' OR 'VICE PRES')
AND    workers.yrs_exp > managers.yrs_exp;
```

The FROM clause in the preceding statement enables the employee table to be processed as though it were two identical tables: one named *workers* and the other named *managers*.

As in a normal join operation, the WHERE clause defines the conditions of the join, establishing dept\_no as the column whose values are common to both tables.

The statement is processed by first selecting managers rows that contain a value of either 'MANAGER' or 'VICE PRES' in the jobtitle column. These rows are then joined to the workers rows using a merge join operation with this join condition:

- A workers row must contain a dept\_no value that matches the dept\_no value in a managers row.
- The matching workers row must also contain a yrsexp value that is greater than the yrs\_exp value in the managers row.

The following result is returned:

name ----	yrsexp -----	dept_no -----	name ----	yrsexp -----
Greene W	15	100	Jones M	13
Carter J	20	200	Watson L	8
Aguilar J	11	600	Regan R	10
Leidner P	13	300	Phan A	12
Ressel S	25	300	Phan A	12

## Example: FROM Clause Left Outer Join

The following example illustrates a left outer join. See [Outer Joins](#). In the example, the skills table lists various skills and the associated codes, and the emp table lists employee numbers and a skills codes.

skills		emp		
skill_no	skill_name		emp_no	skill_no
1	baker		6123	1
2	doctor		6234	1
3	farmer		6392	3
4	lawyer		7281	5
5	mason		7362	4
6	tailor		6169	1

You can use this query to determine which skill areas do not have assigned employees:

```
SELECT skills.skill_name, emp.emp_no
FROM skills LEFT OUTER JOIN emp ON skills.skill_no=emp.skill_no;
```

The following result is returned. Notice that nulls are displayed as a QUESTION MARK (?) character, which is how BTEQ reports nulls. For more information about BTEQ, see *Basic Teradata® Query Reference*, B035-2414.

skill_name -----	emp_no -----
baker	6123
baker	6234
baker	6169
doctor	?
farmer	6392
lawyer	7362
mason	7281
tailor	?

To include all skills in the result, you must specify an OUTER JOIN. An implicit join like this example that uses just the simple FROM clause does not return rows for nulls (that is, when there are no corresponding employees) and would not list doctor or tailor in the above result.

```
...
FROM employee, skills
...
```

## Related Topics

For information about:

- Using the FROM clause with temporal tables, see “FROM Clause” in *Teradata Vantage™ Temporal Table Support*, B035-1182.
- Self-joins, see [Self-Join](#).
- Outer joins, see [Outer Joins](#).
- Table functions, see “CREATE FUNCTION (Table Form)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.
- How to qualify column and table specifications, see *Teradata Vantage™ SQL Fundamentals*, B035-1141.

## Table Function

Use this option in the FROM clause to invoke a user-defined table function that returns row results as a derived table.



The TABLE option can only be specified once in a FROM clause.

The TABLE option cannot be specified in a FROM clause that also specifies the JOIN option.

For information about creating table functions, see “CREATE FUNCTION (Table Form)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

## Syntax Elements

### Table Function Name

#### TABLE

Keyword to indicate a table function.

#### *function\_name*

Name of the user-defined table function.

#### *expression*

Any valid SQL expression.

If *expression* references a column from a table or subquery in the FROM clause, the table or subquery must appear before the TABLE option.

### RETURNS Clause

#### RETURNS *database\_name.table\_name*

Name of the table and its containing database, if different from the current database, into which the rows processed by the dynamic result row table function are to be inserted.

#### RETURNS *column\_name-data\_type*

Name and data type of one or more columns to be assigned to the row fields returned by a dynamic result row table function.

You cannot use the BLOB AS LOCATOR or CLOB AS LOCATOR forms to specify columns that have a BLOB or CLOB data type.

For more information, see “CREATE FUNCTION (Table Form)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

### LOCAL ORDER BY Clause

Orders qualified rows on each AMP in preparation to be input to a table function.

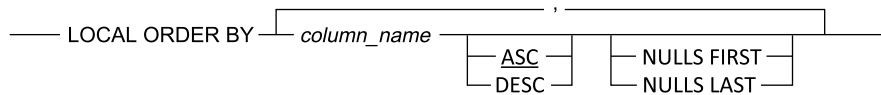
You can only specify a LOCAL ORDER BY clause for a query whose result is to be input to a table function.

The scope of input to the LOCAL ORDER BY clause is limited to:

- Derived tables

- Views
- WITH clause objects

You cannot specify a LOCAL ORDER BY clause with a derived table, view, or WITH clause object that specifies set operations.



LOCAL ORDER BY is a Teradata extension to the ANSI SQL:2011 standard.

### **LOCAL ORDER BY *column\_name***

#### ***column\_position***

#### ***column\_expression***

Qualified rows are ordered on each AMP in preparation to be input to a table function.

You cannot specify a PARTITION BY ANY clause with a LOCAL ORDER BY clause in the same ON clause.

You cannot specify a DIMENSION clause with a LOCAL ORDER BY clause in the same ON clause.

If you use multiple ON clauses and you specify only LOCAL ORDER BY in one of them, all other ON clauses can only specify the DIMENSION option.

If you use multiple ON clauses, you cannot use a LOCAL ORDER BY clause in addition to a DIMENSION clause in the same ON clause.

If you specify multiple HASH BY clauses with LOCAL ORDER BY clauses, the following restrictions apply:

- All the clauses must have the same number of LOCAL ORDER BY columns.
- The data types of the columns must be the same type or matched using an implicit cast.

### **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.

### **NULLS FIRST**

NULL results are to be listed first.

**NULLS LAST**

NULL results are to be listed last.

**HASH BY Clause****HASH BY *column\_name***

Hashes rows across the AMPs in preparation to be input to a table function.

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

You can only specify a HASH BY clause for a statement whose result is to be input to a table function.

HASH BY must be specified as part of a FROM clause. See [FROM Clause](#).

The scope of input to the HASH BY clause is limited to:

- Derived tables
- Views
- WITH clause objects

You cannot specify a HASH BY clause with a derived table, view, or WITH clause object that specifies set operations.

You cannot specify more than one HASH BY clause per statement.

You can specify a HASH BY clause by itself or with a LOCAL ORDER BY clause. If you specify both, the HASH BY clause must precede the LOCAL ORDER BY clause.

When you use a multiple input table operator that has multiple hash by clauses, the following restrictions apply:

- All columns must have the same number of partitioning attributes.
- The corresponding attributes must be the same type or must be types that are compatible so that they can be implicitly cast.

If you specify a LOCAL ORDER BY clause with HASH BY input, the following is required:

- All of the ON clauses must have the same number of LOCAL ORDER BY columns.
- The data types of the columns must be the same or matched using an implicit cast.

**AS Derived Table Name****AS *derived\_table\_name***

Name of a temporary derived table that other clauses in the SELECT statement can reference.

AS is an introductory optional keyword for derived table.

***column\_name***

Optional list of column names that other clauses in the SELECT statement can reference.

If specified, the number of names in the list must match the number of columns in the RETURNS TABLE clause of the CREATE FUNCTION statement that installed the *function\_name* table function on the Teradata platform. The alternate names list corresponds positionally to the corresponding column names in the RETURNS TABLE clause.

If omitted, the names are the same as the column names in the RETURNS TABLE clause of the CREATE FUNCTION statement that installed the *function\_name* table function on the Teradata platform.

## Examples

### Example: Specifying a TABLE Function in the FROM Clause

The following statement inserts all of the rows that the *sales\_retrieve* table function produces into *salestable*:

```
INSERT INTO salestable
SELECT s.store, s.item, s.quantity
FROM TABLE (sales_retrieve(9005)) AS s;
```

### Example: Hash Ordering Input Parameters to a Table Function

This example shows the use of the HASH BY clause for a table UDF. The table function *add2int* takes two integer values as input and returns both of them and their sum.

The query in this example selects all columns from *add2int*, which specifies that its input, *dt*, is hashed by *dt.y1*. The specified hashing might not be relevant to the *add2int* function and is only used for illustration.

The expected processing of the query is that *dt* is first spooled and then hashed by *y1* among the AMPs. The final hashed spool is used as the input to *add2int*.

```
CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER);

CREATE TABLE t2 (
  a2 INTEGER,
  b2 INTEGER);

CREATE FUNCTION add2int (
  a INTEGER,
  b INTEGER)
RETURNS TABLE (
```

```

    addend1 INTEGER,
    addend2 INTEGER,
    mysum INTEGER)
SPECIFIC add2int
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
NOT DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'CS!add3int!add2int.c';
WITH dt(x1,y1) AS (SELECT a1,b1
                    FROM t1)

SELECT *
FROM TABLE (add2int(dt.x1,dt.y1)
HASH BY y1) AS aa;

```

## Example: HASH BY and LOCAL ORDER BY Clauses in the Same Statement

This example shows the use of the HASH BY and LOCAL ORDER BY clauses for sorting the input to a table UDF.

Some applications need to enforce the ordering of input to table UDFs. Rather than sorting the input arguments in the application or table UDF, you can specify the HASH BY and LOCAL ORDER BY clauses when you invoke the table UDF in the FROM clause of a SELECT request. The scope of input includes derived tables, views, and WITH objects.

Consider this table definition:

```

CREATE TABLE tempdata (
    tid INTEGER,
    tts TIMESTAMP,
    x    INTEGER,
    y    INTEGER);

```

Suppose the data in *tempdata* looks like this:

tid	tts	x	y
---	---	-	-
1001	'2008-02-03 14:33:15'	10	11
1001	'2008-02-03 14:44:20'	20	24
1001	'2008-02-03 14:59:08'	31	30

tid	tts	x	y
---	---	-	-
1002	'2008-02-04 11:02:19'	10	11
1002	'2008-02-04 11:33:04'	22	18
1002	'2008-02-04 11:48:27'	29	29

Now consider a table UDF named *char\_from\_rows* that produces a text string that represents all of the timestamp, x, and y values in rows that have the same value for *tid*. Furthermore, the values in the text string are ordered by timestamp. Here is the definition of the table UDF:

```
CREATE FUNCTION char_from_rows(tid INTEGER,
                               tts TIMESTAMP,
                               x  INTEGER,
                               y  INTEGER)
RETURNS TABLE(outID  INTEGER,
               outCHAR VARCHAR(64000))
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!charfromrows!udfsrc/charfromrows.c'
PARAMETER STYLE SQL;
```

The following statement invokes the *char\_from\_rows* table UDF, using a nonrecursive WITH clause to hash the input by *tid* and value order the input on each AMP by *tts*:

```
WITH wq (tID1, tTS1, x1, y1) AS
  (SELECT tID, tTS, x, y
   FROM tempData)
SELECT *
FROM TABLE (char_from_rows(wq.tID1, wq.tTS1, wq.x1, wq.y1)
HASH BY tID1 LOCAL ORDER BY tTS1) AS tudf;
```

The output looks like this:

```
outID outCHAR
-----
1001  2008-02-03 14:33:1510112008-02-03 14:44:2020242008-02-03 14:59:083130
1002  2008-02-04 11:02:1910112008-02-04 11:33:0422182008-02-04 11:48:272929
```

## Example: Local Ordering of Input Parameters to a Table Function

The following example illustrates the use of the LOCAL ORDER BY clauses for a table UDF. The table function *add2int* takes two integer values as input and returns both of them and their sum.

The query in this example selects all columns from *add2int*, which specifies that its input, *dt*, be value-ordered on each AMP by *dt.x1*. Note that the specified local ordering might not be relevant to the *add2int* function and is only used for illustration.

The expected outcome of the query is that the rows on each AMP are sorted by the value of *x1*. The final hashed and sorted spool is used as the input to *add2int*.

```
CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER);

CREATE TABLE t2 (
  a2 INTEGER,
  b2 INTEGER);

CREATE FUNCTION add2int (
  a INTEGER,
  b INTEGER)
RETURNS TABLE (
  addend1 INTEGER,
  addend2 INTEGER,
  mysum INTEGER)
SPECIFIC add2int
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
NOT DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'CS!add3int!add2int.c';
WITH dt(x1,y1) AS (SELECT a1,b1
                      FROM t1)

SELECT *
FROM TABLE (add2int(dt.x1,dt.y1)
LOCAL ORDER BY x1) AS tf;
```

## Table Operator

### Purpose

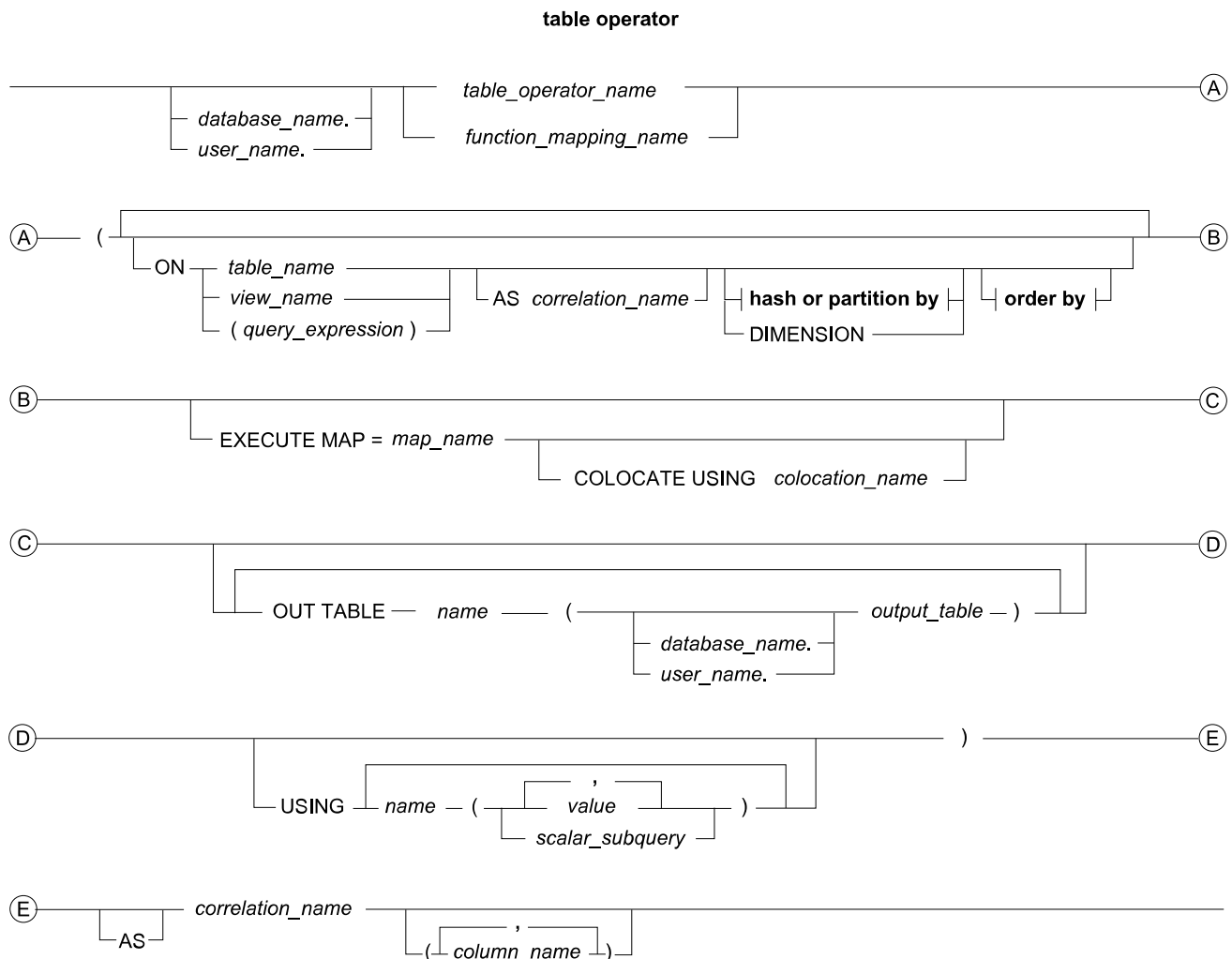
Table operators read one or more input tables, perform operations on the data such as partitioning or aggregation, then write output rows. Table operators can accept an arbitrary row format for each input table and, based on the operation and input row types, produce an arbitrary output row format. For more information on the table operators that Teradata provides, see *Teradata Vantage™ SQL Operators and*

*User-Defined Functions*, B035-1210. For more information on creating table operators, see *Teradata Vantage™ SQL External Routine Programming*, B035-1147.

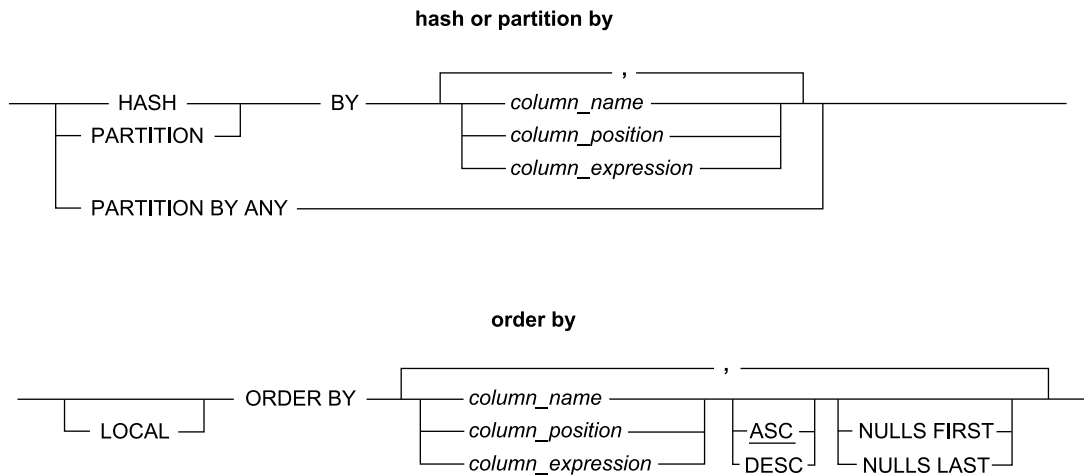
## Required Privileges

You must have the EXECUTE FUNCTION privilege on the function mapping or the containing database.

## Syntax







## Syntax Elements

## Table Operator Name

***database\_name.***

***user\_name.***

Optionally, you can specify the containing database or user. If you do not specify the *database\_name* or *user\_name*, the database or user defaults to one of the following, in order of precedence:

- Current default database for the session
- SYSLIB
- TD SYSFNLIB

***table\_operator\_name***

Name for the table operator. This name cannot be a Teradata reserved word.

### Function Mapping Name

See [Executing a Function Mapping](#) and [Examples: Table Operator Function Mapping](#).

***database\_name.***

***user\_name.***

Optionally, you can specify the containing database or user. If you do not specify the *database\_name* or *user\_name*, the database or user defaults to one of the following, in order of precedence:

- Current default database for the session
- SYSLIB

- TD\_SYSFNLIB

**function\_mapping\_name**

Function mapping name.

For information on how to create a function mapping, see "CREATE FUNCTION MAPPING" in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

**ON Clause**

You can have up to 16 ON clauses. You can use multiple ON clauses with all supported languages for protected and non-protected mode table operators.

If you use multiple ON clauses, the order of the ON clauses must match the order of the stream numbers in the table operator that you use.

Cogroups are used for table operators with multiple ON clauses. For information about cogroups, see *Teradata Vantage™ SQL Operators and User-Defined Functions*, B035-1210.

You cannot use scalar subqueries in table operators with multiple ON clauses or ON clauses using PARTITION BY or HASH BY.

See [Function Mapping and the ON Clause](#).

**table\_name****view\_name**

Table or view expression that is input to the table operator.

**(query\_expression)**

Query expression that is input to the table operator.

**AS correlation\_name**

Alias for the table.

**HASH\_BY column****column\_position****column\_expression**

Optional set of column names on which to hash order globally the columns input to a table function or table operator.

You cannot use scalar subqueries in table operators with multiple ON clauses or ON clauses using PARTITION BY or HASH BY.

When you use a multiple input table operator that has multiple hash by clauses, the following restrictions apply:

- All of them must have the same number of partitioning attributes.
- The corresponding attributes must be the same type or matched using an implicit cast.

If you specify a LOCAL ORDER BY clause along with a HASH BY clause, the following restrictions apply:

- All the clauses must have the same number of LOCAL ORDER BY columns.

- The data types of the columns must be the same type or matched using an implicit cast.

**PARTITION BY *column\_name******column\_position******column\_expression***

Partition for a table specified as input to a table operator.

You can specify a column by name or position, or use an expression that resolves to a column.

You cannot use scalar subqueries in table operators with multiple ON clauses or ON clauses using PARTITION BY or HASH BY.

You cannot specify a row-level security constraint column as a partitioning column.

If you have multiple PARTITION BY clauses, the following restrictions apply:

- All of them must have the same number of partitioning attributes.
- The corresponding attributes must be the same type or matched using an implicit cast.

When you specify an ORDER BY clause with a PARTITION BY clause, the following restrictions apply:

- All the clauses must have the same number of ORDER BY columns.
- The data types of the columns must be the same type or matched using an implicit cast.

**Note:**

Partition keys are important to performance. If the partition keys hash to the same AMP, then some AMPs may be overloaded compared with others. Performance may also be affected if the partition key is a VARCHAR, because VARCHARs may take longer to compare than some other data types.

**PARTITION BY ANY**

Specifies a table with no partitioning or order by attributes.

PARTITION BY ANY preserves the existing distribution of the rows on the AMPs. A PARTITION BY ANY clause followed by ORDER BY clause means that all the rows are ordered by the ORDER BY clause on that partition, and it is functionally equivalent to using LOCAL ORDER BY without using a HASH BY clause.

The following restrictions apply:

- You cannot specify a PARTITION BY ANY clause and a LOCAL ORDER BY clause in the same ON clause.
- If you specify multiple ON clauses with a table operator, you can only specify one PARTITION BY ANY clause. All other clauses must be DIMENSION.

**DIMENSION**

Specifies that a duplicate copy of the dimension table is created for every partition on which the function operates.

You can specify zero or more DIMENSION clauses for each ON clause.

DIMENSION input is useful when the input table is a small look up table for facts or is a trained model, such as that used for machine learning. For a look up table using DIMENSION, the rows are duplicated to all the AMPs. Each AMP holds one instance of the look up table in memory and uses it to process each row of another input.

For machine learning, you can store a model in the database that predicts the outcome of a particular data set and then use the stored model as dimensional input to a function.

The following restrictions apply:

- You cannot use a LOCAL ORDER BY clause and a DIMENSION clause in the same ON clause.
- If you have only one ON clause as input to a table operator, you cannot use DIMENSION in it. You must have at least one PARTITION BY or HASH BY clause in a second ON clause to use DIMENSION.
- If you use multiple ON clauses and you specify only LOCAL ORDER BY in one of them, then all other ON clauses can only specify DIMENSION.
- If you use the SCRIPT table operator, you cannot use a DIMENSION in the ON clause.

For information about the SCRIPT table operator, see *Teradata Vantage™ SQL Operators and User-Defined Functions*, B035-1210.

#### **LOCAL ORDER BY *column\_name***

##### ***column\_position***

##### ***column\_expression***

Qualified rows are ordered on each AMP in preparation to be input to a table function.

You cannot specify a PARTITION BY ANY clause with a LOCAL ORDER BY clause in the same ON clause.

You cannot specify a DIMENSION clause with a LOCAL ORDER BY clause in the same ON clause.

If you use multiple ON clauses and you specify only LOCAL ORDER BY in one of them, all other ON clauses can only specify the DIMENSION option.

If you use multiple ON clauses, you cannot use a LOCAL ORDER BY clause in addition to a DIMENSION clause in the same ON clause.

If you specify multiple HASH BY clauses with LOCAL ORDER BY clauses, the following restrictions apply:

- All the clauses must have the same number of LOCAL ORDER BY columns.
- The data types of the columns must be the same type or matched using an implicit cast.

#### **ORDER BY**

Defines how result sets are sorted. If you do not use this clause, result rows are returned unsorted.

You cannot specify ORDER BY as the only option in an ON clause. You must combine it with a PARTITION BY, PARTITION BY ANY, HASH BY, or DIMENSION clause.

If you specify an ORDER BY clause along with a PARTITION BY clause, the following restrictions apply:

- All the clauses must have the same number of ORDER BY columns.
- The data types of the columns must be the same type or matched using an implicit cast.

When NORMALIZE and ORDER BY are specified in a query, the rows are normalized first and the result is ordered on the normalized result.

## EXECUTE MAP Clause

Specify a contiguous or sparse map and, optionally, colocation name for table operator execution. The table operator executes only on the AMPs in the map. The database distributes input rows to the AMPs in the specified or default map before running the table operator on the specified or default map if the input rows are not already distributed per that map.

You must have been granted the specified map.

When the EXECUTE MAP clause is not specified, the execute map will be as defined by the table operator. See the EXECUTE MAP option under CREATE FUNCTION and REPLACE FUNCTION (Table Form) in User-Defined Function Statements of *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

### *map\_name*

Name of contiguous or sparse map.

You cannot specify TD\_DataDictionaryMap or TD\_GlobalMap.

### COLOCATE USING *colocation\_name*

Name for collocating the function on the same AMPs with other functions, tables, join indexes, or hash indexes.

You can only specify this option for a sparse map. For a contiguous map, a colocation name is not needed for colocation and the *colocation\_name* is set to NULL.

If you do not specify a colocation name, the name defaults to *database\_operator*, where *database* is the name of the database or user followed by an underscore (\_) and *operator* is the name of the table operator. If *database* exceeds 63 characters, *database* is truncated to 63 characters. If *operator* exceeds 64 characters, *operator* is truncated to 64 characters.

## Example: Specifying a Sparse Map for a Table Operator

Because this example specifies a sparse map, *colocation\_name* defaults to db1\_tableop1, which overrides the colocation name specified, if any, when the table operator was created.

```
SELECT * FROM db1.tableop1
  ( ON ( SELECT * FROM tab1) PARTITION BY col1
    ON ( SELECT * FROM tab2) PARTITION BY col2
  EXECUTE MAP = OneAMPMap
```

```

        USING myvalue(10,20)
    ) AS d1;

```

## Output Table Clause

You can only specify the output table clause for a function mapping. See "CREATE FUNCTION MAPPING" in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

### OUT TABLE

Keywords to specify an output table.

#### ***name***

Output argument name, which must match one of the OUT TABLE names in the function mapping definition.

#### ***database\_name***

Name of the database containing the output table.

#### ***user\_name***

Name of the user containing the output table.

#### ***output\_table***

Name of the output table.

## USING Custom Clause

### USING

Keyword to introduce the custom clause.

You can specify variables. During function processing, variable values are substituted for the corresponding parameters in the function mapping definition. The variables are not sent for function processing. See [Function Mapping and the USING Clause](#).

#### ***name (value)***

One or more name-value pairs.

#### ***name (scalar\_subquery)***

You can also specify a subquery that returns a single row. Only one subquery is allowed for each name. The subquery must return only one row and the WHERE clause of the subquery must reference a unique primary index or unique secondary index. The row can have multiple values (columns), but all values must be the same data type.

## AS Correlation Name

### AS

Optional keyword introducing correlation\_name.

ANSI SQL refers to aliases as correlation names. They are also referred to as range variables.

If you have multiple ON clauses, each *correlation name* used in the query must be unique.

The *correlation\_name* is an alias for the column that is referenced by *column\_name*.

For (*column\_name*), you can specify one or more column names.

***correlation\_name***

An alias for the column that is referenced by *column\_name*.

***column\_name***

One or more column names.

## Usage Notes

### Executing a Function Mapping

In this scenario, the Attribution function mapping is on the local Teradata system in SYSLIB and in the database *alias\_db*.

In the statement below, because user *u1* does not specify a database, the system checks for the Attribution function mapping object in database *u1*. If the Attribution object is not on *u1*, the system checks SYSLIB. Because the Attribution is on the local system, the system invokes local implementation of Attribution.

```
SELECT * from Attribution (
  ON appl_view_db.input_table PARTITION BY 1
  ON appl_view_db.model1_table as model1 DIMENSION
  USING
    TimestampColumn('tscol')
    EventColumn('eventcol')
    WindowSize('rows:10')
) as dt;
```

Then, user *u1* specifies another database.

```
DATABASE alias_db;
```

Then, user *u1* can invoke the function mapping Attribution on the remote system with either of the following statements.

```
SELECT * FROM Attribution (
  ON appl_view_db.input_table PARTITION BY 1
  ON appl_view_db.model1_table AS model1 DIMENSION
  USING
    TimestampColumn('tscol')
    EventColumn('eventcol')
```

```

        WindowSize('rows:10')
    ) AS dt;

SELECT * FROM alias_db.Attribution (
    ON appl_view_db.input_table PARTITION BY 1
    ON appl_view_db.model1_table AS model1 DIMENSION
    USING
        TimestampColumn('tscol')
        EventColumn('eventcol')
        WindowSize('rows:10')
    ) AS dt;

```

## Function Mapping and the ON Clause

For information on how to create a function mapping, see "CREATE FUNCTION MAPPING" in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

The *correlation\_name* must correspond to a *name* IN TABLE parameter in the function mapping definition. The *correlation\_name* can be any name if the function mapping definition specifies ANY IN TABLE.

The statement below defines the function mapping Attribution.

```

CREATE FUNCTION MAPPING appl_view_db.Attribution
  FOR attribution SERVER coprocessor
  MAP JSON ( '{ "function_version": "1.0" }' )
  USING
    ANY IN TABLE,
    conversion IN TABLE,
    excluding IN TABLE,
    optional IN TABLE,
    model1 IN TABLE,
    model2 IN TABLE,
    EventColumn, TimestampColumn, WindowSize;

```

The SELECT statement below executes the function mapping Attribution.

```

SELECT * FROM appl_view_db.Attribution (
    ON appl_view_db.input_table PARTITION BY 1
    ON appl_view_db.model1 AS model1 DIMENSION
    USING
        TimestampColumn('tscol')
        EventColumn('eventcol')
        WindowSize('rows:10')
    ) AS dt;

```



## Function Mapping and the USING Clause

Values that you specify for parameters in the USING clause of a function mapping override the default values in the function mapping definition.

Arguments that are defined with values in the mapping definition, but not specified in the USING custom clause, are appended by the system when the function mapping is executed. If you specify default values in the mapping definition, those default values do not have to be specified in the function mapping execution.

In this scenario, the function mapping definition specifies a default clicklag value of 20.

```
CREATE FUNCTION MAPPING appl_view_db.sessionize
  FOR sessionize SERVER coprocessor
  USING
    Timecolumn, timeout(100), clicklag(20), emitnull;
```

Because the SELECT statement below specifies the clicklag value as 200, the function mapping execution uses the clicklag value of 200 instead of the clicklag value of 20 specified in the function mapping definition.

```
SELECT * FROM test_db.sessionize (
  ON test_user.t1 AS InputTable PARTITION BY 1
  USING
    Timecolumn('Test')
    clicklag(200)
) AS dt;
```

This SELECT statement below does not specify the emitnull parameter and because the mapping definition does not include a default value for emitnull, the emitnull parameter is not passed to the remote system during execution.

```
SELECT * FROM test_db.sessionize (
  ON test_user.t1 AS InputTable PARTITION BY 1
  USING
    Timeout(10)
    Timecolumn('Test')
    clicklag(200)
) AS dt;
```

Although the SELECT statement below does not include the clicklag and timeout parameters, the function mapping definition specifies the clicklag and timeout parameters with default values.

```
SELECT * FROM test_db.sessionize (
  ON test_user.t1 AS InputTable PARTITION BY 1
  USING
```

```
Timecolumn('Test')
) AS dt;
```

The SELECT statement function mapping must provide all parameters required by remote system to execute the function. Otherwise, an error occurs.

To execute the Sessionize function, for example, the Timecolumn parameter is mandatory. Because the function mapping definition below does not include a default value for Timecolumn, you must specify a Timecolumn value in the SELECT statement function mapping.

```
CREATE FUNCTION MAPPING appl_view_db.sessionize
  FOR sessionize SERVER coprocessor
  USING
    Timecolumn, Timeout, clicklag(20), emitnull;
```

The SELECT statement below specifies a Timecolumn value for the function mapping.

```
SELECT * FROM test_db.sessionize (
  ON test_user.t1 AS InputTable PARTITION BY 1
  USING
    Timeout(10)
    Timecolumn('Test')
    clicklag(20)
) AS dt;
```

## Examples

### Examples: Table Operator Function Mapping

This SELECT statement includes a function mapping with an ON clause.

```
SELECT * FROM appl_view_db.CCMPPrepare
  ( ON tab1 PARTITION BY id);
```

This SELECT statement includes a function mapping that specifies an input table, output table, and additional parameters.

```
SELECT * FROM appl_view_db.GLM (
  ON loan_info2 AS InputTable
  OUTPUT TABLE OutputTable(loan_info2_out_t)
  USING
    ColumnNames ( 'DELINQUENT2PLUS', 'CREDIT_SCORE', 'FIRST_PAYMENT_DATE',
                  'MORTGAGE_INSURANCE_PERCENTAGE', 'NUMBER_OF_UNITS',
                  'CLTV, DTI_RATIO', 'ORIGINAL_UPB', 'ORIGINAL_LTV',
                  'ORIGINAL_INTEREST_RATE', 'ORIGINAL_LOAN_TERM',
```

```

        'NUMBER_OF_BORROWERS' )
Family ('logistic') Link ('LOGIT')
Weight (1) Threshold (0.01) MaxIterNum (25)
Step ('false') Intercept ('true') ) AS dt;

```

## Example: SELECT and the Function Mapping Definition

In a SELECT statement with a function mapping, each *correlation\_name*, USING custom clause parameter, or OUT TABLE *name* that you specify must correspond to a *correlation\_name*, USING custom clause parameter, or OUT TABLE *name* in the function mapping definition. For information on how to create a function mapping, see "CREATE FUNCTION MAPPING" in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

Below is the function mapping definition *coxph* for the function *coxph* on the server *coprocessor*.

```

CREATE FUNCTION MAPPING appl_view_db.coxph
FOR coxph SERVER coprocessor
MAP JSON ( '{ "function_version": "1.0" }' )
  USING
    InputTable IN TABLE,
    CoefficientTable OUT TABLE,
    LinearPredictorTable OUT TABLE,
    TimeIntervalColumn, EventColumn, Threshold, MaxInterNum,
    FeatureColumns;

```

In the SELECT statement below, each *correlation\_name*, USING custom clause parameter, or OUT TABLE *name* corresponds to a *correlation\_name*, USING custom clause parameter, or OUT TABLE *name* in the function mapping definition.

```

SELECT * FROM appl_view_db.coxph (
  ON appl_view_db.input_table AS inputtable PARTITION BY 1
  OUTPUT TABLE CoefficientTable(coefficient_table)
  OUTPUT TABLE LinearPredictorTable(linearpredictor_table)
  USING
    FeatureColumns('c1','c2')
    TimeIntervalColumn('t1')
    EventColumn('e1')
) AS dt;

```

## Examples: Function Processing with Variable Substitution

### Example: Single Value Specified for a Concatenated Variable Expression

In the example below, *MaxStep* is substituted for *Maxnum* at function processing. However, *ValueColumn* is not submitted because *AttributeValueColumn* does not have a value.

This is the function mapping definition *usr\_SVMSparse* for the function *SparseSVMTrainer*. The *ValueColumn* value consists of a concatenated variable expression that includes the variables *AttributeValueColumn* and *Maxnum*.

```
CREATE FUNCTION MAPPING usr_SVMSparse
FOR SparseSVMTrainer SERVER TD_SERVER_DB.coprocessor
USING
  InputTable IN TABLE ,
  ModelTable OUT TABLE ,
  IDColumn ,
  SampleIdColumn ,
  AttributeColumn,
  LabelColumn,
  ValueColumn(AttributeValueColumn||'_'||Maxnum),
  HashProjection , "Hash" , HashBuckets ,
  Cost , Bias , ClassWeights ,
  MaxStep(Maxnum) ,
  Epsilon , Seed , SequenceInputBy
;
```

The SELECT statement specifies a value of 150 for *Maxnum*.

```
SELECT * FROM usr_SVMSparse (
ON svm_iris_input_train as InputTable
OUT TABLE ModelTable (svm_iris_model)
USING
  SampleIDColumn ('id')
  AttributeColumn ('attribute')
  LabelColumn ('species')
  Maxnum (150)
  Seed ('0')
) as dt;
```

Upon submittal, the query is rewritten as follows for processing by the function *SparseSVMTrainer*. Per the function mapping definition, the variable *MaxStep* is substituted for *Maxnum*. *ValueColumn* is not submitted for processing because the SELECT statement did not specify a value for *AttributeValueColumn*.

```

SELECT * FROM SparseSVMTrainer (
ON svm_iris_input_train as InputTable
OUT TABLE ModelTable (svm_iris_model)
USING
SampleIDColumn ('id')
AttributeColumn ('attribute')
LabelColumn ('species')
MaxStep (150)
Seed ('0')
) as dt;

```

### Example: Parameter with a Concatenated Variable Expression Submitted

The function mapping definition *usr\_SVMSparse* includes the parameter *ValueColumn* with a concatenated variable expression consisting of *AttributeValueColumn* and *Maxnum*.

```

CREATE FUNCTION MAPPING usr_SVMSparse
FOR SparseSVMTrainer SERVER TD_SERVER_DB.coprocessor
USING
InputTable IN TABLE ,
ModelTable OUT TABLE ,
IDColumn ,
SampleIdColumn ,
AttributeColumn,
LabelColumn,
ValueColumn(AttributeValueColumn||'_'||Maxnum),
HashProjection , "Hash" , HashBuckets ,
Cost , Bias , ClassWeights ,
MaxStep(Maxnum) ,
Epsilon , Seed , SequenceInputBy
;

```

The SELECT statement specifies values for *AttributeValueColumn* and *Maxnum*.

```

SELECT * FROM usr_SVMSparse (
ON svm_iris_input_train as InputTable
OUT TABLE ModelTable (svm_iris_model)
USING
SampleIDColumn ('id')
AttributeColumn ('attribute')
LabelColumn ('species')
AttributeValuecolumn('value')
Maxnum (150)
Seed ('0')
) as dt;

```

Upon submittal, the query is rewritten as follows. The concatenated variable expression for *ValueColumn* resolves to *value\_150*. Because the function mapping definition substitutes *MaxStep* for *Maxnum*, *Maxstep* is submitted with a value of 150.

```
SELECT * FROM SparseSVMTrainer (
ON svm_iris_input_train as InputTable
OUT TABLE ModelTable (svm_iris_model)
USING
SampleIDColumn ('id')
AttributeColumn ('attribute')
LabelColumn ('species')
Valuecolumn('value_150')
MaxStep (150)
Seed ('0')
) as dt;
```

### Example: Scalar Subquery Variable Substitution

In the example below, the scalar subqueries in the SELECT statement are executed to provide values for *AttributeValueColumn* and *Maxnum*. Upon submittal, the query is rewritten to substitute the resolved values for *ValueColumn* and *MaxStep*.

The function mapping definition *usr\_SVMSparse* includes the parameter *ValueColumn* with a concatenated variable expression consisting of *AttributeValueColumn* and *Maxnum*.

```
CREATE FUNCTION MAPPING usr_SVMSparse
FOR SparseSVMTrainer SERVER TD_SERVER_DB.coprocessor
USING
InputTable IN TABLE ,
ModelTable OUT TABLE ,
IDColumn ,
SampleIdColumn ,
AttributeColumn,
LabelColumn,
ValueColumn(AttributeValueColumn||'_'||Maxnum),
HashProjection , "Hash" , HashBuckets ,
Cost , Bias , ClassWeights ,
MaxStep(Maxnum) ,
Epsilon , Seed , SequenceInputBy
;
```

The SELECT statement specifies scalar subqueries for *AttributeValuecolumn* and *Maxnum*.

```
SELECT * FROM usr_SVMSparse (
ON svm_iris_input_train as InputTable
OUT TABLE ModelTable (svm_iris_model)
```

```

USING
SampleIDColumn ('id')
AttributeColumn ('attribute')
LabelColumn ('species')
AttributeValuecolumn(SELECT colvarch from ssqtbl WHERE username=CURRENT_USER)
Maxnum (SELECT colint FROM ssqtbl WHERE username=USER)
Seed ('0')
) as dt;

```

Upon submittal, the query is rewritten as follows for processing by the function *SparseSVMTrainer*. The scalar subquery for *AttributeValueColumn* resolves to *value* and the scalar subquery for *Maxnum* resolves to 1. The concatenated variable expression for *ValueColumn* resolves to *value\_1*. Per the function mapping definition, *MaxStep* is substituted for *Maxnum*.

```

SELECT * FROM SparseSVMTrainer (
ON svm_iris_input_train as InputTable
OUT TABLE ModelTable ('svm_iris_model')
USING
SampleIDColumn ('id')
AttributeColumn ('attribute')
LabelColumn ('species')
Valuecolumn('value_1')
MaxStep (1)
Seed ('0')
) as dt;

```

## Example: Function Processing with Variable Substitution for Input Tables and Parameters

The function mapping definition `usr_AllPairsShortestPath` specifies variables for input tables and parameters.

```

CREATE FUNCTION MAPPING usr_AllPairsShortestPath
FOR AllPairsShortestPath SERVER TD_SERVER_DB.coprocessor
USING
vertices(vertex) IN TABLE ,
edges(edge) IN TABLE ,
sources(source) IN TABLE ,
targets(trgt) IN TABLE ,
TargetKey(GroupColumn) ,
EdgeWeight('calls') ,
MaxDistance(Distance) ,
Directed ,GroupSize ,SequenceInputBy;

```

The SELECT statement specifies values for the variables *Distance* and *GroupColumn*.

```
SELECT * FROM usr_AllPairsShortestPath (
  ON callers AS vertices PARTITION BY callerid
  ON calls AS edges PARTITION BY callerfrom
  USING
  GroupColumn ('callerto')
  Distance ('-1')
) as dt ORDER BY source, target;
```

Upon submittal, the query is rewritten as follows. Per the mapping definition, *TargetKey* is substituted for *GroupColumnquery* and *MaxDistance* is substituted for *Distance*. *EdgeWeight* is also included because the mapping definition specifies a default value for this parameter.

```
SELECT * FROM AllPairsShortestPath (
  ON callers AS vertices PARTITION BY callerid
  ON calls AS edges PARTITION BY callerfrom
  TargetKey ('callerto')
  EdgeWeight ('calls')
  MaxDistance ('-1')
) ORDER BY source, target;
```

## Example: Function Processing with a Concatenated Variable Expression

The function mapping definition *usr\_AllPairsShortestPath* for the function *AllPairsShortestPath* specifies variables for input tables and parameters, including a concatenated variable expression for *TargetKey*.

```
CREATE FUNCTION MAPPING usr_AllPairsShortestPath
FOR AllPairsShortestPath SERVER TD_SERVER_DB.coprocessor
USING
vertices(vertex) IN TABLE ,
edges(edge) IN TABLE ,
sources(source) IN TABLE ,
targets(trgt) IN TABLE ,
TargetKey(GroupColumn||'_'||Distance) ,
EdgeWeight('calls') ,
MaxDistance(Distance) ,
Directed ,GroupSize ,SequenceInputBy;
```

The SELECT statement specifies a value for the variable *Distance*.

```
SELECT * FROM usr_AllPairsShortestPath (
  ON callers AS vertices PARTITION BY callerid
  ON calls AS edges PARTITION BY callerfrom
  USING
```



```
Distance ('1')
) as dt ORDER BY source, target;
```

Upon submittal, the query is rewritten as follows. Per the mapping definition, *MaxDistance* is substituted for *Distance*. *EdgeWeight* is also included because the mapping definition specifies a default for this parameter. *TargetKey* is not included because the *GroupColumn* variable in the concatenated variable expression is not resolved.

```
SELECT * FROM AllPairsShortestPath (
ON callers AS vertices PARTITION BY callerid
ON calls AS edges PARTITION BY callerfrom
EdgeWeight ('calls')
MaxDistance ('1')
) ORDER BY source, target;
```

## Examples: Function Processing with Default Values for Variables

The function mapping definition *usr\_AllPairsShortestPath* for the function *AllPairsShortestPath* includes a concatenated expression for the variable *TargetKey* and default values for *EdgeWeight* and *Distance*.

```
CREATE FUNCTION MAPPING usr_AllPairsShortestPath
FOR AllPairsShortestPath SERVER TD_SERVER_DB.coprocessor
USING
vertices(vertex) IN TABLE ,
edges(edge) IN TABLE ,
sources(source) IN TABLE ,
targets(trgt) IN TABLE ,
TargetKey(GroupColumn||'_'||Distance) ,
EdgeWeight('calls') ,
MaxDistance(Distance) , Distance(1),
Directed ,GroupSize ,SequenceInputBy;
```

The SELECT statement specifies a value for the variable *GroupColumn*.

```
SELECT * FROM usr_AllPairsShortestPath (
ON callers AS vertices PARTITION BY callerid
ON calls AS edges PARTITION BY callerfrom
USING
GroupColumn('tab1')
) as dt ORDER BY source, target;
```

Upon submittal, the query is rewritten as follows. *TargetKey* is included because the mapping definition provides a default value for *Distance* and the query provides a value for *GroupColumn* to resolve the concatenated variable expression.

Per the mapping definition, *MaxDistance* is substituted for *Distance*. *EdgeWeight* is included because the mapping definition provides a default value.

```
SELECT * FROM AllPairsShortestPath (
ON callers AS vertices PARTITION BY callerid
ON calls AS edges PARTITION BY callerfrom
USING
TargetKey('tabl_1')
MaxDistance('1')
EdgeWeight('calls')
) as dt ORDER BY source, target;
```

## Example: Function Processing with Nested Variables

The function mapping definition *user\_SparseSVMTrainer* for the function *SparseSVMTrainer* includes nested parameters:

```
Var1(old_var1),
var2(old_var2||'_'|| var3),
old_var1(var3)

CREATE FUNCTION MAPPING user_SparseSVMTrainer
FOR SparseSVMTrainer SERVER coprocessor
Using
ANY IN TABLE,
inputtable IN TABLE,
ModelTable,
IDColumn,
AttributeColumn,
LabelColumn ('species') ,
Valuecolumn(var2 || '_' || var1) ,
MaxStep (var1) ,
Seed ('0'),
var1(old_var1),
var2(old_var2||'_'|| var3),
old_var1(var3),
var3(10)
;
```

This SELECT statement specifies values for *var3* and *old\_var2*.

```
SELECT * FROM user_SparseSVMTrainer (
ON svm_iris_input_train as InputTable PARTITION BY 1
USING
ModelTable ('svm_iris_model')
```

```

LabelColumn ('species')
MaxStep (10)
Seed ('1')
var3 (2)
old_var2('sample')
IDColumn('id')
AttributeColumn('attribute')
) as dt1;

```

The query is rewritten as follows. *var2* resolves to *sample\_2* based on the concatenated expression of *old\_var2* and *var3*. The mapping definition substitutes *var1* for *old\_var1*. *old\_var1* is substituted for *var3*, which is specified as 2 in the query. The concatenated expression defined for *ValueColumn* resolves to *sample\_2\_2*.

```

SELECT * FROM user_SparseSVMTrainer (
ON svm_iris_input_train as InputTable PARTITION BY 1
USING
ModelTable ('svm_iris_model')
LabelColumn ('species')
Valuecolumn('sample_2_2')
MaxStep (10)
Seed ('1')
IDColumn('id')
AttributeColumn('attribute')
) as dt1;

```

This SELECT statement specifies values for *var1* and *var2*.

```

SELECT * FROM user_SparseSVMTrainer (
ON svm_iris_input_train as InputTable PARTITION BY 1
USING
ModelTable ('svm_iris_model')
LabelColumn ('species')
IDColumn('id')
AttributeColumn('attribute')
Seed ('1')
var1 (2)
var2('sample')
) as dt1;

```

The query is rewritten as follows. The function mapping defines *ValueColumn* as the concatenation of *var1var2*. *MaxStep* is substituted for *var1*.

```

SELECT * FROM user_SparseSVMTrainer (
ON svm_iris_input_train as InputTable PARTITION BY 1

```

```

USING
ModelTable ('svm_iris_model')
IDColumn('id')
AttributeColumn('attribute')
LabelColumn ('species')
Valuecolumn('sample_2')
MaxStep (2)
Seed ('1')
) as dt1;

```

This SELECT statement specifies values for *old\_var1* and *var2*.

```

SELECT * FROM user_SparseSVMTrainer (
ON svm_iris_input_train as InputTable PARTITION BY 1
USING
ModelTable ('svm_iris_model')
LabelColumn ('species')
IDColumn('id')
AttributeColumn('attribute')
Seed ('1')
old_var1 (2)
var2('sample')
) as dt1;

```

The query is rewritten as follows. Per the function mapping definition, *var1* is substituted for *old\_var1* with a value of 2. The *ValueColumn* concatenated expression resolves to *sample\_2*. *MaxStep* is substituted for *var1* with a value of 2.

```

SELECT * FROM user_SparseSVMTrainer (
ON svm_iris_input_train as InputTable PARTITION BY 1
USING
ModelTable ('svm_iris_model')
LabelColumn ('species')
IDColumn('id')
AttributeColumn('attribute')
Valuecolumn('sample_2')
MaxStep (2)
Seed ('1')
) as dt1;

```

## Example: Function Processing with a Scalar Subquery

The function mapping definition *usr\_AllPairsShortestPath* for the function *AllPairsShortestPath* specifies *TargetKey* as a concatenated variable expression consisting of *GroupColumn* and *Distance*. The default value for *Distance* is derived from the scalar subquery:

```
SELECT distnum FROM ssqtbl WHERE username=CURRENT_USER

CREATE FUNCTION MAPPING usr_AllPairsShortestPath
FOR AllPairsShortestPath SERVER TD_SERVER_DB.coprocessor
USING
vertices(vertex) IN TABLE ,
edges(edge) IN TABLE ,
sources(source) IN TABLE ,
targets(trgt) IN TABLE ,
TargetKey(GroupColumn||'_'||Distance) ,
EdgeWeight('calls') ,
MaxDistance(Distance) , Distance(SELECT distnum FROM ssqtbl WHERE
username=CURRENT_USER),
Directed ,GroupSize ,SequenceInputBy;
```

This SELECT statement specifies a scalar subquery for *GroupColumn*. The value for *distnum* is 5 and *tablename* is *tbl1* in the scalar subquery table *ssqtbl* for the CURRENT logged in user.

```
SELECT * FROM usr_AllPairsShortestPath (
ON callers AS vertices PARTITION BY callerid
ON calls AS edges PARTITION BY callerfrom
USING
GroupColumn(SELECT tablename FROM ssqtbl WHERE username=CURRENT_USER)
) as dt ORDER BY source, target;
```

Upon submittal, the query is rewritten as follows. The function mapping definition specifies a concatenated expression for *TargetKey*, which resolves to *tbl\_5*. The *Distance* scalar subquery resolves to 5. *MaxDistance* is substituted for *Distance*.

```
SELECT * FROM AllPairsShortestPath (
ON callers AS vertices PARTITION BY callerid
ON calls AS edges PARTITION BY callerfrom
USING
TargetKey('tbl_5')
MaxDistance('5')
EdgeWeight('calls')
) as dt ORDER BY source, target;
```

## Example: Function Processing with Input Table Variable Substitution

If an input table correlation name in the SELECT statement matches an IN TABLE substitution variable in the function mapping definition, the corresponding IN TABLE parameter name in the function mapping definition is substituted for function processing.

The function mapping definition `usr_AllPairsShortestPath` for the function `AllPairsShortestPath` includes IN TABLE variables for substitution.

```
CREATE FUNCTION MAPPING usr_AllPairsShortestPath
FOR AllPairsShortestPath SERVER TD_SERVER_DB.coprocessor
USING
vertices(vertex) IN TABLE ,
edges(edge) IN TABLE ,
sources(source) IN TABLE ,
targets(trgt) IN TABLE ,
TargetKey(GroupCol) ,
EdgeWeight(2) ,
MaxDistance(Distance) ,
Directed ,GroupSize ,SequenceInputBy;
```

This SELECT statement specifies the *vertex* and *edge* as table correlation names.

```
SELECT * FROM usr_AllPairsShortestPath (
ON callers AS vertex PARTITION BY callerid
ON calls AS edge PARTITION BY callerfrom
USING
TargetKey ('callerto')
EdgeWeight ('calls')
MaxDistance ('-1')
) as dt ORDER BY source, target;
```

The query is rewritten as follows during function processing. The table correlation name *vertices* is substituted for *vertex* and the table correlation name *edges* is substituted for *edge*.

```
SELECT * FROM AllPairsShortestPath (
ON callers AS vertices PARTITION BY callerid
ON calls AS edges PARTITION BY callerfrom
TargetKey ('callerto')
EdgeWeight ('calls')
MaxDistance ('-1')
) ORDER BY source, target;
```

## Example: Function Processing with Any Input Table Substitution

The function mapping definition *usr\_AllPairsShortestPath* for the function *AllPairsShortestPath* specifies that *vertices* can be substituted for any input table that does not correspond to any of the other specified input tables.

```
CREATE FUNCTION MAPPING syslib.usr_AllPairsShortestPath
FOR AllPairsShortestPath SERVER coprocessor
USING
vertices(ANY) IN TABLE,
edges(edge) IN TABLE,
sources IN TABLE,
targets(dest) OUT TABLE,
TargetKey('col1'),MaxDistance(Distance),EdgeWeight('2'),
GroupCol('Col_'||GroupColumn||'_'||Distance),
Distance(SELECT distnum FROM ssqtbl WHERE username=CURRENT_USER),
Directed;
```

This SELECT statement specifies *vertex* and *edges* as table correlation names.

```
SELECT * FROM usr_AllPairsShortestPath (
ON callers AS vertex PARTITION BY callerid
ON calls AS edges PARTITION BY callerfrom
USING
TargetKey ('callerto')
EdgeWeight ('calls')
MaxDistance ('-1')
) as dt ORDER BY source, target;
```

The query is rewritten as follows during function processing. Because the function mapping definition specifies that *vertices* can be substituted for any table that does not correspond to any of the other specified input tables, *vertices* is substituted for *vertex*.

```
SELECT * FROM AllPairsShortestPath (
ON callers AS vertices PARTITION BY callerid
ON calls AS edges PARTITION BY callerfrom
TargetKey ('callerto')
EdgeWeight ('calls')
MaxDistance ('-1')
) ORDER BY source, target;
```

This SELECT statement specifies *vertices* and *edges* as table correlation names.

```

SELECT * FROM usr_AllPairsShortestPath (
ON callers AS vertices PARTITION BY callerid
ON calls AS edges PARTITION BY callerfrom
USING
TargetKey ('callerto')
EdgeWeight ('calls')
MaxDistance ('-1')
) as dt ORDER BY source, target;

```

The query is rewritten as follows during function processing, specifying the *vertices* and *edges* input tables.

```

SELECT * FROM AllPairsShortestPath (
ON callers AS vertices PARTITION BY callerid
ON calls AS edges PARTITION BY callerfrom
TargetKey ('callerto')
EdgeWeight ('calls')
MaxDistance ('-1')
) ORDER BY source, target;

```

This SELECT statement does not specify a correlation for the table *callers* and specifies *edges* as a correlation for *calls*.

```

SELECT * FROM usr_AllPairsShortestPath (
ON callers PARTITION BY callerid
ON calls AS edges PARTITION BY callerfrom
USING
TargetKey ('callerto')
EdgeWeight ('calls')
MaxDistance ('-1')
) as dt ORDER BY source, target;

```

The query is rewritten as follows during function processing. Because the function mapping definition specifies that *vertices* can be substituted for any table that does not have a correlation, *vertices* is correlated to *callers*.

```

SELECT * FROM AllPairsShortestPath (
ON callers AS vertices PARTITION BY callerid
ON calls AS edges PARTITION BY callerfrom
TargetKey ('callerto')
EdgeWeight ('calls')
MaxDistance ('-1')
) ORDER BY source, target;

```



## Derived Tables

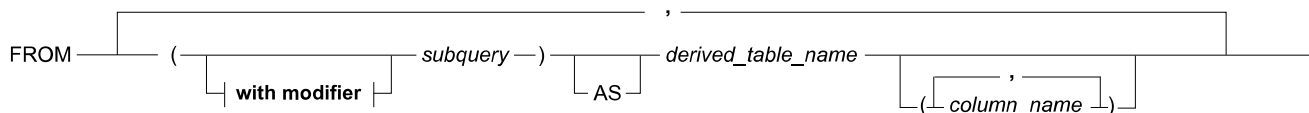
In SQL statements, a referenced table can be a base table, a derived table, a queue table, or a view. The semantics of derived tables and views are identical. For information about views, see “CREATE VIEW” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

A derived table is obtained from one or more other tables as the result of a subquery.

This option enables the FROM clause to specify a spool made up of selected data from an underlying table set. The derived table is similar to a viewed table.

You can invoke a scalar UDF from any point in a derived table.

### Syntax



### Syntax Elements

#### with modifier

See [WITH Modifier](#).

#### (subquery)

Nested SELECT expressions.

You can specify NORMALIZE in a subquery.

You cannot specify SELECT AND CONSUME statements in a subquery. You cannot specify an EXPAND ON clause in a subquery.

#### AS

Optional introductory clause to the derived table name.

#### derived\_table\_name

Name of the derived table.

#### column\_name

List of column names or expressions listed in the subquery. Enables referencing subquery columns by name.

#### Note:

Specify column names only. Do not use forms such as *table\_name.column\_name* or *database\_name.table\_name.column\_name*.

If specified, the number of names in the list must match the number of columns in the RETURNS TABLE clause of the CREATE FUNCTION statement that installed the *function\_name* table

function. The alternate names list corresponds positionally to the corresponding column names in the RETURNS TABLE clause.

If omitted, the names are the same as the column names in the RETURNS TABLE clause of the CREATE FUNCTION statement that installed the *function\_name* table function.

## ANSI Compliance

Derived tables are ANSI SQL:2011-compliant.

## Usage Notes

### Uses for Derived Tables

Derived tables are an optional feature that you can specify in the FROM clause of SELECT, ABORT, ROLLBACK, and DELETE statements.

You can use derived tables for these purposes:

- To avoid CREATE TABLE and DROP TABLE statements for storing retrieved information that is only needed for one request.
- To enable you to code more sophisticated, complex join queries.

For examples, see [Inner Joins](#) and [Outer Joins](#).

For more information about specific statements, see:

- [ABORT](#)
- [DELETE](#)
- [ROLLBACK](#)

## Rules and Restrictions for Derived Tables

The rules and restrictions are:

- The semantics of derived tables and views are identical, as are their restrictions.
- The scope of a derived table is limited to the level of the SELECT statement calling the subquery.
- A unique table correlation name is required for each derived table you create within a statement.
- You cannot specify these options in a derived table:
  - ORDER BY
  - WITH ... BY
- Fully-qualified column names are mandatory when you specify otherwise ambiguous column names in the select list of a subquery used to build a derived table.

This rule is consistent with the rules for creating a view.

In this query, the columns specified in the select list of the subquery that builds the derived table are not qualified:

```
SELECT *
FROM (SELECT *
      FROM tab1 AS t1, tab2 AS t2
      WHERE t1.col2 = t2.col3) AS derived_table;

*** Failure 3515 Duplication of column COL1 in creating a Table,
View, Macro or Trigger.
Statement# 1, Info =95
*** Total elapsed time was 1 second.
```

The query is correctly written as follows:

```
SELECT *
FROM (SELECT t1.col1, t1.col2, t1.col3, t2.col1,
            t2.col2, t2.col3
      FROM tab1 AS t1, tab2 AS t2
      WHERE t1.col2=t2.col3) AS derived_table (t1_col1,
            t1_col2, t1_col3, t2_col1, t2_col2, t2_col3);
```

- The subquery that defines the contents of a derived table cannot contain SELECT AND CONSUME statements.

## Examples

### Example: Using Derived Tables To Do Multilevel Aggregation

You cannot specify aggregates in a WHERE clause predicate. However, derived tables make such calculations easy to perform.

The following example shows how derived tables facilitate this.

In this example, the WHERE condition compares values from rows in the base table employee with the (in this case single) values of the column average\_salary in the derived table workers.

```
SELECT name, salary, average_salary
FROM (SELECT AVG(salary)
      FROM employee) AS workers (average_salary), employee
WHERE salary > average_salary
ORDER BY salary DESC;
```

The query returns this answer set:

name	salary	average_salary
Russel S	55,000.00	38,147.62
Watson L	56,000.00	38,147.62
Phan A	55,000.00	38,147.62
Aguilar J	45,000.00	38,147.62
Carter J	44,000.00	38,147.62

The following example is more elaborate, grouping the same information by dept\_no. In this example, the statement returns all salaries that exceed the respective department averages.

In this example, the derived table is a grouped table, providing a set of rows. The outer WHERE clause has an equality join between the department number (dept\_no) of the base table, employee, and the derived table.

```
SELECT name, salary, dept_no, average_salary
FROM (SELECT AVG(salary), dept_no
      FROM employee
      GROUP BY dept_no) AS workers (average_salary,dept_num),
      employee
WHERE salary > average_salary
AND    dept_num = dept_no
ORDER BY dept_no, salary DESC;
```

The answer set might look like this:

name	salary	dept_no	average_salary
Chin M	38,000.00	100	32,625.00
Moffit H	35,000.00	100	32,625.00
Russel S	55,000.00	300	47,666.67
Phan A	55,000.00	300	47,666.67
Watson L	56,000.00	500	38,285.71
Carter J	44,000.00	500	38,385.71
Smith T	42,000.00	500	38,285.71
Aguilar J	45,000.00	600	36,650.00

# WHERE Clause

## Purpose

Filters rows that satisfy a conditional expression in SELECT, DELETE, INSERT, UPDATE, ABORT, and ROLLBACK statements.

## Syntax

— WHERE *search\_condition* —

## Syntax Elements

### WHERE

An introduction to the search condition in the SELECT statement.

### *search\_condition*

A conditional search expression, also referred to as a conditional expression or predicate, that must be satisfied by the row or rows returned by the statement.

The arguments can be any valid SQL expression, including individual values, user-defined functions, DEFAULT functions, and subqueries, but the overall expression must be of a form that returns a single boolean (TRUE or FALSE) result.

Logical expressions include comparisons of numeric values, character strings, and partial string comparisons.

You cannot specify an expression that returns an ARRAY or VARRAY data type in a WHERE clause search condition. For more information about relational operators and expressions, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

Scalar subqueries are valid search conditions.

You cannot specify expressions that contain LOBs in a search condition unless you first cast them to another data type (for example, casting a BLOB to BYTE or VARBYTE or a CLOB to CHARACTER or VARCHAR) or pass them to a function whose result is not a LOB.

Subqueries in a search condition cannot specify SELECT AND CONSUME.

You can only specify a scalar UDF for *search\_condition* if it is invoked within an expression and returns a value expression.

If you specify the value for a row-level security constraint in a search condition, it must be expressed in its encoded form.

## ANSI Compliance

The WHERE clause is ANSI SQL:2011-compliant.

## Related Topics

For more information about the use of the WHERE clause, see:

- [HAVING Clause](#)
- [QUALIFY Clause](#)

## Usage Notes

### Aggregates in a WHERE Clause

Aggregates are not allowed in a WHERE clause except when the clause is part of a correlated subquery inside a HAVING clause, and the aggregate is applied on outer variables.

### Expressions Containing LOBs

You cannot specify expressions that contain LOBs in a search condition unless you first cast them to another type (for example, casting a BLOB to BYTE or VARBYTE or a CLOB to CHARACTER or VARCHAR) or pass them to a function whose result is not a LOB.

### UDT Comparisons

If a WHERE clause search condition specifies a UDT comparison, then:

- Both values specified in the comparison must be of the same UDT type.
- The UDT must have an ordering defined for it.

### ARRAY or VARRAY Data Type

You cannot specify an expression that evaluates to an ARRAY or VARRAY data type in a WHERE clause. For more information about ARRAY operators and expressions, see *Teradata Vantage™ Data Types and Literals*, B035-1143.

### Row-Level Security Constraint Values

If you specify the value for a row-level security constraint in a search condition, that value must be expressed in its encoded form.

### SELECT AND CONSUME Subqueries

Subqueries in a search condition cannot contain SELECT AND CONSUME.

## DEFAULT Function in a Search Condition

The DEFAULT function takes a single argument that identifies a relation column by name. The function evaluates to a value equal to the current default value for the column. For cases where the default value of the column is specified as a current built-in system function, the DEFAULT function evaluates to the current value of system variables at the time the statement is executed.

The following rules apply to the use of the DEFAULT function as part of the search condition within a WHERE clause.

- [DEFAULT Function Data Type](#)
- [DEFAULT Function Column Name](#)
- [DEFAULT Function Comparison Operators](#)

### DEFAULT Function Data Type

The resulting data type of the DEFAULT function is the data type of the constant or built-in function specified as the default unless the default is NULL. If the default is NULL, the resulting data type of the DEFAULT function is the same as the data type of the column or expression for which the default is being requested.

### DEFAULT Function Column Name

The DEFAULT function can be specified as DEFAULT or DEFAULT (*column\_name*). When you do not specify a column name, the system derives the column based on context. If the column context cannot be derived, the system returns an error.

You can specify a DEFAULT function with a column name argument within a predicate. The system evaluates the DEFAULT function to the default value of the column specified as its argument. Once the system has evaluated the DEFAULT function, it treats it like a constant in the predicate.

You can specify a DEFAULT function without a column name argument within a predicate only if there is one column specification and one DEFAULT function as the terms on each side of the comparison operator within the expression.

### DEFAULT Function Comparison Operators

Following existing comparison rules, a condition with a DEFAULT function used with comparison operators other than IS NULL or IS NOT NULL is unknown if the DEFAULT function evaluates to null.

A condition other than IS NULL or IS NOT NULL with a DEFAULT function compared with a null evaluates to unknown.

IF a DEFAULT function is used with ...	THEN the comparison is ...
IS NULL	<ul style="list-style-type: none"> <li>• TRUE if the default is null</li> <li>• Else it is FALSE</li> </ul>
IS NOT NULL	<ul style="list-style-type: none"> <li>• FALSE if the default is null</li> <li>• Else it is TRUE</li> </ul>

For more information about the DEFAULT function, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

## AND and OR Logical Operators

You can specify the AND and OR logical operators in the WHERE clause to create more complex search conditions. For example, this statement selects employees who are in department 100 AND who have either a college degree OR at least 5 years of experience:

```
SELECT *
FROM employee
WHERE dept = 100
AND (edlev >= 16
OR yrsexp >= 5);
```

## Using WHERE to Filter Character Data

You can use the WHERE clause to filter character data by searching for a text string.

For example, this statement is processed by searching the *employee* table for every row that satisfies the condition: the *job\_title* column contains the character string “analyst”. The *name* and *dept\_no* fields for those rows are then listed.

```
SELECT name, dept_no
FROM employee
WHERE UPPER (job_title) LIKE '%ANALYST%';
```

## Scalar UDFs

You can specify a scalar UDF in a WHERE clause search condition if the UDF returns a value expression. See [Example: Invoking an SQL UDF Within a Search Condition](#).



## Scalar Subqueries

You can specify a scalar subquery as an operand of a scalar predicate in the WHERE, HAVING, and QUALIFY clauses of a query.

## SAMPLE Clause in a Subquery

You cannot specify a SAMPLE clause in a subquery used as a WHERE clause predicate.

## WHERE Clause Defines Condition for Joining Table Rows

A WHERE clause can define a condition for joining table rows, typically for a situation in which the values in a common column must match in both tables.

The following statement, which asks for the name of each employee from the *employee* table and the location of the department for each employee from the *department* table, is processed by joining the *employee* and *department* tables on the WHERE clause equality condition `employee.dept_no=department.dept_no`.

```
SELECT name, loc
FROM employee, department
WHERE employee.dept_no = department.dept_no;
```

## EXISTS Quantifier

SQL supports the EXISTS ( $\exists$ ) logical quantifier for testing the result of subquery search conditions. See [Specifying Subqueries in Search Conditions](#). For more information about the EXISTS quantifier, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

If the subquery would return response rows, then the WHERE condition is considered to be satisfied. Specifying the NOT qualifier for the EXISTS predicate inverts the test.

A subquery used in a search condition does not return any rows. It returns a boolean value to indicate whether responses would or would not be returned.

The subquery can be correlated with an outer query.

## Specifying a Column PARTITION or PARTITION#L *n*

You can specify a system-derived column PARTITION or PARTITION#L *n* (*n* between 1 and 62) in a WHERE clause if the referenced table does not have a user-defined column named *partition* or PARTITION#L *n*, respectively. You can specify these system-derived columns for a table that does not

have partitioning, but the value returned for such a system-derived column is always 0, so the only reason to do so is to determine if a nonempty table is partitioned.

PARTITION is equivalent to a value expression where the expression is identical to the combined partitioning expression defined for the table with column references appropriately qualified as needed.

PARTITION#L *n*, where *n* ranges from 1 to 62, inclusive, is equivalent to a value expression where the expression is identical to the partitioning expression at the corresponding level (or 1 if this is a column partitioning level), or zero if the table is not partitioned.

Therefore, a query made on a partitioned table that specifies the predicate WHERE PARTITION <> combined\_partitioning\_expression should always return 0 rows. If any rows are returned, then they are not partitioned properly, and the table should be revalidated immediately.

## Use Consistent Predicate Domains

Specify consistent predicate domains in the WHERE clause search condition. For example, if min\_level is defined as NUMERIC(5,2), then min\_level values are compared with numeric values with greater precision in the WHERE condition, the result may not provide the accuracy you expect:

```
...
WHERE min_level = 133.002
...
```

## Join Efficiency and Indexes

The efficiency of a join operation depends on whether the WHERE condition uses values for columns on which primary, secondary, or multitable join indexes are defined.

If indexes are defined on the *dept\_no* columns in both the *employee* and *department* tables, specifying an equality condition between the values in each indexed column, as in the preceding example, allows the rows in the two tables to be matched using the values in both indexed columns.

Efficiency is increased if a primary index is defined on one or both of these columns. For example, define *dept\_no* as the unique primary index for the *department* table. This is not possible if one or both of the tables being joined is an ordinary NoPI table or a NoPI column-partitioned table. See [NoPI Tables, Column-Partitioned Tables, and WHERE Clause Search Conditions](#) for suggestions about how to work around this potential problem.

For all-AMP tactical queries against row-partitioned tables, you should specify a constraint on the partitioning column set in the WHERE clause.

If a query joins row-partitioned tables that are partitioned identically, using their common partitioning column set as a constraint enhances join performance still more if you also include an equality constraint between the partitioning columns of the joined tables.

## NoPI Tables, Column-Partitioned Tables, and WHERE Clause Search Conditions

NoPI and NoPI column-partitioned tables require a full-table scan to retrieve rows. Column partition elimination or row partition elimination may occur that make the scans less than full-table scans.

Because NoPI and NoPI column-partitioned tables support USIs and NUSIs, consider specifying USIs for conditions designed to retrieve single rows and NUSIs for conditions designed to retrieve row sets. For details, see [NoPI Tables and SELECT Statements](#).

Although you cannot specify join indexes in WHERE clause conditions, the Optimizer uses them if they exist and can be used to avoid a full-table scan. For information about how the Optimizer uses join indexes, see *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

Defining join indexes on a NoPI or NoPI column-partitioned table can slow down the loading of rows into the table.

## Unconstrained Joins

An unconstrained join is one for which a WHERE clause is not specified for the tables that are joined. See [Cross Join](#).

The result of an unconstrained join is a Cartesian product, which is rarely the desired result. A Cartesian product is the product of the number of rows in each table that is joined. An unconstrained join can produce a great many rows, returning a result that not only is not desired, but one that places a large performance burden on the system.

If a SELECT statement specifies correlation and real names (for example, correlation names in a WHERE clause and real names in the select list), the Optimizer may specify an unconstrained join, depending on what the conditions of the query are and how they are specified.

## Examples

### Example: Simple WHERE Clause Predicate

The following statement can be used to select the name and job title of every employee in Department 100. In this statement, the predicate `dept_no=100` is the conditional expression.

```
SELECT name, jobtitle
FROM employee
WHERE dept_no = 100;
```

### Example: Using the *table\_name.\** Syntax

The following statement returns only those rows from the employee table whose values in the empno column match the values in the mgr\_no column of the department table.

This example uses the *table\_name.\** form of the select list.

```
SELECT employee.*
FROM employee, department
WHERE employee.empno=department.mgr_no;
```

### Example: Delete from a Row-Partitioned Table Using the System-Defined PARTITION Column in the Search Condition

This statement deletes all orders in the orders table from partition 1 of the SLPP1 table orders.

```
DELETE FROM orders
WHERE orders.PARTITION=1;
```

If orders was a multilevel partitioned table, you might substitute PARTITION#L1 for PARTITION. However, this substitution would delete all orders in partition 1 of level 1 from the table, which is probably not the result you anticipated.

A DELETE statement for the same multilevel partitioned table specifying PARTITION in place of PARTITION#L1 would only delete the rows in combined partition 1.

### Example: INSERT ... SELECT and DELETE Operations from a Row-Partitioned Table Using the PARTITION Column as a Search Condition

This example performs an INSERT ... SELECT to copy orders from partitions 1 and 2 of the SLPP1 orders table into the old\_orders table using the system-defined PARTITION column as a search condition, and then deletes them from orders in the second part of the multistatement request.

```
INSERT INTO old_orders
SELECT *
FROM orders
WHERE orders.PARTITION IN (1,2)
;DELETE FROM orders
WHERE orders.PARTITION IN (1,2);
```

## Example: DEFAULT Function in a WHERE Clause Search Condition

The following set of examples shows the DEFAULT function in a WHERE clause search condition.

The examples assume these table definitions:

```
CREATE TABLE table14 (
  col1 INTEGER,
  col2 INTEGER DEFAULT 10,
  col3 INTEGER DEFAULT 20,
  col4 CHARACTER(60) );

CREATE TABLE table15 (
  col1 INTEGER ,
  col2 INTEGER NOT NULL,
  col3 INTEGER NOT NULL DEFAULT NULL,
  col4 INTEGER CHECK (col4 > 10) DEFAULT 9 );

CREATE TABLE table16 (
  col1 INTEGER,
  col2 INTEGER DEFAULT 10,
  col3 INTEGER DEFAULT 20,
  col4 CHARACTER(60) );
```

In this statement, the DEFAULT function evaluates to the default value of col2, which is 10.

```
SELECT col2, col3
FROM table16
WHERE col1 < DEFAULT(col2);
```

Following is an equivalent statement that uses an explicitly defined search condition in the WHERE clause.

```
SELECT col2, col3
FROM table16
WHERE col1 < 10;
```

You can specify a DEFAULT function with a column name argument within a predicate.

In this statement, the DEFAULT function evaluates to the default value of col3, which is 20.

```
SELECT col2, col3
FROM table16
WHERE col1 + 9 > DEFAULT(col3)+ 8;
```

Following is an equivalent statement that uses an explicitly defined search condition in the WHERE clause.

```
SELECT col2, col3
FROM table16
WHERE col1 + 9 > 20 + 8;
```

In this example, the DEFAULT function evaluates to the default value of col3, compares the returned value with the result of a subquery, and returns the result.

```
SELECT col2, col3
FROM table16
WHERE DEFAULT(col3) < ANY (SELECT col2
                           FROM table14);
```

You can specify a DEFAULT function without a column name argument in a comparison predicate when there is only one column specification and one DEFAULT function as the terms on the each side of the comparison operator within the expression.

In this example, there is only one column reference, and the DEFAULT function is compared directly with it; therefore, the DEFAULT function evaluates to the default value of col2.

```
SELECT col2, col3
FROM table16
WHERE col2 > DEFAULT;
```

The following two statements are semantically equivalent.

```
SELECT col2, col3
FROM table16
WHERE col2 > DEFAULT
AND    DEFAULT > col3;

SELECT col2, col3
FROM table16
WHERE col2 > DEFAULT(col2)
AND    DEFAULT(col3) > col3;
```

Following the existing comparison rules, a condition with a DEFAULT function used with comparison operators other than IS NULL or IS NOT NULL is unknown if the DEFAULT function evaluates to null.

Assume this table definition.

```
CREATE TABLE table17 (
  col1 INTEGER,
  col2 INTEGER NOT NULL DEFAULT 10,
  col3 INTEGER DEFAULT NULL );
```

In this example, the DEFAULT function evaluates to NULL, the predicate is UNKNOWN, and the WHERE condition is FALSE; therefore, the query returns no rows.

```
SELECT col1
FROM table17
WHERE col1 = DEFAULT;
```

In this example, the DEFAULT function evaluates to NULL. The first condition, DEFAULT(col3) > 5, is UNKNOWN, but the second condition is TRUE; therefore, the WHERE condition is TRUE and the statement returns all of the rows from table17.

```
SELECT col1, col2
FROM table17
WHERE DEFAULT(col3) > 5
OR    DEFAULT(col3) IS NULL;
```

In this example, the DEFAULT function evaluates to NULL, the predicate is UNKNOWN, and the WHERE condition is FALSE. Therefore, the statement does not return any rows.

```
SELECT col1
FROM table17
WHERE col2 < DEFAULT(col3) + 3;
```

In this example, the DEFAULT function evaluates to 10 and the first condition is TRUE. The second condition, DEFAULT(col2) IS NULL, is FALSE; therefore, the WHERE condition is TRUE and the statement returns all of the rows from table17.

```
SELECT col1, col2
FROM table17
WHERE DEFAULT(col2) > 5
OR    DEFAULT(col2) IS NULL;
```

## Example: Scalar Subquery in the WHERE Clause of a SELECT Statement

You can specify a scalar subquery as an operand of a scalar predicate in the WHERE, HAVING, and QUALIFY clauses of a query.

The following example specifies a scalar subquery (SELECT AVG(price)...) as an operand in its WHERE clause.

```
SELECT category, title, price
FROM movie_titles AS t2
WHERE (SELECT AVG(price)
      FROM movie_titles AS t1
      WHERE t1.category = t2.category) < (SELECT AVG(price)
      FROM movie_titles);
```

## Example: Invoking an SQL UDF Within a Search Condition

You can invoke an SQL UDF within a WHERE clause search condition if the UDF returns a value expression.

This example shows a correct invocation of the SQL UDF *value\_expression* within a WHERE clause.

```
SELECT test.value_expression(t1.a1, t2.a2)
FROM t1, t2
WHERE t1.b1 = t2.b2
AND test.value_expression(t1.a1, t2.a2) = 10;
```

## Example: Specifying a Row-Level Security Constraint in a Search Condition

If you have the required security credentials to read all rows in the table or have override select constraint privileges on the table, this statement returns emp\_name and group\_membership row-level security constraint value name and value code for all managers. If you do not have the required credentials or override privileges on the table, no rows are returned.

You cannot specify a row-level security constraint value name as the WHERE clause search condition. You must specify a value code. In this example, the value code 90 represents the value name manager.

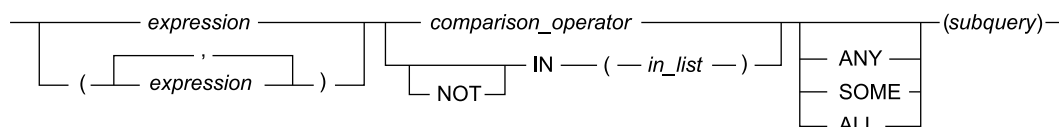
```
SELECT emp_name, group_membership
FROM emp_record
WHERE group_membership=90;
```

## Specifying Subqueries in Search Conditions

### Purpose

Permits a more sophisticated and detailed query of a database through the use of nested SELECT queries.

### Syntax: IN, NOT IN, ANY, SOME, and ALL Logical Predicates



### Syntax Elements

#### *expression*

SQL expression.



To specify multiple expressions, enclose the expressions in PARENTHESIS characters and separate each expression with a COMMA character.

### ***comparison\_operator***

SQL comparison operator.

See *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145 for information about the SQL comparison operators.

### **IN (*in\_list*)**

*expression* is either IN the parenthetically delimited list of values.

For information about the IN logical predicates and the valid expressions that you can specify in an *in\_list*, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

### **NOT IN (*in\_list*)**

*expression* is NOT IN the parenthetically delimited list of values.

For information about the NOT IN logical predicates and the valid expressions that you can specify in an *in\_list*, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

### **ANY**

The ANY logical quantifier.

For more information about the ANY logical quantifier, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

### **SOME**

The SOME logical quantifier.

For more information about the SOME logical quantifier, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

### **ALL**

The ALL logical quantifier.

For more information about the ALL logical quantifier, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

### ***subquery***

SQL subquery.

## **Syntax: EXISTS Logical Expression**

— EXISTS — *subquery* —

## **Syntax Elements**

### **EXISTS**

The EXISTS ( $\exists$ ) logical quantifier.

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

**subquery**

SQL subquery.

**ANSI Compliance**

Subqueries are ANSI SQL:2011-compliant.

**Related Topics**

For more information related to simple subqueries, see:

- [Correlated Subqueries](#)
- [Scalar Subqueries](#)
- [QUALIFY Clause](#)
- *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145

**Usage Notes****Specifying Subqueries**

The rules and restrictions are:

- You can specify up to 64 levels of nesting of subqueries. Each subquery can specify a maximum of 128 tables or single-table views.

This limit is established using the DBS Control field MaxParseTreeSegs. The value for MaxParseTreeSegs must be set to 256 or higher define an upper limit of 64 subquery nesting levels.

See *Teradata Vantage™ - Database Utilities*, B035-1102, for information about how to change the MaxParseTreeSegs setting in the DBS Control record.

- Subqueries in search conditions cannot contain SELECT AND CONSUME statements.
- Subqueries in search conditions cannot specify the TOP *n* operator.

**Examples****Example: Simple Subquery Using the IN Logical Predicate**

This example selects the names and department locations of those employees who report to manager Aguilar (whose employee number is 10007).

```
SELECT name, loc
FROM employee, department
WHERE employee.dept_no = department.dept_no
AND employee.dept_no IN (SELECT dept_no
```

```
FROM department
WHERE mgr_no = 10007);
```

## Example: Simple Subquery Using an AVG Aggregate Expression

The following SELECT statement finds every employee in the *employee* table with a salary that is greater than the average salary of all employees in the table.

```
SELECT name, dept_no, jobtitle, salary
FROM employee
WHERE salary > (SELECT AVG(salary)
                FROM employee)
ORDER BY name;
```

The statement returns this results table.

name ----	dept_no -----	jobtitle -----	salary -----
Aguilar J	600	Manager	45,000.00
Carter J	500	Engineer	44,000.00
Omura H	500	Programmer	40,000.00
Phan A	300	Vice President	55,000.00
Regan R	600	Purchaser	44,000.00
Russell S	300	President	65,000.00
Smith T	500	Engineer	42,000.00
Smith T	700	Manager	45,000.00
Watson L	500	Vice President	56,000.00

## Example: Simple Subquery Using the ALL Logical Predicate

The following SELECT statement retrieves the employees with the highest salary and the most years of experience:

```
SELECT emp_no, name, job_title, salary, yrs_exp
FROM employee
WHERE (salary,yrs_exp) >= ALL (SELECT salary,yrs_exp
                               FROM employee);
```

The query returns this many rows ...	WHEN this condition evaluates to TRUE ...
0	employees with the highest salary do not also have the most years experience.
1	there is only one employee with the highest salary and years experience.
multiple	there is more than one employee with the highest salary and years experience.

For the current data in the *employee* table, the result looks like this:

EmpNo	Name	JobTitle	Salary	YrsExp
-----	-----	-----	-----	-----
10018	Russell S	President	65,000.00	25

The result shows that one employee has both the highest salary and the most years of experience.

## Example: Subquery Using the SUM and COUNT Aggregate Functions

The following statement uses a nested ordered analytical function and a HAVING clause to find those items that appear in the top 1 percent of profitability in more than 20 stores:

```
SELECT item, COUNT(store)
FROM (SELECT store,item,profit,QUANTILE(100,profit) AS percentile
      FROM (SELECT ds.store, it.item, SUM(sales)-
              COUNT(sales) * it.item_cost AS profit
            FROM daily_sales AS ds, items AS it
            WHERE ds.item = it.item
            GROUP BY ds.store, it.item, it.item_cost) AS item_profit
      GROUP BY store, item, profit
      QUALIFY percentile = 0) AS top_one_percent
GROUP BY item
HAVING COUNT(store) >= 20;
```

The results of this query might look like this:

Item	Count(Store)
-----	-----
Chilean sea bass	99
Mackerel	149
Tako	121
Herring	120
Salmon	143

## Example: Subqueries Using the RANK Functions and QUALIFY Clauses

The following example uses nested RANK functions and QUALIFY clauses to report the top 100 items by profitability and the top items by revenue, matching those that appear in both lists using an OUTER JOIN:

```
SELECT *
FROM (SELECT item, profit, RANK(profit) AS profit_rank
      FROM item, sales
      QUALIFY profit_rank <= 100 AS p)
FULL OUTER JOIN
      (SELECT item, revenue, RANK(revenue) AS revenue_rank
      FROM item, sales
      QUALIFY revenue_rank <= 100 AS r)
ON p.item = r.item;
```

The results of this query might look something like the following table:

item ----	profit -----	profitrank -----	item ----	revenue -----	revenue_rank -----
Dress slacks	17804	74	Dress slacks	180211	56
Dress shirts	16319	68	?	?	?
Dresses	55888	82	Dresses	652312	77
Blouses	9849	48	Blouses	771849	92
?	?	?	Skirts	817811	55

## Correlated Subqueries

A subquery is correlated when it references columns of outer tables in an enclosing or containing outer query.

### Definition of a Correlated Subquery

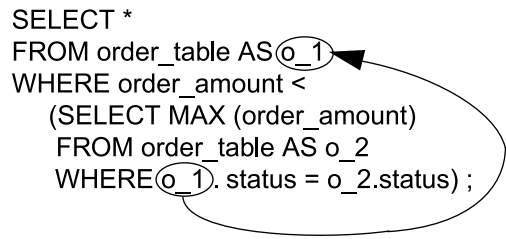
The expression *correlated subquery* comes from the explicit requirement for the use of correlation names in any correlated subquery in which the same table is referenced in both the internal and external query.

The reference from the inner query to its containing query is shown in this illustration.

```

SELECT *
FROM order_table AS o_1
WHERE order_amount <
  (SELECT MAX (order_amount)
   FROM order_table AS o_2
   WHERE o_1.status = o_2.status);

```



When the inner table references a foreign key column of a different table, the column must be fully qualified in the interior WHERE clause.

The effect of a correlated subquery is to provide an implicit loop function within any standard SQL DML statement, including ABORT, DELETE, INSERT, and UPDATE.

A reference to variables in an outer expression by a variable in an inner expression is called an *outer reference*.

IF a scalar subquery references tables in its containing query that are ...	THEN it is said to be ...
not referenced in its own FROM clause	correlated.
referenced in its own FROM clause	noncorrelated.

If a column reference is made from an inner query to an outer query, then it must be fully qualified with the appropriate table name from the FROM clause of the outer query.

## ANSI Compliance

Correlated subqueries are ANSI SQL:2011-compliant.

## Related Topics

For more information about correlated subqueries, see:

- [Specifying Subqueries in Search Conditions](#)
- [Scalar Subqueries](#)

## Usage Notes

### Using Correlated Subqueries

The rules and restrictions are:

- Correlated subqueries can specify combinations of equality and inequality between the tables of an outer query and the tables of a subquery.

For example, this SELECT statement specifies an equality condition in its subquery and specifies an inequality condition in the main query:

```

SELECT *
FROM table_1
WHERE x < ALL (SELECT y
                FROM table_2
                WHERE table_2.n = table_1.n);

```

- You can nest correlated subqueries to a depth of 64.

For example, this SELECT statement specifies three levels of depth:

```

SELECT *
FROM table_1
WHERE x = (SELECT y
            FROM table_2
            WHERE table_2.n = table_1.n
            AND t2.m = (SELECT y
                        FROM table_3
                        WHERE table_3.n = table_2.m
                        AND table_3.l = table_1.l));

```

- A correlated subquery can reference the same table as a table specified in the FROM clause of an outer query.

To differentiate between the references to the inner and outer tables, one of them must be renamed with a correlation name.

In the first example, the outer table is aliased, while in the second example, the inner table is aliased.

Also note that in the first example, the outer reference is a .n, while in the second example, the outer reference is table\_1.n.

```

SELECT *
FROM table_1 AS a
WHERE x < (SELECT AVG(table_1.x)
            FROM table_1
            WHERE table_1.n = a.n);

SELECT *
FROM table_1
WHERE x < (SELECT AVG(a.x)
            FROM table_1 AS a
            WHERE table_1.n = a.n);

```

- Correlated subqueries cannot contain SELECT AND CONSUME statements.
- Correlated subqueries cannot contain the TOP *n* operator.
- You can specify a SELECT in an outer query, a main query, or a subquery.

SELECT Specified	Description
Main query	Any table referenced in the main query should be specified in the FROM clause of the main query SELECT.
Subquery	Any table referenced in that subquery must be specified either in the FROM clause of that query or in some outer query. If the select expression list of the subquery specifies a column reference to a table in the FROM clause of an outer query, the column reference must be fully qualified. If the correlated condition specifies an unqualified column reference, then Teradata Database searches for the column starting with the tables and views in the current subquery toward the tables and views in the outermost query.

- Correlated subqueries can be specified in a FROM clause under the following conditions:

IF a table referenced in a subquery is identified in the FROM clause of ...	THEN the table reference is ...
a subquery	local and the subquery is noncorrelated.
an outer query	not local and the subquery is correlated.

The following rules apply to specifying a FROM clause in a correlated subquery.

- A FROM clause is required in any subquery specification.
- The FROM clause is not required in the outer query. However, you should always write your applications to specify a FROM clause because future releases might enforce the ANSI SQL:2011 rule that all tables referenced in a SELECT must be specified in a FROM clause. More importantly, specifying tables explicitly in the FROM clause helps to prevent errors in coding your queries because it enhances their clarity.

Teradata Database returns a warning message if you do not specify a referenced table in the FROM clause.

## EXISTS Quantifier and Correlated Subqueries

The EXISTS quantifier is supported as the predicate of a search condition in a WHERE clause.

EXISTS tests the result of the subquery. The subquery is performed for each row of the table specified in the outer query when the subquery is correlated.

If the subquery returns response rows, then its WHERE condition is satisfied.



## Outer References in Correlated Subqueries

Outer references behave as described in the following process. This process does not mirror the query plan generated by the Optimizer. Instead, this process describes how a correlated subquery works at a conceptual level.

1. For each row of an outer query, the values of the outer references in that row are used to evaluate the result of the inner subquery.
2. The inner subquery expression result is joined with its associated row from the outer query based on the specified join constraint for the inner query.

The semantics of correlated subqueries imply that an inner subquery expression is executed once for each row of its immediate outer query expression. The semantics do not guarantee that each iteration of the subquery produces a unique result.

The following example of this behavior uses the following simple employee table.

employee			
emp_no	emp_name	sex	age
PI	NN		
101	Friedrich	F	23
102	Harvey	M	47
103	Agrawal	M	65
104	Valdureiz	M	34
105	Cariño	F	39
106	Au	M	28
107	Takamoto	F	51
108	Ghazal	F	26

The following SELECT statement shows the behavior of a simple correlated subquery. Because the same table is referenced in both the inner and outer queries, both references are given correlation names for clarity, though only one of them (it makes no difference which) must be aliased.

```
SELECT *
FROM employee AS e1
WHERE age < (SELECT MAX(age)
             FROM employee AS e2
             WHERE e1.sex = e2.sex);
```

Here is the process:

1. Two copies of the table described earlier are generated, one as e1 and the other as e2.
2. Evaluation of the inner query requires data from the outer, containing, query.

The evaluation of the inner query becomes a set of virtual individual queries such as the following:

```
SELECT 101, 'Friedrich', 'F', 23
FROM employee AS e1
WHERE 23 < (SELECT MAX(age)
           FROM employee AS e2
           WHERE 'F' = e2.sex;

...

SELECT 108, 'Ghazal', 'F', 26
FROM employee as e1
WHERE 26 < (SELECT MAX(age)
           FROM employee AS e2
           WHERE 'F' = e2.sex;
```

3. The expanded individual queries once the subquery has been evaluated for each row in the inner query look like the following:

```
SELECT 101, 'Friedrich', F, 23
FROM employee AS e1
WHERE 23 < 51;

SELECT 102, 'Harvey', M, 47
FROM employee AS e1
WHERE 47 < 65;

SELECT 103, 'Agrawal', M, 65
FROM employee AS e1
WHERE 65 < 65;

SELECT 104, 'Valduries', M, 34
FROM employee AS e1
WHERE 34 < 65;

SELECT 105, 'Cariño', F, 39
FROM employee AS e1
WHERE 39 < 51;

SELECT 106, 'Au', M, 28
FROM employee AS e1
WHERE 28 < 65;

SELECT 107, 'Takamoto', F, 51
```

```

FROM employee AS e1
WHERE 51 < 51;

SELECT 108, 'Ghazal', F, 26
FROM employee AS e1
WHERE 26 < 51;

```

4. Teradata Database performs the same evaluations for each row in the outer query.
5. Employee 103, Agrawal, is eliminated from the result because his age is not less than the maximum male age in the table. Similarly, employee 107, Takamoto, is eliminated because her age is not less than the maximum female age in the table.

The final result is reported in the following table.

emp_no	emp_name	sex	age
-----	-----	---	---
101	Friedrich	F	23
102	Harvey	M	47
104	Valduries	M	34
105	Cariño	F	39
106	Au	M	28
108	Ghazal	F	26

## Guidelines for Coding Noncorrelated Subqueries

To perform a subquery as an noncorrelated subquery and not as a correlated subquery, observe the following guidelines.

- For each main and subquery you code, ensure that its scope is local by specifying a FROM clause that includes all the tables it references.
- Instead of specifying join operations in the main queries of DELETE and UPDATE statements, restrict them to subqueries.

## Comparing Correlated and Noncorrelated Subqueries

In the following SELECT statement, no entry in predicate\_2 of the subquery references any entry in table\_list\_1 of the main query; therefore, the subquery is completely self-contained and is not correlated. A self-contained subquery is also referred to as a local subquery.

```

SELECT column_list_1
FROM table_list_1
WHERE predicate_1 (SELECT column_list_2
                   FROM table_list_2
                   WHERE predicate_2);

```

The relationship between inner and outer query predicates is referred to as their correlation. The subquery in the previous statement is said to be uncorrelated, or noncorrelated, because it does not reference tables in the outer query. Because its subquery is local, the request restricts the number of its iterations to one. The results of the query are then joined with the results of the query made by the outer SELECT statement.

Correlated subqueries perform the subquery in parentheses once for each result row of the outer query. When a subquery references tables in the outer query that are not referenced within itself, the subquery is said to be correlated because its result is directly correlated with the outer query. A correlated subquery performs once for each value from its containing outer query. It does not necessarily produce a unique result for each of those iterations.

The following example shows the behavior of an SQL request that contains a correlated subquery.

Assume that table\_1 has columns col\_1 and col\_2, while table\_2 has columns col\_3 and col\_4. The following four rows exist in the two tables.

table_1		table_2		
col_1	col_2		col_3	col_4
100	1		100	1
50	1		50	1
20	2		20	2
40	2		40	2

The following SELECT statement uses a correlated subquery in its WHERE clause to query table\_1 and table\_2.

```
SELECT *
FROM table_1
WHERE col_1 IN (SELECT MAX(col_3)
                FROM table_2
                WHERE table_1.col_2=table_2.col_4);
```

The statement returns two response rows because the subquery is performed four times: once for each row in table\_1.

The result contains only 2 response rows because of the MAX(col\_3) aggregation constraint and two of the subquery executions return a response row where col\_1 is not in the result.

The two rows returned are these:

- col\_1=100, col\_2=1
- col\_1=40, col\_2=2.

The four executions of the subquery return the following response rows:

col_3	col_4
100	1
100	1
40	2
40	2

Only the first and fourth rows of table\_1 have a value for col\_1 in this result set. If you had not specified the MAX aggregate function, then all four rows of table\_1 would have been returned.

## Examples

### Example: Combining Equality and Inequality Conditions in a Correlated Subquery

You can use correlated subqueries to specify a combination of equality and inequality constraints with the subquery.

For example, to select the names of all students who are younger than all students at the same grade, you can perform the following query:

```
SELECT name
FROM student st1
WHERE age < ALL (SELECT age
                  FROM student st2
                  WHERE st1.grade = st2.grade
                  AND st1.stno <> st2.stno);
```

### Example: Using SELECT COUNT(\*) in a Correlated Subquery

Select the names of the publishers whose book count values in the library match the actual count of books.

```
SELECT pubname, bookcount
FROM library
WHERE (bookcount, pubnum) IN (SELECT COUNT(*), book.pubnum
                              FROM book
                              GROUP BY pubnum);
```

If the book count for a publisher in the library is zero, then the name of that publisher is not returned because no row is returned by the subquery for this publisher.

Note that the result data type for a COUNT operation is different for ANSI and Teradata session modes, as described by the following table. For more information, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

IN this session mode ...	THE data type of the result for a COUNT operation is ...
ANSI	DECIMAL( $p,0$ ) where $p$ represents the precision of the number. <ul style="list-style-type: none"> <li>If the DBS Control field MaxDecimal is set to any of the following values, then the value of <math>p</math> is 15.               <ul style="list-style-type: none"> <li>0</li> <li>15</li> <li>18</li> </ul> </li> <li>If the DBS Control field MaxDecimal is set to 31, then the value of <math>p</math> is also 31.</li> <li>If the DBS Control field MaxDecimal is set to 38, then the value of <math>p</math> is also 38.</li> </ul>
Teradata	INTEGER

Another equivalent SELECT statement uses two noncorrelated subqueries to return the same answer set as follows:

```
SELECT pub_name, book_count
FROM library
WHERE (book_count, pub_num) IN (SELECT COUNT(*), pub_num
                                FROM book
                                GROUP BY pub_num)
    OR NOT IN (SELECT book.pub_num
              FROM book
              GROUP BY pub_num)
    AND book_count = 0;
```

The following SELECT statement, which is less complicated and more elegant, uses a correlated subquery to return the same correct answer as the previous query.

```
SELECT pub_name, book_count
FROM library
WHERE book_count IN (SELECT count(*)
                    FROM book
                    WHERE book.pub_num = library.pub_num);
```

## Scalar Subqueries

### Definition of a Scalar Subquery

A scalar subquery is a subquery expression that can return a maximum of one value.

As with other types of subqueries, there are two types of scalar subqueries:

- Correlated
- Noncorrelated

A correlated scalar subquery returns a single value for each row of its correlated outer table set.

A noncorrelated scalar subquery returns a single value to its containing query.

You can think of a correlated scalar subquery as an extended column of the outer table set to which it is correlated. In the same way, you can think of a noncorrelated scalar subquery as a parameterized value.

Accordingly, you can specify a correlated scalar subquery in a statement in the same way as a column, while you can specify a noncorrelated scalar subquery in the same way as you would a parameterized value wherever a column or parameterized value is allowed. This includes specifying a scalar subquery as a standalone expression in various clauses of a DML statement, or specified within an expression.

## ANSI Compliance

Scalar subqueries are compliant with the ANSI SQL:2011 standard.

## Specifying Scalar Subqueries in SQL Statements

The rules and restrictions are:

- A scalar subquery can be correlated to one or more outer tables.
- A scalar subquery can have nested levels of subqueries, which can be either scalar or nonscalar.
- A scalar subquery can be either correlated or noncorrelated.
- A scalar subquery can specify joins, both inner and outer, aggregation, and other similar SQL DML statement attributes.
- A scalar subquery specified as the right hand side operand of a predicate can return more than one column, but a scalar subquery that is specified elsewhere in the statement or that is specified within an expression must specify exactly one column or column expression in its select list.
- A scalar subquery cannot be specified in an SQL DML statement for which scalar subqueries are not supported.

The following DML statements support scalar subqueries.

- ABORT
- DELETE
- INSERT
- ROLLBACK
- SELECT

The recursive statement of a recursive query and the SELECT AND CONSUME statement do not support scalar subqueries.

- UPDATE
- UPDATE (Upsert Form)

You cannot specify a correlated scalar subquery in the condition of a SET or WHERE clause for the Upsert form of UPDATE.

- You cannot specify a scalar subquery in a statement that supports scalar subqueries, but violates a usage restriction.
- The cardinality of a scalar subquery result cannot be greater than one.
- If a scalar subquery returns 0 rows, then it evaluates to null.

For example, the following WHERE clause evaluates to TRUE for those rows in salestable that do not match any rows in prodttable on prod\_no:

```
SELECT ... FROM salestable AS s
WHERE (SELECT 1
      FROM prodttable AS p
      WHERE s.prod_no = p.prod_no) IS NULL;
```

- You cannot specify a scalar subquery in a DDL statement with the following exceptions.
  - CREATE MACRO
  - CREATE RECURSIVE VIEW

You can specify a scalar subquery in the seed statement of a recursive view definition, but not in its recursive statement.

- CREATE TABLE ... AS
- CREATE TRIGGER
- CREATE VIEW
- REPLACE MACRO
- REPLACE TRIGGER
- REPLACE VIEW

You cannot specify a scalar subquery in a join index definition.

- You cannot specify a scalar subquery in the recursive statement of a recursive statement.
- You cannot specify a scalar subquery in a SELECT AND CONSUME statement.
- You cannot specify a correlated scalar subquery in the condition of a SET or WHERE clause for an UPDATE (Upsert Form) statement.
- You cannot specify a noncorrelated scalar subquery as a value that is assigned to an identity column in the value list of a simple INSERT statement.
- You cannot specify a noncorrelated scalar subquery as a value in the value list of a simple INSERT in the body of a row trigger.
- You cannot specify a correlated scalar subquery in the ON clause of a MERGE statement.
- You cannot specify a correlated scalar subquery in a value list.

For example, the following INSERT statement returns an error:



```
INSERT INTO t1 (1,2,3 (SELECT d2
                        FROM t2
                        WHERE a2=t1.a1));
```

A value list is a list of simple values that can be either constants or parameterized values. Therefore you can only specify a noncorrelated scalar subquery that returns a value of a primitive data type in a value list. You cannot specify a scalar subquery that returns a column of UDT or LOB data in a value list.

- You can only use a forward reference to an aliased scalar subquery (SSQ) expression in the top level of the referencing expression. That is, the aliased SSQ reference cannot be nested in another SELECT block that is defined within the referencing expression. For example, an SSQ expression can contain a forward reference to an aliased SSQ expression in the select list, a WHERE clause, a GROUP BY clause, or a HAVING clause:

```
SELECT (SELECT ssq2+a1 FROM t1) AS ssq1,
       (SELECT MAX(b3) FROM t3) AS ssq2
FROM t2;
```

You cannot nest the forward reference in another query block that is defined within the referencing expression. For example, the following statement, which includes a forward reference to ssq2 from the SSQ expression ssq1, is nested in a SELECT block within ssq1, and returns an error indicating incorrect use of a subquery:

```
SELECT (SELECT (SEL ssq2) + a1 FROM t1) AS ssq1,
       (SELECT MAX(b3) FROM t3) AS ssq2
FROM t2;
```

To avoid the error, switch the two expressions in the select-list as shown below:

```
SELECT (SELECT MAX(b3) FROM t3) AS ssq2,
       (SELECT (SEL ssq2) + a1 FROM t1) AS ssq1
FROM t2;
```

## Related Topics

For more information about scalar subqueries, see [Specifying Subqueries in Search Conditions](#).

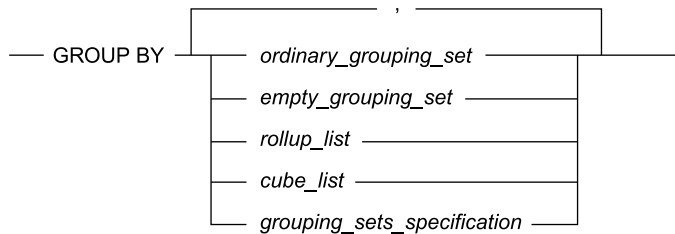
## GROUP BY Clause

### Purpose

Groups result rows by the values in one or more columns or by extended GROUP BY operations on specified column expressions.

For information about the GROUP BY TIME clause, see *Teradata Vantage™ Time Series Tables and Operations*, B035-1208.

## Syntax



## Syntax Elements

### GROUP BY

Reference to one or more expressions in the select expression list.

#### *ordinary\_grouping\_set*

Column expression by which the rows returned by the statement are grouped.

You cannot specify BLOB, CLOB, Period, ARRAY, VARRAY, XML, or JSON columns in the grouping expression.

*ordinary\_grouping\_set* can include:

- *column\_name*
- *column\_position*
- *column\_expression*

For the definitions of these expressions, see [Ordinary Grouping Set Expressions](#).

#### *empty\_grouping\_set*

Contiguous LEFT PARENTHESIS, RIGHT PARENTHESIS pair without an argument. You use this syntax to request a grand total, that is, a summation of all the individual group totals, not a summation of the nonaggregate data.

The term grand total refers to .

#### *rollup\_list*

ROLLUP expression that reports result rows in a single dimension with one or more levels of detail.

For more information, see [ROLLUP Grouping Set Option](#).

The expression cannot group result rows that have a BLOB, CLOB, ARRAY, or VARRAY type.

#### *cube\_list*

A CUBE expression that reports result rows in multiple dimensions with one or more levels of detail. For more information, see [CUBE Grouping Set Option](#).

The expression cannot group result rows that have a BLOB, CLOB, ARRAY, or VARRAY type.

#### *grouping\_sets\_specification*

A GROUPING SETS expression that reports result rows in one of two ways:

- As a single dimension, but without a full ROLLUP.

- As multiple dimensions, but without a full CUBE.

For more information, see [GROUPING SETS Option](#).

## ANSI Compliance

The GROUP BY clause is ANSI SQL:2011-compliant with extensions.

## Usage Notes

### GROUP BY Clause Terminology

A GROUP BY clause is said to be *simple* if it does not contain any of the following elements:

- *rollup\_list*
- *cube\_list*
- *grouping\_sets\_specification*

A GROUP BY clause is said to be *primitive* if it does not contain any of the following elements:

- *rollup\_list*
- *cube\_list*
- *grouping\_sets\_specification*
- a *grouping\_expression* and an *empty\_grouping\_set*.

See [GROUPING SETS Option](#) for definitions of *rollup\_list*, *cube\_list*, *grouping\_sets\_specification*, and *empty\_grouping\_set*.

See the following topics for examples of non-simple, non-primitive GROUP BY clauses:

- [ROLLUP Grouping Set Option](#)
- [CUBE Grouping Set Option](#)
- [GROUPING SETS Option](#)

### Ordinary Grouping Set Expressions

The following table provides the definitions for the valid ordinary grouping set expressions you can specify in a GROUP BY clause.

Ordinary Grouping Set Expression	Definition
<i>column_name</i>	Set of column names drawn from the list of tables specified in the FROM clause of the SELECT statement that is used in the GROUP BY clause to specify the columns by which data is to be grouped. The maximum number of columns you can specify is 64.

Ordinary Grouping Set Expression	Definition
	<p>You cannot include LOB columns in the grouping expression.</p> <p>You can specify a <i>column_name_alias</i> instead of <i>column_name</i> as long as it does not have the same name as a physical column in the table definition. In this case, you must specify <i>column_position</i>, not <i>column_name_alias</i>.</p>
<i>column_position</i>	<p>Sequential numeric position of columns within the <i>column_list</i> clause of the SELECT statement that is used in the GROUP BY clause to specify the order by which data is to be grouped.</p> <p>The value you specify must be a positive constant integer literal with a value between 1 and the number of columns specified in the select list, inclusive. Note that Teradata Database treats macro and procedure parameters as expressions, not as the specification of a column position.</p> <p>You cannot include LOB columns in the grouping expression.</p> <p>This is a Teradata extension to the ANSI SQL:2011 standard.</p>
<i>column_expression</i>	<p>List of valid SQL expressions specified for the GROUP BY clause.</p> <p>You can specify <i>column_name</i>, <i>column_position</i>, and <i>expression</i> either as single entries or as a list.</p> <p>You can specify a scalar subquery as an ordinary grouping set expression.</p> <p>You can also specify a scalar UDF as an ordinary grouping set expression.</p> <p>You cannot include LOB columns in the ordinary grouping set.</p> <p>Use of <i>column_expression</i> is a Teradata extension to the ANSI SQL:2011 standard.</p>

## GROUP BY and Aggregate Operations

For aggregate operations, including SUM, AVERAGE, MAX, MIN, and COUNT, GROUP BY can be used to return a summary row for each group.

Aggregate operators can be used only in the SELECT expression list or in the optional HAVING clause.

## GROUP BY and Nonaggregate Operations

Nonaggregated variables in SELECT, ORDER BY, and HAVING need to appear in the group by list.

All nonaggregate groups in a SELECT expression list or HAVING expression list must be included in the GROUP BY clause.

## GROUP BY and DISTINCT

For cases when the GROUP BY is semantically equivalent to DISTINCT, the optimizer makes a cost-based decision to eliminate duplicates either by way of a combined sort and duplicate elimination or an aggregation step.

## Comparing GROUP BY and Correlated Subqueries Using a Scalar UDF

You can specify a correlated subquery using a scalar UDF in the same way that you would specify a column name or parameterized value in the GROUP BY clause.

## GROUP BY and Built-in Ordered Analytic Functions

For built-in ordered analytic functions specific to Teradata Database, such as CSUM and MAVG, GROUP BY determines the partitions over which the function executes. For an example, see [Example: Specifying GROUP BY Using an Ordered Analytic Function](#).

For window functions, such as SUM and AVG, GROUP BY collapses all rows with the same value for the group-by columns into a single row. The GROUP BY clause must include all the columns specified in the:

- Select list of the SELECT clause
- Window functions in the select list of a SELECT clause
- Window functions in the search condition of a QUALIFY clause
- The condition in a RESET WHEN clause

For examples and more information on GROUP BY and ordered analytical functions, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

## BLOB or CLOB Columns Not Allowed in a GROUP BY Expression

You cannot specify BLOB or CLOB columns in a grouping expression, rollup list, cube list, or grouping sets specification.

## GROUP BY and Recursive Queries

You cannot specify a GROUP BY clause in a recursive statement of a recursive query. However, you can specify a GROUP BY clause in a nonrecursive seed statement in a recursive query.

## ORDER BY and GROUP BY

If you specify an ORDER BY clause, then any group contained in the ORDER BY clause must also be included in the GROUP BY clause.

## WHERE, GROUP BY, and HAVING Clause Evaluation

WHERE, GROUP BY, and HAVING clauses in a SELECT statement are evaluated in this order:

- WHERE
- GROUP BY
- HAVING

## Reason for Unexpected Row Length Errors: Sorting Rows for Grouping

Before performing the sort operation that groups the rows to be returned to the requestor, Teradata Database creates a sort key and appends it to the rows to be sorted. If the length of this temporary data structure exceeds the system row length limit of 64 KB, the operation returns an error to the requestor. Depending on the situation, the message text is one of the following:

- A data row is too long.
- Maximum row length exceeded in *database\_object\_name*.

For explanations of these messages, see *Teradata Vantage™ - Database Messages*, B035-1096.

The BYNET only looks at the first 4096 bytes of the sort key created to sort the specified fields, so if the field the sort key is based on is greater than 4096 bytes, the key is truncated and the data might not be returned in the desired order.

## How Teradata Database Resolves Multiple Grouping Sets Specifications

Teradata Database resolves multiple grouping sets specifications by concatenating pairwise the individual elements of the different sets. For information about grouping sets specifications, see:

- [CUBE Grouping Set Option](#)
- [GROUPING SETS Option](#)
- [ROLLUP Grouping Set Option](#)

This is trivial in the case of a simple grouping specification because it is a set containing only one element. However, when applied to more complicated specifications that contain multiple grouping specification elements, the resolution is more complicated.

For example, the following two GROUP BY clauses are semantically identical:

```
GROUP BY GROUPING SETS ((A,B), (C)), GROUPING SETS ((X,Y),())
GROUP BY GROUPING SETS ((A,B,X,Y), (A,B), (C,X,Y), (C))
```

## Example: Semantically Identical Grouping Sets Specifications and Their Resolution

The following three queries return the same results set because they are semantically identical.

```
SELECT y,m,r, SUM(u)
FROM test
GROUP BY CUBE(y,m), CUBE(r)
```

```
ORDER BY 1,2,3;

SELECT y,m,r,SUM(u)
FROM test
GROUP BY CUBE(y,m,r)
ORDER BY 1,2,3;

SELECT y,m,r,SUM(u)
FROM test
GROUP BY GROUPING SETS(y,()), GROUPING SETS(m,()),
        GROUPING SETS(r,())
ORDER BY 1,2,3;
```

The following three queries return the same results set because they are semantically identical.

```
SELECT y, m, r, s, SUM(u)
FROM test
GROUP BY ROLLUP(y,m),ROLLUP(r,s)
ORDER BY 1,2,3,4;

SELECT y, m, r, s, SUM(u)
FROM test
GROUP BY GROUPING SETS((y, m),(),y),ROLLUP(r,s)
ORDER BY 1,2,3,4;

SELECT y, m, r, s, SUM(u)
FROM test
GROUP BY GROUPING SETS((y,m),(),y),GROUPING SETS((),r,(r,s))
ORDER BY 1,2,3,4;
```

## Examples

### Example: Simple GROUP BY Operation

Generate a report of salary totals by department, the result might look something like the report that follows.

```
SELECT dept_no, SUM(salary)
FROM employee
GROUP BY dept_no;
```

The result might be similar to the report that follows.

dept_no -----	SUM(salary) -----
100	180,500.00
300	143,000.00
500	268,000.00
600	146,600.00
700	113,000.00

### Example: Specifying a GROUP BY Clause on Nonaggregate Expressions When the Select List Includes an Aggregate

If you specify an aggregate in the select expression list of a query, then you must also specify a GROUP BY clause that includes all nonaggregate expressions from the select list. Otherwise, the system returns the following message.

Selected non-aggregate values must be part of the associated group.

The system returns error message 3504 whenever an aggregate query includes a nonaggregate expression in its SELECT list, WHERE clause, ORDER BY clause, or HAVING clause, but not in a GROUP BY clause.

The system also returns this error when ORDER BY and WITH clauses specify aggregates, but the select list for the query does not.

For example, Teradata Database aborts the following query because it does not specify a GROUP BY clause that groups on the only nonaggregate expression in the select list, which is *department\_number*:

```
SELECT department_number, SUM(salary_amount)
FROM employee
WHERE department_number IN (100, 200, 300);
```

To work as intended, the query must be rewritten with an appropriate GROUP BY clause:

```
SELECT department_number, SUM(salary_amount)
FROM employee
WHERE department_number IN (100, 200, 300)
GROUP BY department_number;
```

The following statement aborts and returns an error because it does not specify all of the nonaggregate columns from the select list in its GROUP BY clause.



```
SELECT employee.dept_no, department.dept_name, AVG(salary)
FROM employee, department
WHERE employee.dept_no = department.dept_no
GROUP BY employee.dept_no;
```

In this case, the qualified nonaggregate column *department.dept\_name* is missing from the GROUP BY clause.

The following statement aborts and returns an error because the nonaggregate grouping column specified in the ORDER BY clause is not also specified in the GROUP BY clause.

```
SELECT employee.dept_no, AVG(salary)
FROM employee, department
WHERE employee.dept_no = department.dept_no
ORDER BY department.dept_name
GROUP BY employee.dept_no;
```

The following statement, based on the table definitions shown, aborts and returns an error because the aggregate query includes a nonaggregate expression, *d1*, in its HAVING clause, but not in the GROUP BY clause.

```
CREATE TABLE t1(a1 int, b1 int, c1 int, d1 int);
CREATE TABLE t2(a2 int, b2 int, c2 int, d2 int);
SELECT min(a1) as i, max(b1) as j from t1
GROUP BY c1
HAVING 30 >= (sel count(*) from t2 where t1.d1=5);
```

The correct form of the query includes the nonaggregate expression, *d1*, in its HAVING clause and in the GROUP BY clause.

```
SELECT min(a1) as i, max(b1) as j from t1
GROUP BY c1, d1
HAVING 30 >= (sel count(*) from t2 where t1.d1=5);
```

## Example: Specifying GROUP BY Using an Ordered Analytic Function

The following statement specifies a GROUP BY clause with an ordered analytical function to generate report breaks where the function resets and computes a new value for the next grouping.

The example groups all items into percentile by profitability for each store and then returns only the items of interest, which, in this case, are the lowest percentile for each store.

```
SELECT store, item, profit, QUANTILE(100, profit) AS percentile
FROM (SELECT items.item, SUM(sales) -
      (COUNT(sales)*items.item_cost) AS profit
      FROM daily_sales, items
```

```

WHERE daily_sales.item = items.item
GROUP BY items.item,items.itemcost) AS item_profit
GROUP BY store, item, profit, percentile
QUALIFY percentile = 99;

```

The result of this query looks like the following table:

store -----	item ----	profit -----	percentile -----
Eastside	Golf balls	100.19	99
Westside	Tennis balls	-110.00	99
Central	Bowling balls	-986.81	99
South	Codfish balls	- 1,891.89	99
North	Bocce balls	1,180.88	99

## Example: SELECT Statement With a Scalar Subquery in Its GROUP BY Clause

The following example specifies a scalar subquery in its GROUP BY clause:

```

SELECT sale_date, SUM(amount)
FROM sales_table AS s
GROUP BY sale_date, (SELECT prod_name
                     FROM prod_table AS p
                     WHERE p.prod_no = s.prod_no);

```

## Example: GROUP BY and PERIOD Value Expressions

The following example shows how the GROUP BY clause operates on PERIOD value expressions, where *period\_of\_stay* is the PERIOD value expression.

```

SELECT emp_no, period_of_stay
FROM employee
GROUP BY emp_no, period_of_stay;

```

## Related Topics

For more information related to the GROUP BY clause, see:

- [How Teradata Database Resolves Multiple Grouping Sets Specifications](#)

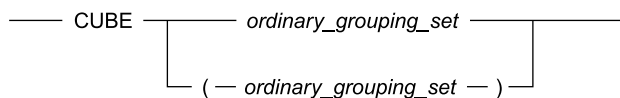
- [CUBE Grouping Set Option](#)
- [GROUPING SETS Option](#)
- [ROLLUP Grouping Set Option](#)
- [ORDER BY Clause](#)

## CUBE Grouping Set Option

### Purpose

Analyzes data by grouping it into multiple dimensions.

### Syntax



### Syntax Elements

#### *ordinary\_grouping\_set*

one or more expressions used to group cubed report rows across multiple dimensions.

All ordinary grouping sets must make explicit column references. You cannot specify column positions with CUBE.

You cannot specify more than 8 columns in the ordinary grouping sets.

This limit is actually on the number of table dimensions that can be cubed. The value is much smaller than the limits for the ROLLUP and GROUPING SETS options because it specifies a limit of 2<sup>8</sup>, or 256 different *combinations* of columns, not just columns.

You cannot reference columns that have a LOB data type unless they are first CAST to another type or passed to a function whose result is not a LOB. For example, casting a BLOB to BYTE or VARBYTE or a CLOB to CHARACTER or VARCHAR.

### ANSI Compliance

CUBE is ANSI SQL:2011 compliant.

### Rules and Restrictions for CUBE

You cannot specify a SAMPLE clause in a query that specifies the CUBE option in a GROUP BY clause, with the exception of a query where the CUBE option in the GROUP BY clause appears in a derived table or view and the SAMPLE clause appears in the outer query.

## How CUBE Summarizes Data

Given  $n$  dimensions, CUBE generates  $2n$  different kinds of groups. Teradata Database reports each group as a single row.

For example, with 2 dimensions, CUBE generates 4 groups as follows:

group number	dimension 1	dimension 2
1	X	X
2	X	
3		X
4		

With 3 dimensions, CUBE generates 8 groups as follows:

group number	dimension 1	dimension 2	dimension 3
1	X	X	X
2	X	X	
3	X		X
4	X		
5		X	X
6		X	
7			X
8			

## Example: CUBE Grouping

Suppose you have the following *sales\_view* table data:

sales_view						
PID	cost	sale	margin	state	county	city
1	38350	50150	11800	CA	Los Angeles	Long Beach
1	63375	82875	19500	CA	San Diego	San Diego
1	46800	61200	14400	CA	Los Angeles	Avalon
1	40625	53125	12500	CA	Los Angeles	Long Beach
2	20000	24750	4750	CA	Los Angeles	Long Beach

sales_view						
2	4800	5940	1140	CA	San Diego	Santee
1	57600	71280	13680	CA	San Diego	San Diego

You are interested in determining the effect of *county* and *PID* on sale price.

The following SELECT statement analyzes the effect of *county* and *PID* on sale price using CUBE in its GROUP BY clause:

```
SELECT pid, county, SUM(sale)
FROM sales_view
GROUP BY CUBE (pid, county);
```

The query reports the following data, where the QUESTION MARK character indicates a null:

PID	County	Sum(sale)
---	-----	-----
2	Los Angeles	24750
1	Los Angeles	164475
2	San Diego	5940
1	San Diego	154155
2	?	30690
1	?	381630
?	Los Angeles	189225
?	San Diego	160095
?	?	349320

This query computes four different levels of grouping over the dimensions of *county* and *PID* by:

- *county* and *PID* (groups 1 through 4).
- *PID* only (groups 5 and 6).
- *county* only (groups 7 and 8).
- Nothing (group 9), which aggregates sales over all counties and PIDs.

Notice that nulls are used to represent the empty set, not missing information. The nulls mean that information is not reported at the grouping level represented, not that the information is missing from the *sales\_view* base table.

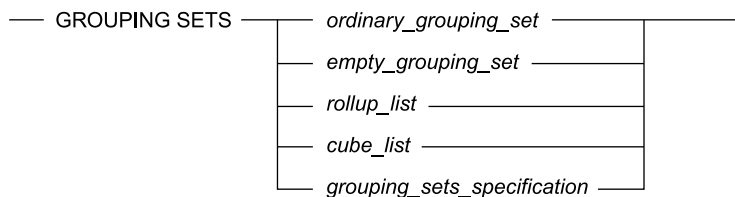
# GROUPING SETS Option

## Purpose

Analyzes data in one of two possible ways:

- In a single dimension without performing a full ROLLUP.
- In multiple dimensions without performing a full CUBE.

## Syntax



## Syntax Elements

### *ordinary\_grouping\_set*

One or more expressions used to group cubed report rows across multiple dimensions.

All ordinary grouping sets must make explicit column references. You cannot specify column positions with GROUPING SETS.

You cannot specify more than 190 columns in the ordinary grouping sets.

You cannot reference columns that have a BLOB or CLOB data type unless they are first CAST to another type or passed to a function whose result is not a LOB. For example, casting a BLOB to a BYTE or VARBYTE type, or casting a CLOB to a CHARACTER or VARCHAR type.

The expression cannot group result rows that have a BLOB, CLOB, ARRAY, or VARRAY type.

### *empty\_grouping\_set*

A contiguous LEFT PARENTHESIS, RIGHT PARENTHESIS pair with no argument. This syntax is used to provide a grand total.

The term *grand total* here refers to a summation of all the individual group totals, not a summation of the nonaggregate data.

### *rollup\_list*

A standard grouping expression set specified with ROLLUP. If you specify more than one expression, the expression list must be enclosed by parentheses.

The expression cannot group result rows that have a BLOB, CLOB, ARRAY, or VARRAY type.

See [ROLLUP Grouping Set Option](#) for more information.

***cube\_list***

a standard grouping expression set specified with CUBE. If you specify more than one expression, the expression list must be enclosed by parentheses.

The expression cannot group result rows that have a BLOB, CLOB, ARRAY, or VARRAY type.

See [CUBE Grouping Set Option](#) for more information.

***grouping\_sets\_specification***

Any of the following:

- *ordinary\_grouping\_set* enclosed in parenthesis
- *rollup\_list*
- *cube\_list*
- *empty\_grouping\_set*
- *grouping\_sets\_specification*

There is no limit to the number of nesting levels for grouping sets specifications other than your system configuration.

You cannot reference columns that have a LOB data type.

**ANSI Compliance**

GROUPING SETS is ANSI SQL:2011 compliant.

**SAMPLE Clause and GROUPING SETS**

You cannot specify a SAMPLE clause in a query that also specifies the GROUPING SETS option in a GROUP BY clause. The exception is a query where the GROUPING SETS option is specified in a derived table or a view, and the SAMPLE clause is specified in the outer query.

**How GROUPING SETS Summarizes Data**

GROUPING SETS allows you to group your results set multiple times. Empty parentheses () specify a single grand total group that sums all the groups specified in the query.

**Example: GROUPING SETS**

The following example query forms the first group by state, county then forms the second group by city then forms the third group consisting of the whole table.

Suppose you have the following *sales\_view* table data:

sales_view						
PID	cost	sale	margin	state	county	city
1	38350	50150	11800	CA	Los Angeles	Long Beach
1	63375	82875	19500	CA	San Diego	San Diego
1	46800	61200	14400	CA	Los Angeles	Avalon

sales_view						
1	40625	53125	12500	CA	Los Angeles	Long Beach

You are interested in looking at sales summaries by county within state, by city, and by state.

The following SELECT statement analyzes the data in *sales\_view* for this information:

```
SELECT state, county, city, SUM(margin)
FROM sales_view
GROUP BY GROUPING SETS ((state,county),(city),());
```

The query reports the following results, where the QUESTION MARK character indicates a null:

state	county	city	SUM(margin)
-----	-----	----	-----
CA	Los Angeles	?	38700
CA	San Diego	?	19500
?	?	Long Beach	24300
?	?	San Diego	19500
?	?	Avalon	14400
?	?	?	58200

Notice that nulls are used to represent the empty set in this answer set, not missing information. The nulls mean that information is not reported at the grouping level represented, not that the information is missing from the *sales\_view* base table. This usage is similar to the way nulls are used in outer joins to indicate empty sets of nonmatching rows.

## ROLLUP Grouping Set Option

### Purpose

Analyzes a set of data across a single dimension with more than one level of detail.

### Syntax

```
— ROLLUP —
      |
      | ordinary_grouping_set
      |
      | ( — ordinary_grouping_set — )
      |
```



## Syntax Elements

### *ordinary\_grouping\_set*

One or more expressions used to group rolled up report rows across a single dimension.

The expression cannot group result rows that have a BLOB, CLOB, ARRAY, or VARRAY type.

All ordinary grouping sets must make explicit column references. You cannot specify column positions with ROLLUP.

Rolling up over a single column is equivalent to a simple GROUP BY on that same column.

You cannot specify more than 190 columns in the ordinary grouping sets.

You cannot reference columns that have a LOB data type unless they are first CAST to another data type or passed to a function whose result is not a LOB. For example, casting a BLOB to BYTE or VARBYTE or casting a CLOB to CHARACTER or VARCHAR.

## ANSI Compliance

ROLLUP is ANSI SQL:2011 compliant.

## SAMPLE Clause and ROLLUP

You cannot specify a SAMPLE clause in a query that also specifies the ROLLUP option in a GROUP BY clause, with the exception of a query where the ROLLUP option in the GROUP BY clause is specified in a derived table or a view, and the SAMPLE clause is specified in the outer query.

## How ROLLUP Summarizes Data

Given  $n$  column references, ROLLUP groups data at  $n$  levels of detail in one dimension. For example, in [Example: Basic ROLLUP](#), the GROUP BY clause rolls up over three columns and reports three levels of control breaks.

## Examples

### Example: Basic ROLLUP

The following example returns the total margin in each county, city, state, and as a whole. This example shows how ROLLUP groups data at three levels of detail in the single dimension of geography.

Assume the following *sales\_view* table data, where the QUESTION MARK character indicates a null:

sales_view						
PID	cost	sale	margin	state	county	city
1	38350	50150	11800	CA	Los Angeles	Long Beach
1	63375	82875	19500	CA	San Diego	San Diego
1	46800	61200	14400	CA	Los Angeles	Avalon

sales_view						
1	40625	53125	12500	CA	Los Angeles	Long Beach
1	30000	33000	3000	CA	San Diego	?

This SELECT statement against this table uses ROLLUP in its GROUP BY clause to roll up over *state*, *county*, and *city*:

```
SELECT state, county, city, SUM(margin)
FROM sales_view
GROUP BY ROLLUP (state, county, city);
```

The query reports the following data:

state	county	city	SUM(margin)
-----	-----	----	-----
CA	San Diego	San Diego	19500
CA	San Diego	?	3000
CA	Los Angeles	Avalon	14400
CA	Los Angeles	Long Beach	24300
CA	Los Angeles	?	38700
CA	San Diego	?	22500
CA	?	?	61200
?	?	?	61200

This query computes four different levels of grouping, or control breaks by:

1. *state*, *county*, and then *city*.
2. *state* and then *county*.
3. *state*.
4. All the data in the table.

Each control break is characterized by a null in the column being rolled up. These nulls represent an empty set, not missing information.

The following table summarizes this for the current example:

This control break number ...	Breaks on this column set ...	Groups on this column set in the order indicated ...	And has nulls in this column set to represent the rolled up columns...
1	City	State County City	None. Note that the null in the <i>city</i> column in the second row indicates missing information for that column in that row of the table, not an empty set representing a control break point in the report. See <a href="#">Example: ROLLUP With GROUPING Function</a> for a way to eliminate this ambiguity using CASE expressions.
2	City and County	State County	City
3	City, County, and State	State	City and County
4	All	All	City, County, and State

### Example: ROLLUP With GROUPING Function

Nulls are used to represent both missing information and the empty set in the report generated in [Example: Basic ROLLUP](#). For example, the fifth row of *sales\_view* has a null city in San Diego County, and this is reflected in the report by the second line, where the null maps 1:1 with the null representing the city for the fifth row of the table.

The nulls for *city* in lines 5 through 8 of the report indicate an empty set, not missing information. The nulls mean that information is not reported at the level they represent, not that the information is missing from the *sales\_view* base table. This appears to be confounded in row 6 of the report, because there *is* missing information about one of the reported cities in San Diego county; however, because the information for all cities in San Diego county is summed in this row, there is no problem because it is known that the null city is in that county.

- The first level of summation, represented symbolically as (*state, county*), breaks on counties, so in line 5 of the report, all cities are collapsed for Los Angeles County and in line 6 all cities are collapsed for San Diego County.
- The second level of summation, represented symbolically as (*state*), breaks on states, so in line 7, all cities and counties are collapsed for the state of California.
- The final level of summation reports the sum across all states, and line 8 reflects this level of summation. Because data for only one state, California, is recorded in the *sales\_view* table, the totals for lines 7 and 8 are identical. This would not be the case had there been additional state data recorded in the *sales\_view* table.

You can use CASE expressions with the GROUPING function (see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145) to better distinguish between nulls that represent missing information and nulls that represent the empty set, by reporting instances of the empty set using a character

string instead of the QUESTION MARK character. In this example, the character string representing the empty set is the phrase (-all-):

```
SELECT CASE GROUPING(state)
      WHEN 1
      THEN '(-all-)'
      ELSE state
      END AS state,
      CASE GROUPING(county)
      WHEN 1
      THEN '(-all-)'
      ELSE county
      END AS county,
      CASE GROUPING(city)
      WHEN 1
      THEN '(-all-)'
      ELSE city
      END AS city,
      SUM(margin)
FROM sales_view
GROUP BY ROLLUP (state, county, city);
```

This query reports the identical information as the previous query, but the representation is much cleaner:

state	county	city	SUM(margin)
-----	-----	----	-----
CA	San Diego	San Diego	19500
CA	San Diego	?	3000
CA	Los Angeles	Avalon	14400
CA	Los Angeles	Long Beach	24300
CA	Los Angeles	(-all-)	38700
CA	San Diego	(-all-)	22500
CA	(-all-)	(-all-)	61200
(-all-)	(-all-)	(-all-)	61200

# HAVING Clause

## Purpose

A conditional expression that must be satisfied for a group of rows to be included in the result data.

## Syntax

—— HAVING —— *search\_condition* ——

## Syntax Elements

### HAVING

An introduction to the conditional clause in the SELECT statement.

### *search\_condition*

One or more conditional expressions that must be satisfied by the result rows. You can specify aggregate operators, scalar subqueries, and DEFAULT functions as conditional expressions with HAVING.

HAVING *search\_condition* selects rows from a single group defined in the SELECT expression list that has only aggregate results, or it selects rows from the group or groups defined in a GROUP BY clause.

The HAVING search condition cannot reference BLOB, CLOB, ARRAY, or VARRAY columns.

If you specify the value for a row-level security constraint in a search condition, that value must be expressed in its encoded form.

## ANSI Compliance

The HAVING clause is ANSI SQL:2011-compliant.

## Usage Notes

### LOB Columns and the HAVING Clause

You cannot specify LOB columns in the HAVING search condition.

## Aggregates and the HAVING Clause

The conditional expression can define one or more aggregates (for example, MAX, MIN, AVG, SUM, COUNT) and can be applied to the rows of the following group conditions:

- A single group defined in the SELECT expression list, which has only aggregate results
- One or more groups defined in a GROUP BY clause

## Recursive Queries and the HAVING Clause

HAVING cannot be specified in a recursive statement of a recursive query. However, a nonrecursive seed statement in a recursive query can specify the HAVING clause.

## Row-level Security Constraints and the HAVING Clause

If you specify the value for a row-level security constraint in a search condition, that value must be expressed in its encoded form.

## Scalar Subqueries and the HAVING Clause

You can specify a scalar subquery as an operand of a scalar predicate in the HAVING clause of a query.

## DEFAULT Function and the HAVING Clause

The following rules apply to the use of the DEFAULT function as part of the search condition within a HAVING clause:

- You can specify a DEFAULT function with a column name argument within a predicate. The system evaluates the DEFAULT function to the default value of the column specified as its argument. Once the system has evaluated the DEFAULT function, it treats it like a constant in the predicate.
- You can specify a DEFAULT function without a column name argument within a predicate only if there is one column specification and one DEFAULT function as the terms on each side of the comparison operator within the expression.
- Following existing comparison rules, a condition with a DEFAULT function used with comparison operators other than IS NULL or IS NOT NULL is unknown if the DEFAULT function evaluates to null.

A condition other than IS NULL or IS NOT NULL with a DEFAULT function compared with a null evaluates to unknown.

DEFAULT Function Used with this Condition	Comparison Result
IS NULL	<ul style="list-style-type: none"> <li>• TRUE if the default is null</li> </ul>

DEFAULT Function Used with this Condition	Comparison Result
	<ul style="list-style-type: none"> <li>• Else it is FALSE</li> </ul>
IS NOT NULL	<ul style="list-style-type: none"> <li>• FALSE if the default is null</li> <li>• Else it is TRUE</li> </ul>

For more information about the DEFAULT function, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

## SAMPLE Clause and the HAVING Clause

You cannot specify a SAMPLE clause in a subquery used as a HAVING clause predicate.

## Evaluation Order of WHERE, GROUP BY, and HAVING Clauses

When the WHERE, GROUP BY, and HAVING clauses are used together in a SELECT statement, the order of evaluation is as follows:

1. WHERE
2. GROUP BY
3. HAVING

## Tables References in a HAVING Clause

Tables referenced in a HAVING clause must be specified in one of the following:

- FROM clause
- WHERE clause
- SELECT expression list
- Non-aggregate condition

## Aggregating a Join In a HAVING Clause

You can use the HAVING clause when referencing columns from two or more tables.

## Examples

### Examples: HAVING Clause

The following example shows the use of HAVING as applied to the aggregate results of a single group defined in the SELECT list, which is particularly useful in a SELECT subquery.

```
SELECT COUNT(employee)
FROM department
WHERE dept_no = 100
HAVING COUNT(employee) > 10;
```

The following SELECT statements are additional examples of the correct use of the HAVING clause.

```
SELECT SUM(t.a)
FROM t,u
HAVING SUM(t.a)=SUM(u.a);
SELECT SUM(t.a), SUM(u.a)
FROM t,u
HAVING SUM(t.b)=SUM(u.b);
SELECT SUM(t.a)
FROM t,u
HAVING SUM(t.b)=SUM(u.b)
AND u.b = 1
GROUP BY u.b;
```

### Example: Grouping Departments by Average Salaries

Display salary ranges for specified departments whose salaries average more than \$37,000:

```
SELECT dept_no, MIN(salary), MAX(salary), AVG(salary)
FROM employee
WHERE dept_no IN (100,300,500,600)
GROUP BY dept_no
HAVING AVG(salary) > 37000;
```

The result is:

dept_no	MIN(salary)	MAX(salary)	AVG(salary)
300	23,000.00	65,000.00	47,666.67
500	22,000.00	56,000.00	38,285.71



## Example: Using a HAVING Clause to Aggregate a Join

The columns named price and sales\_qty are from two different tables, sales\_hist as table\_1 and unit\_price\_cost as table\_2. Use the following SELECT statement to find which category of items is sold for a profit margin greater than \$1000.

```
SELECT table_1.category,
       (table_2.price - table_2.cost) * SUM (table_1.sales_qty) AS margin
FROM sales_hist AS table_1, unit_price_cost AS table_2
WHERE table_1.prod_no=table_2.prodno
GROUP BY table_1.category, table_2.price, table_2.cost
HAVING margin > 1000;
```

A subquery can have a join on a view with an aggregate operation and a HAVING clause that references more than one table.

## Related Topics

For more information related to the HAVING clause, see:

- [WHERE Clause](#)
- [QUALIFY Clause](#)

## QUALIFY Clause

### Purpose

A conditional clause in the SELECT statement that filters results of a previously computed ordered analytical function according to user-specified search conditions.

### Syntax

```
— QUALIFY — search_condition —————
```

### Syntax Elements

#### QUALIFY

An introduction to a conditional clause that, similar to HAVING, further filters rows from a WHERE clause. The major difference between QUALIFY and HAVING is that with QUALIFY the filtering is based on the result of performing various ordered analytical functions on the data.

#### *search\_condition*

One or more conditional expressions that must be satisfied by the result rows.

You can specify ordered analytical functions and scalar subqueries as search conditions with QUALIFY.

You can use aggregate operators and DEFAULT functions within a QUALIFY search condition.

The QUALIFY condition cannot reference LOB columns.

If you specify the value for a row-level security constraint in a search condition, that value must be expressed in its encoded form.

## ANSI Compliance

The QUALIFY clause is a Teradata extension to the ANSI standard.

## Related Topics

For more information related to the QUALIFY clause, see:

- [WHERE Clause](#)
- [HAVING Clause](#)

## Usage Notes

### Rules and Restrictions for the QUALIFY Clause

The rules and restrictions are:

- You cannot specify LOB columns in a QUALIFY clause search condition.
- You cannot specify an OR condition connected with a subquery in a QUALIFY clause search condition.

The following query fails because of the OR condition connected with the subquery.

```
SELECT RANK (column_1)
FROM table_1
QUALIFY column_1 IN (SELECT table_2.column_1
                     FROM table_2)
OR table_1.column.1 = 10000;
```

- When you specify a QUALIFY clause in a query, you must also specify a statistical function in one of the following locations within the query.
  - The select list of the SELECT clause
  - The grouping key of the GROUP BY clause
  - The search condition of the QUALIFY clause
- If you specify the value for a row-level security constraint in a search condition, that value must be expressed in its encoded form.

- The following rules apply to the use of the DEFAULT function as part of the search condition within a QUALIFY clause.
  - You can specify a DEFAULT function with a column name argument within a predicate. Teradata Database evaluates the DEFAULT function to the default value of the column specified as its argument and uses the value like a constant in the predicate.
  - You can specify a DEFAULT function without a column name argument within a predicate only if there is one column specification and one DEFAULT function as the terms on each side of the comparison operator within the expression.
  - Following existing comparison rules, a condition with a DEFAULT function used with comparison operators other than IS NULL or IS NOT NULL is unknown if the DEFAULT function evaluates to null.

A condition other than IS NULL or IS NOT NULL with a DEFAULT function compared with a null evaluates to unknown.

IF a DEFAULT function is used with ...	THEN the comparison is ...
IS NULL	<ul style="list-style-type: none"> <li>▪ TRUE if the default is null</li> <li>▪ Else it is FALSE</li> </ul>
IS NOT NULL	<ul style="list-style-type: none"> <li>▪ FALSE if the default is null</li> <li>▪ Else it is TRUE</li> </ul>

See *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145 for more information about the DEFAULT function.

- A SELECT statement that specifies the TOP *n* operator cannot also specify the QUALIFY clause.
- You cannot specify a SAMPLE clause in a subquery used as a QUALIFY clause predicate.

## Evaluation Order of WHERE, GROUP BY, and QUALIFY Clauses

When the WHERE, GROUP BY, and QUALIFY clauses are used together in a SELECT statement, the order of evaluation is as follows:

1. WHERE clause
2. GROUP BY clause
3. QUALIFY clause

The detailed steps are as follows:

1. Teradata Database evaluates the WHERE clause conditions on the FROM clause tables.
2. The system groups the resulting rows using the GROUP BY columns.
3. Teradata Database evaluates the ordered analytical functions on the grouped table.
4. The system applies the QUALIFY clause to the resulting set.

Teradata Database-specific functions such as CSUM and MAVG that are invoked in both the select list and in the search condition of the QUALIFY clause are computed on the grouped rows without eliminating any rows and then the system applies the search condition of the QUALIFY clause.

For window functions, such as SUM and AVG, the GROUP BY collapses all rows with the same value for the group-by columns into a single row.

## QUALIFY Error Conditions

Tables referenced in a QUALIFY clause must be specified in one of the following:

- FROM clause
- WHERE clause
- SELECT expression list
- Non-aggregate condition

Otherwise, the system returns an error to the requestor. QUALIFY is similar to HAVING in that rows are eliminated based on the value of a function computation. With QUALIFY, rows are eliminated based on the computation of the ordered analytical functions.

## Examples

### Example: Using the RANK Function in a QUALIFY Clause

The following statement displays each item in a *sales* table, its total sales, and its rank within the top 100 selling items:

```
SELECT itemid, sumprice, RANK() OVER (ORDER BY sumprice DESC)
FROM (SELECT a1.item_id, SUM(a1.sale)
      FROM sales AS a1
      GROUP BY a1.itemID) AS t1 (item_id, sumprice)
QUALIFY RANK() OVER (ORDER BY sum_price DESC) <=100;
```

### Example: Reporting the Bottom Percentile of Items Using QUANTILE in a QUALIFY Clause

The following example reports the bottom percentile of items by profitability:

```
SELECT item, profit, QUANTILE(100, profit) AS percentile
FROM (SELECT item, SUM(sales)-(COUNT(sales)*items.itemcost)
      AS profit
      FROM daily_sales, items
      WHERE daily_sales.item = items.item
```

```
GROUP BY item) AS itemprofit
QUALIFY percentile = 99;
```

The results of this query might look something like the following table.

item	profit	percentile
Carrot-flavored ice cream	10.79	99
Low fat carrot-flavored ice cream	- 100.55	99
Low fat sugar-free carrot-flavored ice cream	- 1,110.67	99
Regular broccoli-flavored ice cream	- 2,913.88	99
Low fat broccoli-flavored ice cream	- 4,492.12	99

## Example: Behavior of OLAP Aggregate Functions That Return Zeros

When you specify an ordered analytical aggregate function in the search condition of a QUALIFY clause, it can return a result of 0.

When there are no values to aggregate, ordered analytical aggregate functions return a 0 instead of a null.

This example first shows the rows in the *demogr* table that are used to calculate the ordered analytical aggregate result for the examples of current behavior that follow.

The statement returns 12 rows with valueless aggregate rows reported with zeros rather than nulls in the Remaining Count(inc) column.

```
SELECT line, da, mon, inc, COUNT(inc) OVER(PARTITION BY mon
                                           ORDER BY mon, line
                                           ROWS BETWEEN 1 FOLLOWING
                                           AND UNBOUNDED FOLLOWING)
FROM demogr
WHERE yr = 94
AND mon < 13
AND da < 10;
```

\*\*\* Query completed. 12 rows found. 5 columns returned.  
 \*\*\* Total elapsed time was 1 second.

line	da	mon	inc	Remaining Count(inc)
4803	3	1	234737.37	0
3253	2	1	?	1
625	4	2	46706.97	0
3853	3	4	282747.07	0
3687	1	5	172470.52	0
547	9	5	31848.56	1
2292	6	7	170411.66	0
5000	8	9	40548.61	0
3213	8	9	257858.55	1
3948	6	10	217091.30	0
2471	1	10	121299.65	1
1496	7	12	?	0

This example shows the current behavior of a qualify clause over *demogr* using the ordered analytical COUNT function. The query again returns 12 rows, with zeros in place of nulls in the Remaining Count(inc) column.

```

SELECT line, da, mon, inc, COUNT(inc) OVER(PARTITION BY mon
                                           ORDER BY mon, line
                                           ROWS BETWEEN 1 FOLLOWING
                                           AND UNBOUNDED FOLLOWING)
FROM demogr
WHERE yr = 94
AND mon < 13
AND da < 10
QUALIFY COUNT(inc) OVER(PARTITION BY mon ORDER BY mon, line
                        ORDER BY mon, line
                        ROWS BETWEEN 1 FOLLOWING
                        AND UNBOUNDED FOLLOWING) < 3 ;
*** Query completed. 12 rows found. 5 columns returned.
*** Total elapsed time was 1 second.

```

line	da	mon	inc	Remaining Count(inc)
4803	3	1	234737.37	0
3253	2	1	?	1
625	4	2	46706.97	0
3853	3	4	282747.07	0
3687	1	5	172470.52	0
547	9	5	31848.56	1
2292	6	7	170411.66	0
5000	8	9	40548.61	0
3213	8	9	257858.55	1
3948	6	10	217091.30	0
2471	1	10	121299.65	1
1496	7	12	?	0

This query returns the rows from the *demogr* table used to calculate the ordered analytical aggregate result for the examples that show the old style of returning results of an ordered analytical aggregate function specified in the search condition of a QUALIFY clause.

By using this method, valueless aggregates are returned as nulls in the Remaining Sum(1) column rather than as zeros.

```

SELECT line, da, mon, inc, SUM(1) OVER(PARTITION BY mon
                                       ORDER BY mon, line
                                       ROWS BETWEEN 1 FOLLOWING
                                       AND UNBOUNDED FOLLOWING)
FROM demogr
WHERE yr = 94
AND mon < 13
AND da < 10;
*** Query completed. 12 rows found. 5 columns returned.
*** Total elapsed time was 1 second.

```

line	da	mon	inc	Remaining Sum(1)
4803	3	1	234737.37	?
3253	2	1	?	1
625	4	2	46706.97	?
3853	3	4	282747.07	?
3687	1	5	172470.52	?
547	9	5	31848.56	1
2292	6	7	170411.66	?
5000	8	9	40548.61	?
3213	8	9	257858.55	1
3948	6	10	217091.30	?

2471	1	10	121299.65	1
1496	7	12	?	?

This example shows how to use a ordered analytical SUM(1) function as a workaround to return nulls in the result instead of zeros. The query again returns 12 rows, but reports valueless aggregate rows as nulls rather than zeros for the Remaining Sum(1) column.

```

SELECT line, da, mon, inc, SUM(1) OVER(PARTITION BY mon
                                      ORDER BY mon, line
                                      ROWS BETWEEN 1 FOLLOWING
                                      AND UNBOUNDED FOLLOWING)
FROM demogr
WHERE yr = 94
AND mon < 13
AND da < 10
QUALIFY SUM(1) OVER(PARTITION BY mon ORDER BY mon, line
                  ROWS BETWEEN 1 FOLLOWING
                  AND UNBOUNDED FOLLOWING) < 3;
*** Query completed. 12 rows found. 5 columns returned.
*** Total elapsed time was 1 second.

```

line	da	mon	inc	Remaining Sum(1)
4803	3	1	234737.37	?
3253	2	1	?	1
625	4	2	46706.97	?
3853	3	4	282747.07	?
3687	1	5	172470.52	?
547	9	5	31848.56	1
2292	6	7	170411.66	?
5000	8	9	40548.61	?
3213	8	9	257858.55	1
3948	6	10	217091.30	?
2471	1	10	121299.65	1
1496	7	12	?	?

This example shows how to use NULLIFZERO with the ordered analytical COUNT function as a workaround to return nulls in the result instead of zeros.

```

SELECT line, da, mon, inc, NULLIFZERO(COUNT(inc)
                                      OVER(PARTITION BY mon
                                      ORDER BY mon, line
                                      ROWS BETWEEN 1 FOLLOWING
                                      AND UNBOUNDED FOLLOWING))
FROM demogr
WHERE yr = 94
AND mon < 13
AND da < 10
QUALIFY NULLIFZERO(COUNT(inc)OVER(PARTITION BY mon
                                ORDER BY mon, line
                                ROWS BETWEEN 1 FOLLOWING
                                AND UNBOUNDED FOLLOWING)) < 3;
*** Query completed. 12 rows found. 5 columns returned.
*** Total elapsed time was 1 second.

```

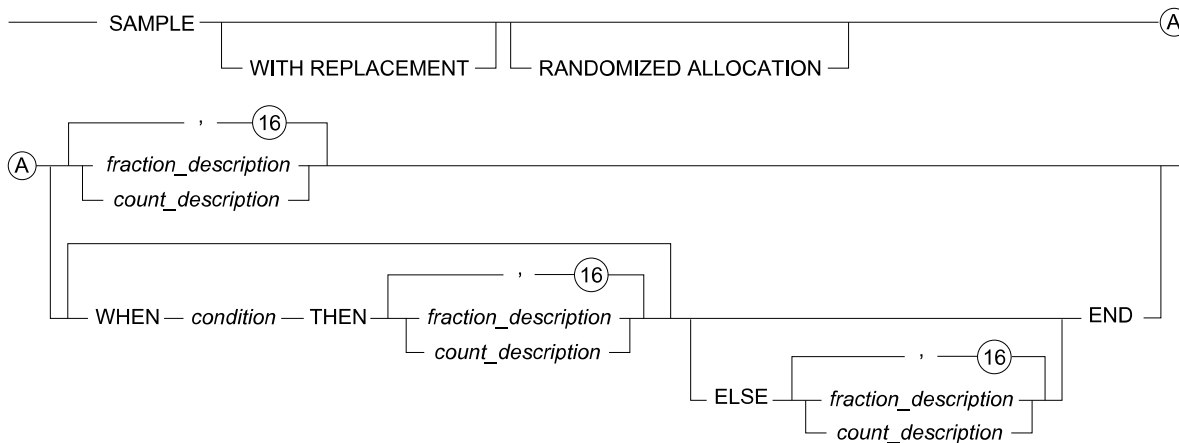
line	da	mon	inc	Remaining Count(inc)
4803	3	1	234737.37	?
3253	2	1	?	1
625	4	2	46706.97	?
3853	3	4	282747.07	?
3687	1	5	172470.52	?
547	9	5	31848.56	1
2292	6	7	170411.66	?
5000	8	9	40548.61	?
3213	8	9	257858.55	1
3948	6	10	217091.30	?
2471	1	10	121299.65	1
1496	7	12	?	?

## SAMPLE Clause

## Purpose

Reduces the number of rows to be considered for further processing by returning one or more samples of rows specified either as a list of fractions of the total number of rows or as a list of numbers of rows from the SELECT query.

## Syntax



## Syntax Elements

## SAMPLE

Introduction to a clause that permits sampling of rows in the SELECT statement.

### WITH REPLACEMENT

Whether sampling is done by returning each sampled row to the table for possible redundant sampling or by withholding sampled rows from resampling.

If you specify WITH REPLACEMENT, then it is possible to request more samples than there are rows in the table.

Sampling without replacement is the default. You select it implicitly by not specifying WITH REPLACEMENT.

## RANDOMIZED ALLOCATION

Rows are sampled randomly across AMPS. Otherwise, rows are sampled proportionate to the number of qualified rows per AMP (proportional allocation).

The proportional allocation option does not provide a simple random sample of the entire population. It provides a random sample stratified by AMPs, but it is much faster, especially for very large samples.

Proportional is the default. You select it implicitly by not specifying RANDOMIZED ALLOCATION.



***fraction\_description***

Any set of unsigned floating point constant numbers in the closed interval (0,1) that specifies the percentage of rows to be sampled for a true search condition.

This is a comma-separated list of fractions, the sum of which must not exceed 1.

The value set specifies the percentage of the homogeneous subgroup defined by *search\_condition* to be sampled for the report.

Up to 16 samples can be requested per fraction description.

***count\_description***

Set of positive integer constants that specifies the number of rows to be sampled for a true search condition.

A warning is returned if there are not enough rows in the result to satisfy the sampling request completely.

Up to 16 samples can be requested per count description.

**WHEN**

Test a set of conditions for truth.

***search\_condition***

Evaluation predicate that defines each homogeneous subgroup in the sample set.

You can only specify a scalar UDF for *search\_condition* if it is invoked within an expression and returns a value expression.

You cannot specify expressions that contain LOBs in a search condition unless you first cast them to another type or pass them to a function whose result is not a LOB, for example, casting a BLOB to BYTE or VARBYTE or casting a CLOB to CHARACTER or VARCHAR.

If you specify the value for a row-level security constraint in a search condition, it must be expressed in its encoded form.

**THEN**

Apply the specified sampling fraction description or count description to the sample.

**ELSE**

Apply the specified sampling fraction description or count description to the sample if none of the WHEN *condition* predicates evaluates to true.

**END**

Termination of the WHEN ... THEN ... ELSE clause.

**ANSI Compliance**

The SAMPLE clause is a Teradata extension to the ANSI SQL:2011 standard.

## Usage Notes, SAMPLE Clause

### Rules and Restrictions for Using the SAMPLE Clause

The rules and restrictions are:

- If a *fraction\_description* results in no rows being returned, a warning is generated.
- If a *count\_description* cannot be completely satisfied, a warning is generated and the sample size is reduced to the number of remaining rows.
- No more than 16 samples can be requested per fraction description or count description.
- A sampling request cannot be repeated. The identical sampling query run twice against the same data will report different rows in the result.
- Sampling can be used in a derived table, view, or INSERT ... SELECT to reduce the number of rows to be considered for further computation.
- If an INSERT ... SELECT statement specifies a SAMPLE clause that selects a set of rows from a source MULTiset table, but inserts them into a target SET table, and the sampled row set contains duplicates, the number of rows inserted into the target SET table might be fewer than the number requested in the SAMPLE clause.

The condition occurs because SET tables reject attempts to insert duplicate rows into them. The result is that the INSERT portion of the INSERT ... SELECT statement inserts only distinct rows into SET target tables. As a result, the number of rows inserted into the target table can be fewer than the number specified in the SAMPLE clause.

For example, if an INSERT ... SELECT statement SAMPLE clause requests a sample size of 10 rows, and there are duplicate rows from the MULTiset source table in the collected sample, the system rejects the duplicate instances when it attempts to insert the sampled rows into the SET table and inserts only the distinct rows from the sample set. In other words, you could request a sample of 10 rows, but the actual number of rows inserted into the target table could be fewer than 10 if there are duplicate rows in the sampled row set.

The system does not return any warning or information message when this condition occurs.

- You cannot specify a SAMPLE clause in a WHERE, HAVING, QUALIFY, or ON clause of a subquery.
- You cannot specify the SAMPLE clause in a SELECT statement that uses the set operators UNION, INTERSECT, or MINUS.
- You can only specify a scalar UDF for *search\_condition* if it is invoked within an expression and returns a value expression.
- You cannot specify a SAMPLE clause in a query that specifies the GROUP BY clause and any of the following extended grouping options.
  - CUBE
  - ROLLUP
  - GROUPING SETS

An exception is a query where the GROUP BY clause and extended grouping option appear in a derived table or view and the SAMPLE clause appears in the outer query.

- A SELECT statement that includes the TOP *n* operator cannot also specify the SAMPLE clause.

## About SAMPLE

- SAMPLE operates on the evaluated output of the table expression, which can include a WHERE clause and GROUP BY, HAVING, or QUALIFY clauses, sampling the result according to user specification.
- You can specify sampling either with or without replacement.
- You can specify sample allocation as either randomized or proportional.
- You can use the SAMPLEID expression to identify the samples to which the rows belong. See [SAMPLEID Expression](#).

## Simple Random Sampling

Simple random sampling is a procedure in which every possible set of the requested size has an equal probability of being selected.

## Stratified Random Sampling

Stratified random sampling, sometimes called proportional or quota random sampling, is a sampling method that divides a heterogeneous population of interest into homogeneous subgroups, or strata, and then takes a random sample from each of those subgroups.

The result of this homogeneous stratification of the population is that stratified random sampling represents not only the overall population, but also key subgroups. For example, a retail application might divide a customer population into subgroups composed of customers who pay for their purchases with cash, those who pay by check, and those who buy on credit.

## Sampling With or Without Replacement

The WITH REPLACEMENT option specifies that sampling is to be done with replacement. The default is sampling *without* replacement. Sampling without replacement is assumed implicitly if you do not specify WITH REPLACEMENT explicitly.

When sampling with replacement, a sampled row, once sampled, is returned to the sampling pool. As a result, a row might be sampled multiple times. Because of this, it is possible to sample more rows than the number of rows in the input. When multiple samples are requested, all the samples are from the whole population and therefore not mutually exclusive.

Sampling without replacement is analogous to selecting rows from a SET table in that each row sampled is unique, and once a row is sampled, is not returned to the sampling pool. As a result, requesting a number of samples greater than the cardinality of the table returns an error or warning. Whenever multiple samples are requested, they are mutually exclusive.

The magnitude of the difference of the results obtained by these two methods varies with the size of the sample relative to the population. The smaller the sample relative to the population, the less the difference in the results of sampling with or without replacement.

## Randomized and Proportional Row Allocation

Randomized allocation means that the requested rows are allocated across the AMPs by simulating random sampling. This is a slow process, especially for large sample sizes, but it provides a simple random sample for the system as a whole.

The default row allocation method is proportional. This means that the requested rows are allocated across the AMPs as a function of the number of rows on each AMP. This method is much faster than randomized allocation, especially for large sample sizes. Because proportional allocation does not include all possible sample sets, the resulting sample is not a simple random sample, but it has sufficient randomness to suit the needs of most applications.

Note that simple random sampling, meaning that each element in the population has an equal and independent probability of being sampled, is employed for each AMP in the system regardless of the specified allocation method.

One way to decide on the appropriate allocation method for your application is to determine whether it is acceptable to stratify the sampling input across the AMPs to achieve the corresponding performance gain, or whether you need to consider the table as a whole.

The SAMPLEID value is simply 1, 2, 3, ...  $n$  across  $n$  specified samples regardless of stratification. That is, for the following SAMPLE clause,

	SAMPLE WHEN state = 'CA' THEN	0.3,	0.2	ELSE	0.5,	0.2
the SAMPLEID correspondence would be:		1	2		3	4

## Examples: Sample Clause

### Example: Using *fraction\_description*

Suppose you want to generate three mutually exclusive sample sets of a customer table for a neural net analysis. The desired percentages are as follows:

- Train, 60%
- Test, 25%
- Validate, 15%

Note that the sum does not exceed 100%.

A SELECT statement to generate the desired result looks like the following.

```
SELECT customer_id, age, income, marital_status, SAMPLEID
FROM customer_table
SAMPLE 0.6, 0.25, 0.15;
```

The result might look something like the following table.

customer_id -----	age ---	income -----	marital_status -----	SAMPLEID -----
1362549	17	0	1	1
1362650	21	17,804	2	1
1362605	34	16,957	2	1
1362672	50	16,319	2	3
1362486	76	10,701	3	1
1362500	40	56,708	1	3
1362489	35	55,888	3	2
1362498	60	9,849	2	1
1362551	27	23,085	1	1
1362503	18	5,787	1	2

Sample 1 is the training group, Sample 2 is the test group, and Sample 3 is the validation group.

### Example: Using *count\_description*

Check if your customers are in at least 100 cities:

```
SELECT COUNT (DISTINCT city)
FROM (SELECT city
      FROM customer_table
      SAMPLE 1000) TEMP;
```

If *customer\_table* is large, the SAMPLE 1000 clause would not require a full scan of the table and the sort for DISTINCT would only handle 1,000 rows.

A 1,000 row sample is more than 95 percent accurate for estimating if the number of distinct values is greater than 100.

If you were to make a similar query without including the SAMPLE clause, the query would first have to sort the large *customer\_table* before performing the DISTINCT. For example:

```
SELECT COUNT (DISTINCT city)
FROM customer_table;
```

### Examples Using Stratified Sampling

This table provides the data used in the examples that follow.

```

SELECT *
FROM stores;

```

storeid	city	state
2	Green Bay	WI
7	San Diego	CA
5	San Jose	CA
8	Los Angeles	CA
3	Madison	WI
1	Racine	WI
6	San Francisco	CA
4	Milwaukee	WI

### Example: Stratified Sampling and Proportional Allocation Without Replacement

The following query uses proportional allocation by default to sample, without replacement, 25 percent of the rows for WI and 50 percent of the rows for CA:

```

SELECT city, state, SAMPLEID
FROM stores
SAMPLE WHEN state = 'WI' THEN 0.25
      WHEN state = 'CA' THEN 0.5
      END
ORDER BY 3;

```

city	state	SAMPLEID
Milwaukee	WI	1
San Diego	CA	2
San Jose	CA	2

### Example: Stratified Sampling and Proportional Allocation With Replacement

The following query uses proportional allocation by default with replacement to sample two samples of 3 rows and 1 row, respectively, which are not mutually exclusive, for WI and two samples of 2 rows each for CA, which are not mutually exclusive:

```

SELECT city, state, SAMPLEID
FROM stores
SAMPLE WITH REPLACEMENT
      WHEN state = 'WI' THEN 3, 1
      WHEN state = 'CA' THEN 2, 2
      END
ORDER BY 3;

```

city	state	SAMPLEID
------	-------	----------

Green Bay	WI	1
Madison	WI	1
Madison	WI	1
Racine	WI	2
San Diego	CA	3
San Jose	CA	3
San Diego	CA	4
San Jose	CA	4

### Example: Stratified Sampling With Randomized Allocation Without Replacement

The following query uses randomized allocation without replacement to sample two mutually exclusive samples of 25 percent and 50 percent, respectively, of the rows from WI and two mutually exclusive samples of 25 percent each for CA:

```
SELECT city, state, SAMPLEID
FROM stores
SAMPLE RANDOMIZED ALLOCATION
    WHEN state = 'WI' THEN 0.25, 0.5
    WHEN state = 'CA' THEN 0.25, 0.25
END
ORDER BY 3;
city      state      SAMPLEID
-----
Green Bay  WI          1
Milwaukee WI          2
Madison   WI          2
San Diego  CA          3
San Francisco CA         4
```

### Example: Stratified Sampling With Randomized Allocation With Replacement

The following query uses randomized allocation with replacement to sample three rows for WI, and two nonspecific, non-WI rows:

```
SELECT city, state, SAMPLEID
FROM stores
SAMPLE WITH REPLACEMENT RANDOMIZED ALLOCATION
    WHEN state = 'WI' THEN 3
    ELSE 2
END
ORDER BY 3;
city      state      SAMPLEID
-----
Racine    WI          1
```

Racine	WI	1
Madison	WI	1
San Diego	CA	2
San Diego	CA	2

### Example: Stratified Sampling With Randomized Allocation With Replacement

The following query samples three rows with replacement using randomized allocation:

```
SELECT city, state
FROM stores
SAMPLE WITH REPLACEMENT RANDOMIZED ALLOCATION 3;
city          state
-----
San Diego     CA
Madison       WI
San Jose      CA
```

### Example: SAMPLE and the PERIOD Value Expression

The following example shows how you can specify a SAMPLE clause on a PERIOD value expression, where *period\_of\_stay* is the PERIOD value expression.

```
SELECT emp_no, period_of_stay, SAMPLEID
FROM employee
SAMPLE 0.5,0.5;
```

## SAMPLEID Expression

### Purpose

Identifies the sample to which a row belongs, distinguishing rows belonging to different samples specified in the SAMPLE clause of a SELECT statement.

### Syntax

— SAMPLEID —

### Syntax Elements

#### SAMPLEID

Select list or ORDER BY clause expression that indicates that sample identifiers are to be used to link result set rows with their originating samples.



## ANSI Compliance

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

### Definition of a Sample ID

The sample ID identifies the sample to which a row belongs in the left-to-right order of the SAMPLE clause specification, from 1 through  $n$  (where  $n$  is the number of samples requested in the SAMPLE clause).

### Rules and Restrictions for SAMPLEID

The rules and restrictions are:

- You can only specify SAMPLEID with a SAMPLE clause, which can appear either as part of a select list or as an ORDER BY clause expression.
- SAMPLEID cannot be the only term specified in the select list. If you specify SAMPLEID, you must also specify at least one other non-SAMPLEID column expression.

### Using SAMPLEID With Stratified Sampling

The SAMPLEID value for stratified sampling is simply 1, 2, 3, ...  $n$  across  $n$  specified samples regardless of stratification. That is, for the following SAMPLE clause,

<code>SAMPLE WHEN state = 'CA' THEN</code>	<code>0.3,</code>	<code>0.2</code>	<code>ELSE</code>	<code>0.5,</code>	<code>0.2</code>
the SAMPLEID correspondence would be:	1	2		3	4

### Example: Three Sample Sets Using SAMPLEID as a Select List Expression

The following SELECT statement provides three mutually exclusive sample sets (60 percent for sample category x, 25 percent for sample category y, and 15 percent for sample category z) from a customer table.

The results are returned in a table with three columns: *cust\_name*, *cust\_addr*, and *SAMPLEID*.

The integer in the SAMPLEID column identifies whether the row belongs to the 0.6 sample, the 0.25 sample, or the 0.15 sample.

The samples are identified as 1 through 3, in left-to-right order from the SAMPLE clause, so 0.6 is identified by 1, 0.25 by 2, and 0.15 by 3.

```
SELECT cust_name, cust_addr, SAMPLEID
FROM customer_table
SAMPLE 0.6, 0.25, 0.15;
```

A partial results table might look something like the following:

<i>cust_name</i>	<i>cust_addr</i>	<i>SAMPLEID</i>
Jones Pharmaceuticals	4235 Lawler Road Memphis, TN	1

cust_name	cust_addr	SAMPLEID
	USA	
Fong Furniture	310 East Highway 5 Hong Kong	2
Subramaniam Spice Exports	455 1/2 Gandhi Lane Hyderabad India	3
Forrester Property Management	1 West Broadway Syracuse, New York USA	1
Otomo Consulting	33 Korakuen Hall Tokyo Japan	1
Adler Music Publishing, Ltd.	5 East 245th Street Nashville, TN USA	
O'Brien Metals	83 Heatherington The Whithers Cobblestone-on-Treads United Kingdom	1
Irama Rice Importers	8562 Rhoma Lane Jakarta Indonesia	2
Abdelwahab Fine Egyptian Rugs	1723 Kulthum Avenue Cairo Egypt	1
Bachar Ouds	18 Rashied Diagonal Baghdad Iraq	1

### Example: Using SAMPLEID With PERIOD Value Expressions

The following example shows how you can specify a PERIOD value expression with a SAMPLEID expression, where *period\_of\_stay* is the PERIOD value expression.

```
SELECT emp_no, period_of_stay, SAMPLEID
FROM employee
SAMPLE 0.5,0.5;
```

# EXPAND ON Clause

## Purpose

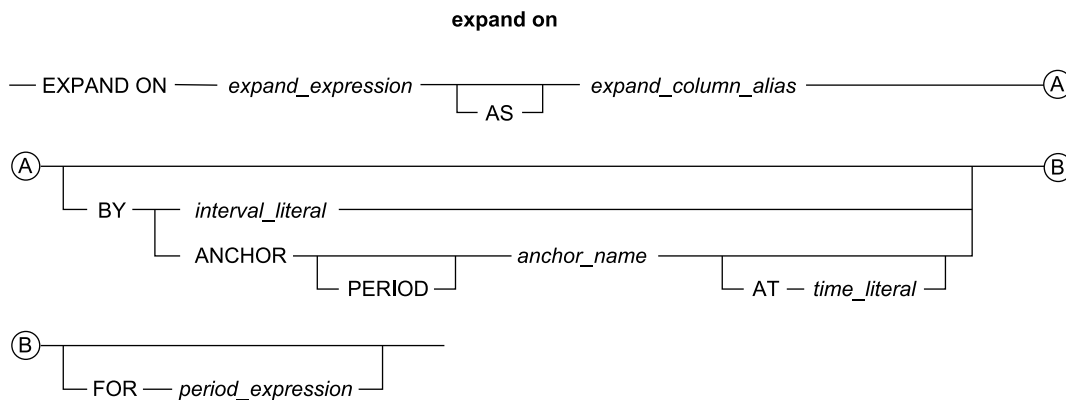
Expands a column having a PERIOD data type, creating a regular time series of rows based on the period value in the input row.

The expansion of a PERIOD column produces one value-equivalent row for each of the time granules in the epoch or timestamp representation of the specified time granule.

You can perform time series expansions only on PERIOD expressions which can be regular period expressions or derived period columns.

For more information about the EXPAND ON clause, see *Teradata Vantage™ Temporal Table Support*, B035-1182.

## Syntax



## Syntax Elements

### EXPAND ON *expand\_expression*

Time series expansion information in a query expression.

The EXPAND ON clause generates a regular time series as a sequence of values at each of the granules or at each point of a predefined interval in the specified period from an input row.

The *expand\_expression* variable specifies a PERIOD column name or PERIOD expression on which the selected rows are to be expanded. This is referred to as the input row for the EXPAND ON operation.

If the expansion period is null, then the expanded row has a null expanded value.

The specified column or expression must have a PERIOD data type or can be a derived period column.

### *expand\_column\_alias*

### AS *expand\_column\_alias*

Aliased name of the PERIOD expression to be expanded.

The aliased column or expression must have a PERIOD data type.

You can reference *expand\_column\_alias* in the select list, including inside an expression that does not reference any other columns and refers to the expanded value for an expanded row.

You cannot specify *expand\_column\_alias* in other clauses within the same query block except for an ORDER BY clause.

You cannot specify *expand\_column\_alias* in any clauses in a subquery or correlated subquery, nor can you specify it in any aggregate or statistical function in the select list.

### **BY *expand\_interval***

Interval literal (interval expression) or anchored literal by which *expand\_column\_name* is to be expanded, where *expand\_interval* is one of the valid *expand\_interval* options, including:

- *interval\_literal*
- ANCHOR *anchor\_name*
- ANCHOR PERIOD *anchor\_name*

This value specifies the granularity of the value-equivalent rows produced by the EXPAND ON clause. For a complete list of the expand interval literals, see [Expand Interval Literal Expressions](#).

If you do not specify the BY *expand\_interval* option, the expansion interval defaults to the granularity of the element type of the PERIOD value for the column.

The expansion commences from the beginning value of the expansion period and terminates at the ending value of the expansion period, incrementing by *interval\_literal* for each expanded row.

In this process, the last expanded row may not cover the expansion interval duration, producing only a partial period. By definition, a partial period is a period with a duration that is less than the expanded interval.

### **ANCHOR *anchor\_name***

#### **ANCHOR PERIOD *anchor\_name***

Expansion is an anchor PERIOD expansion.

Specify PERIOD only for anchor PERIOD expansions.

- If you specify PERIOD for noninterval data, then the expansion is an Anchor PERIOD expansion.
- If you do not specify PERIOD for noninterval data, then the expansion is an Anchor Point expansion.

The variable *anchor\_name* specifies an anchor name literal. For a complete list of the anchor name literals, see [Anchor Name Literal Expressions](#).

### **AT *time\_literal***

Optional Time literal. For a complete list of Time literals, see *SQL Data Types and Literals*.

If you do not specify a time literal value, the value defaults to '00:00:00.000000 00:00' for an ANCHOR\_MILLISECOND, ANCHOR\_SECOND, ANCHOR\_MINUTE, ANCHOR\_HOUR, WEEK\_BEGIN, MONTH\_BEGIN, QUARTER\_BEGIN, or YEAR\_BEGIN anchor and to '23:59:59.999999 00:00' for any other anchors.

**FOR *period\_expression***

Limit the number of rows to be expanded, where *period\_expression* represents the period of interest. For a comprehensive list of valid PERIOD expressions, see *Teradata Vantage™ Data Types and Literals*, B035-1143.

The expansion period is the overlapping period of the qualified row and the PERIOD constant you specify as *period\_expression*.

Otherwise, the expanding period is the PERIOD value of the selected row.

The data type of *period\_expression* must be comparable with the PERIOD data type of the expanded column.

If the specified PERIOD expression is either null or does not overlap with the row, then Teradata Database does not expand the row because it does not qualify for expansion.

**ANSI Compliance**

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

Because of this, Teradata Database reports a warning message if you use an EXPAND ON clause in a session in which the SQL Flagger is set. For information about the SQL Flagger, see *Teradata Vantage™ SQL Fundamentals*, B035-1141.

The Teradata SQL EXPAND ON clause does not share any functionality with the EXPANDING clause of the ANSI SQL:2011 standard.

**Usage Notes****Time Series, Dense Representations, and Sparse Representations of Temporal Data**

A time series is an ordered sequence of measurements of a variable that are arranged according to the time of occurrence. Time series are typically measured at some constant frequency and the data points are generally, but not necessarily, spaced at uniform time intervals.

The characteristic properties of a time series include the following:

- The data points are not independent of one another.
- The dispersion of data points varies as a function of time.
- The data frequently indicates trends.
- The data tends to be cyclic.

Typical business applications for time series analysis include the following:

- Budgetary analysis
- Economic forecasting
- Inventory analysis

- Process control
- Quality control
- Sales forecasting
- Stock market analysis
- Workload projections
- Yield projections

The EXPAND ON clause enables various forms of time series expansion on a PERIOD column value of an input row by producing a set of value-equivalent rows, one for each granule in the specified time period. The number of granules is defined by the anchor name you specify for the clause.

You can expand sparse PERIOD representations of relational data into a dense representation of the same data. Data converted to a dense form can be more easily manipulated by complex analyses such as moving average calculations without having to write complex SQL requests to respond to business questions made against sparse relational data.

The available forms of time series expansion for the EXPAND ON clause are the following.

- Interval expansion, where rows are expanded by user-specified intervals.
- Anchor point expansion, where rows are expanded by user-specified anchored points.
- Anchor PERIOD expansion, where rows are expanded by user-specified anchored periods.

## Rules and Restrictions for the EXPAND ON Clause

The rules and restrictions are:

- There are three different ways to expand rows:
  - By user-specified intervals such as 1 DAY, 3 MONTH, and so on.  
 You must specify an interval literal in the EXPAND ON clause for this form of expansion. This type of expansion is called an interval expansion.  
 An interval expansion can be useful for answering queries such as “compute the moving window average of inventory cost by week during the year 2010.”
  - By user-specified anchored points in a time line.  
 You must specify an anchored interval without specifying a PERIOD keyword for this form of expansion. Such expansion is called an anchor point expansion.  
 An anchor point expansion produces a specific date or time point in the expanded rows, where the date or time points are the anchor points present in the input period being expanded.  
 For example, BY ANCHOR MONTH\_BEGIN returns multiple expanded rows, one for each beginning value of a month present in the input period. This form of expansion might be useful for answering queries such as “get the month end average inventory cost during the last quarter of the year 2010.”
  - By user-specified anchored time durations in a time line, referred to as anchor period expansion.

You must specify an anchored interval with the PERIOD keyword for this form of expansion.

An anchor PERIOD expansion produces rows whose period beginning bounds are always aligned to specific DateTime values derived from the anchor names.

For example, BY ANCHOR PERIOD MONTH\_BEGIN, BY ANCHOR PERIOD MONDAY, and so on.

The PERIOD value of the row along with fixed durations enables weighted computations such as “compute the weekly weighted average of inventory cost.”

- You can use the EXPAND ON clause in query expressions, except for:

- Subqueries in search conditions. See [Specifying Subqueries in Search Conditions](#).

For example, the EXPAND ON clause in the following statement is not valid because of the reference to the column named pd1 in employee:

```
SELECT *
FROM employee
WHERE salary IN (SELECT salary
                  FROM salary_table
                  WHERE salary > 10000
                  EXPAND ON pd1);
```

- A WITH clause is specified in the query expression. See [WITH Clause](#).
- An updatable cursor SELECT statement. See *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148.
- A CREATE JOIN INDEX statement. See “CREATE JOIN INDEX” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.
- The query expression is a SELECT AND CONSUME statement. See [SELECT AND CONSUME](#).
- The expand expression can be a column or column alias from the select list or a number that specifies the positional sequence of the column or expression in the select list that is to be expanded.

If the specified column sequential position is not valid, for example, if the value is greater than the projected number of columns, Teradata Database returns an error.

The specified column must have a data type of PERIOD.

You must specify a table reference in a SELECT statement with an EXPAND ON clause.

- You can specify an EXPAND ON clause in any of the following places:

- Within a derived table.

This includes cases where the derived table is contained within a subquery.

See [Example: Expansion Over a UNION Operator](#).

- As part of the individual query expression of a SELECT statement that specifies set operators such as UNION, INTERSECT, or MINUS/EXCEPT.

See [Example: Expansion Over a UNION Operator](#).

- As part of the SELECT specification in the following SQL DDL statements. See CREATE RECURSIVE VIEW/REPLACE RECURSIVE VIEW seed and the recursive statements of the view definition, CREATE TABLE ... AS, and CREATE VIEW/REPLACE VIEW in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

Note that a view that contains an EXPAND ON clause in its definition is not updatable.

- As part of the SELECT specification in an INSERT ... SELECT statement. See [INSERT/INSERT ... SELECT](#).

- You cannot specify an EXPAND ON clause in the following situations:

- Anywhere in a SELECT AND CONSUME statement. See [SELECT AND CONSUME](#).
- Anywhere in an updatable cursor. See *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148.
- Anywhere in a subquery used as a search condition.
- Anywhere in a SELECT statement that specifies a TOP *n* operator. See [TOP \*n\*](#).

You can work around this restriction by specifying the TOP *n* operator within a derived table and then specifying the EXPAND ON clause in the outer query.

- Anywhere in a SELECT statement that specifies a SAMPLE clause. See [SAMPLE Clause](#).
- Anywhere in a SELECT statement that specifies a WITH clause. See [WITH Clause](#).
- Anywhere in a table function.
- As part of the SELECT specification in a CREATE JOIN INDEX statement. See “CREATE JOIN INDEX” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.
- As part of the SELECT specification in an INSERT... SELECT statement that includes the LOCAL ORDER BY option. This restriction also applies to an INSERT... SELECT statement included in a CREATE TRIGGER, REPLACE TRIGGER, CREATE MACRO, REPLACE MACRO, CREATE PROCEDURE, or REPLACE PROCEDURE statement.
- As part of the SELECT specification in an INSERT... SELECT statement where the target table is defined with NO PRIMARY INDEX. This restriction also applies to an INSERT... SELECT statement included in a CREATE TRIGGER, REPLACE TRIGGER, CREATE MACRO, REPLACE MACRO, CREATE PROCEDURE, or REPLACE PROCEDURE statement.

- All of the operations in a query expression except ORDER BY are performed before any rows are expanded.

The ORDER BY operation is then performed on the expanded rows if you specify an expanded column in the ORDER BY clause.

- If you do not specify a BY *expansion\_interval* clause, the expanding interval defaults to the granularity of the element type of the PERIOD value for the column.

The expansion defaults to the rules for interval literals described in [Rules and Restrictions for Interval Expansion](#).

- To limit the number of rows being expanded, specify a PERIOD expression in the FOR clause, where the PERIOD expression represents the period of interest.



The data type of the PERIOD expression must be compatible with the PERIOD data type of *expanded\_column\_name*.

If the specified period is not null and does not overlap with the row, then the row is not eligible for expansion.

- Teradata Database expands rows based on the interval you specify.

The interval can either be an interval constant, or an anchored interval derived from the anchor name such as any day of a week, the MONTH\_BEGIN keyword, or the MONTH\_END keyword.

- An EXPAND ON clause specification produces one value-equivalent row for each time granule derived from the specified *expansion\_interval*. Rows are said to be value-equivalent if the values in all columns except for their expanded columns are identical.
- If you specify a FOR clause, the expansion period is the overlapping period of the qualified row and the PERIOD constant you specify in the FOR clause. Otherwise, the expanding period is the PERIOD value of the selected row.
- If the expanding PERIOD expression specifies a column from another table that is not specified in the FROM clause, then Teradata Database joins the tables before the expansion.

You must ensure that the appropriate join condition is specified in such cases. If no other join conditions are specified, then the system performs a Product Join on the referenced tables.

See [Example: Join Before Expansion](#).

- When you do not specify an expanded column in the select list, but do specify the DISTINCT operator, the EXPAND operation is nullified.

See [Example: Nullified EXPAND Operation](#).

- If the expanding PERIOD is null, then the expanded row has a null expansion.

See [Example: Null Expansion Period Producing a Null Expanded Value](#).

## Rules and Restrictions for Interval Expansion

The rules and restrictions for the use of Interval Expansions are as follows:

- An expansion commences from the beginning value of the expansion period and ends at its ending value with an incremental value set to the specified interval literal for each row to be expanded.

In this process, the last expanded row may not always cover the expansion interval duration. If this occurs, the result is a partial period, which is a period having a duration that is less than the expansion interval.

- If one or more rows in the expanded result has an expansion period duration that is less than the specified interval, and the interval being expanded also has a higher granularity than the element type of the period being expanded, Teradata Database returns a warning to the requestor.

See [Example: Creating a Time Series Using Expansion By an Interval Constant Value](#).

- Unlike anchor period expansions, Teradata Database does not use a system-defined business calendar set for an interval literal in an EXPAND ON clause.

## Anchor Period and Anchor Point Expansion

### Time Option Not Allowed for PERIOD(DATE)

If the data type of the expansion expression is PERIOD(DATE) and you specify a time literal in the anchored interval, the request returns an error to the requestor. You cannot specify a time option when the data type of the expansion expression is PERIOD(DATE).

### PERIOD(TIME) and PERIOD(TIME WITH TIME ZONE) Not Valid for an Anchored EXPAND ON Clause

You cannot specify an anchored interval if the data type of the expansion expression is either PERIOD(TIME) or PERIOD(TIME WITH TIME ZONE). Otherwise, Teradata Database returns an error.

### Timestamp Data Type Uses the Time Literal Value that You Specify

If the element type of the expansion expression is a Timestamp data type, Teradata Database uses the time literal value that you specify to define the timestamp value of the anchor during expansion.

If you specify DAY for *anchor\_name*, the expansion interval is INTERVAL '1' DAY for each expanded row.

If you do not specify a time literal, the time literal value defaults to '00:00:00.000000+00:00' for an anchor name of ANCHOR\_MILLISECOND, ANCHOR\_SECOND, ANCHOR\_MINUTE, ANCHOR\_HOUR, WEEK\_BEGIN, MONTH\_BEGIN, QUARTER\_BEGIN, or YEAR\_BEGIN and to '23:59:59.999999+00:00' for any other anchor name.

The precision of the default value is set to the precision of the expansion expression. The default value includes the time zone value of +00:00 if the expansion expression specifies a time zone. Otherwise, the value is the session time zone and does not specify a time zone.

The anchor, for example MONTH\_BEGIN, is computed based on the session time zone. Thus, for two sessions that are at different time zones, the output can differ. For an example of this, see [Example: Same Expansion in Two Different Sessions in Different Time Zones](#).

### To Expand a Table by the First Day of the Month, specify MONTH\_BEGIN

To expand a table by the first calendar day of every month, specify MONTH\_BEGIN in the anchored interval clause.

Each expanded value for the row in the result has the first day of the corresponding month as the BEGIN bound and the expansion interval defaults to INTERVAL '1' MONTH. The BEGIN bound of the expanded value for each result row is 'YYYY-MM-01'.

See the first SELECT request in [Example: EXPAND ON MONTH\\_BEGIN and MONTH\\_END](#).

### To Expand a Table by the Last Day of the Month, Specify MONTH\_END

To expand a table by the last calendar day of every month, specify MONTH\_END in the anchored interval clause.

Each expanded value for the result row has the last day of the corresponding month as its BEGIN bound, and the expansion interval defaults to INTERVAL '1' MONTH.

The BEGIN bound of the expanded value for each result row is 'YYYY-MM-DD' where DD is one of 28, 29, 30, or 31, depending on the month and the year.

See the second SELECT request in [Example: EXPAND ON MONTH\\_BEGIN and MONTH\\_END](#).

### **To Expand a Table by a Specific Weekday, Specify the Weekday in the Anchored Interval Clause**

To expand a table by a particular weekday, specify the weekday you want to use in the anchored interval clause. The begin bound of each expanded value for the result row corresponds to that day of the week, and the expanding interval defaults to INTERVAL '7' DAY.

See [Example: Expansion on an Anchor Point Using WEEK\\_DAY](#).

## **Rules and Restrictions for Anchor Point Expansion of the EXPAND ON Clause**

The following cases are possible with Anchor Point Expansion when the expansion interval is longer than the granularity of the element type of the expand expression:

- There is no single anchor point in the expansion period.

For example, if the expansion is by MONDAY and the expansion period is PERIOD(DATE '2007-08-14', DATE '2007-08-17'), then the expansion period starts on Tuesday and ends on Friday in the same week.

In this case, the input row does not produce any expanded rows in the expanded result.

- The beginning bound of the expansion period is not aligned to an anchor point and its duration is longer than the expansion interval.

In this case, the expanded row set does not have a row corresponding to the beginning bound of the expansion period.

For example, if the expansion is by MONDAY and the expansion period is PERIOD(DATE '2007-08-15', DATE '2007-08-25'), the expanded result contains only one point, which is DATE '2007-08-20'.

- The beginning bound of the expansion period is aligned with an anchor point.

In this case, the expanded rows contain a row with the beginning bound of the expansion period.

For example, if the expansion is by MONDAY and the expansion period is PERIOD(DATE '2007-08-20', DATE '2007-08-25'), then the last expanded row contains DATE '2007-08-20'.

The expanded rows correspond to all anchor points (the BEGIN point of an anchor period) that are within the expansion period. Each row expands to as many anchor points that exist in the expansion period.

## Uses of Anchor Period Expansions

The expanded rows correspond to all anchor periods that overlap the expansion period. The key difference between anchor period expansion and anchor point expansion is that for anchor point expansion, the anchor point, which is the beginning bound of an anchor period, must occur within the expansion period, while for anchor period expansions, the anchor period must overlap the expansion period.

Following are the valid uses of anchor period expansions:

- The expansion period has a duration that is less than that of the expansion interval.

For example, if the expansion is by MONDAY and the expansion period is `PERIOD(DATE '2011-08-14', DATE '2011-08-17')`, then the expansion period starts on Tuesday, ends on Friday of the same week, and the expanded row is a singleton with a resulting anchor period of `PERIOD(DATE '2011-08-13', DATE '2011-08-20')`.

In this case, there is only one expanded row for the input row.

- The expansion period is not aligned with an anchor period, but the duration of the expansion period is greater than the expansion interval.

Such a row produces more than one expanded row in its result.

For example, if the expansion is by MONDAY and the expansion period is `PERIOD(DATE '2011-08-15', DATE '2011-08-25')`.

In this case, the expanded result has two anchor periods, `PERIOD(DATE '2011-08-13', DATE '2011-08-20')` and `PERIOD(DATE '2011-08-20', DATE '2011-08-27')`.

- The expanding period is aligned with an anchor period and its duration is an exact multiple of the expansion interval.

For example, if the expansion is by MONDAY and the expansion period is `PERIOD(DATE '2011-08-20', DATE '2011-08-27')`, then the expanded row contains `PERIOD(DATE '2011-08-20', DATE '2011-08-27')`.

## Anchored Interval Uses System-defined Business Calendar Set in the Session

Teradata Database uses a system-defined business calendar set in the session for the anchored interval in an `EXPAND ON` clause.

In an anchor expansion, Teradata Database derives the `BEGIN` bound of the first expanded row from the business calendar set in the current session. The `BEGIN` bound of the first expanded row is derived from the `YearBeginDate`, `WeekStart`, and `CalendarPeriod` columns from the `BusinessCalendarPattern` table.

For example, a row has period from 1-Jan-2008 to 31-May-2008, `YearBeginDate` is 15-Mar-2008, and `CalendarPeriod` is from 1-Jan-08 to 30-Oct-08. Expanding by anchor point `MONTH_BEGIN` results in 4 rows, where the first row is from 15-Jan-08 to 15-Feb-08 and second row is from 15-Feb-08 to 15-Mar-08

and third row is from 15-Mar-08 to 15-Apr-08 and the forth row is from 15-Apr-08 to 15-May-08. The MONTH\_BEGIN value is derived from the YearBeginDate value, which is January 15 and, because the Calendar begins in January, the first month starts on January 15.

## Anchor Names

This table lists the EXPAND ON clause anchor names:

Anchor Name	Expanding Interval	Default Time Literal	EXPAND Result
ANCHOR_MILLISECOND	INTERVAL '1' MILLISECOND	00:00:00. 000000+00:00	Each expanded value for the row in the result adjusts the anchor to the nearest millisecond boundary.
ANCHOR_SECOND	INTERVAL '1' SECOND	00:00:00. 000000+00:00	Each expanded value for the row in the result adjusts the anchor to the nearest second boundary.
ANCHOR_MINUTE	INTERVAL '1' MINUTE	00:00:00. 000000+00:00	Each expanded value for the row in the result adjusts the anchor to the nearest minute boundary.
ANCHOR_HOUR	INTERVAL '1' HOUR	00:00:00. 000000+00:00	Each expanded value for the row in the result adjusts the anchor to the nearest hour boundary.
WEEK_BEGIN	INTERVAL '7' DAY	00:00:00. 000000+00:00	Each expanded value for the row in the result has the first day of the corresponding week as its BEGIN bound.
WEEK_END		23:59:59. 999999+00:00	Each expanded value for the row in the result has the last day of the corresponding week as its BEGIN bound.
MONTH_BEGIN	INTERVAL '1' MONTH	00:00:00. 000000+00:00	Each expanded value for the row in the result has the first day of the corresponding month as its BEGIN bound.
MONTH_END		23:59:59. 999999+00:00	Each expanded value for the row in the result has the last day of the corresponding month as its BEGIN bound.
QUARTER_BEGIN	INTERVAL '3' MONTH	00:00:00. 000000+00:00	Each expanded value for the row in the result has the first day of the corresponding quarter as its BEGIN bound.
QUARTER_END		23:59:59. 999999+00:00	Each expanded value for the row in the result has the last day of the

Anchor Name	Expanding Interval	Default Time Literal	EXPAND Result
			corresponding quarter as its BEGIN bound.
YEAR_BEGIN	INTERVAL '1' YEAR	00:00:00. 000000+00:00	Each expanded value for the row in the result has the first day of the corresponding year as its BEGIN bound.
YEAR_END		23:59:59. 999999+00:00	Each expanded value for the row in the result has the last day of the corresponding year as its BEGIN bound.

## Anchor Results Can Differ Between the ISO Calendar and the Teradata or COMPATIBLE Calendars

For information on how the ISO calendar computes results, see “About ISO Computation” in *Teradata Vantage™ SQL Date and Time Functions and Expressions*, B035-1211.

The following anchors can return different results for the ISO calendar than for the Teradata or COMPATIBLE calendars.

- MONTH\_BEGIN
- MONTH\_END
- QUARTER\_BEGIN
- QUARTER\_END
- YEAR\_BEGIN
- YEAR\_END

## Expand Does Not Consider Exceptions

Expand does not consider exceptions even for a first date.

For example, if you want to produce an expansion by business days, you must perform a normal expansion and then apply business functions on all of the expanded rows using a derived table to get the desired expansion by business days.

## Expand Interval Literal Expressions

The following table explains the meanings of the expand interval syntax variables for the EXPAND ON clause.

Expand Interval	Description
<i>interval_literal</i>	To perform an interval expansion, specify any valid Interval literal value (see <i>Teradata Vantage™ Data Types and Literals</i> , B035-1143 for a comprehensive list of valid Interval literals).
ANCHOR <i>anchor_name</i>	To perform an anchor point expansion, specify an anchor name, but do not specify PERIOD.
ANCHOR PERIOD <i>anchor_name</i>	To perform an anchor period expansion, you must specify PERIOD and an anchor name.

## Anchor Name Literal Expressions

The following table lists the ANCHOR *anchor\_name* variables for the EXPAND ON clause.

Anchor Name	Description
ANCHOR_MILLISECOND	Produces multiple expanded rows, one for each millisecond in the input period.
ANCHOR_SECOND	Produces multiple expanded rows, one for each second in the input period.
ANCHOR_MINUTE	Produces multiple expanded rows, one for each minute in the input period.
ANCHOR_HOUR	Produces multiple expanded rows, one for each hour in the input period.
DAY	The expansion interval is INTERVAL '1' DAY for each expanded row.
WEEK_BEGIN	Produces multiple expanded rows, one for each beginning value of a week present in the input period.
WEEK_END	Produces multiple expanded rows, one for each ending value of a week present in the input period.
MONTH_BEGIN	Produces multiple expanded rows, one for each beginning value of a month present in the input period. An example query that uses such expansion is "Get the month end average inventory cost during the last quarter of the year 2010."
MONTH_END	Produces multiple expanded rows, one for each ending value of a month present in the input period.
QUARTER_BEGIN	Produces multiple expanded rows, one for each beginning value of a quarter present in the input period.
QUARTER_END	Produces multiple expanded rows, one for each ending value of a quarter present in the input period.
YEAR_BEGIN	Produces multiple expanded rows, one for each beginning value of a year present in the input period.
YEAR_END	Produces multiple expanded rows, one for each ending value of a year present in the input period.

Anchor Name	Description
MONDAY	An anchor period expansion produces rows whose period beginning bounds are always aligned to specific DateTime values derived from the anchor names. For this specification, the period beginning bound is the first Monday in the specified period.
TUESDAY	The period beginning bound is the first Tuesday in the specified period.
WEDNESDAY	The period beginning bound is the first Wednesday in the specified period.
THURSDAY	The period beginning bound is the first Thursday in the specified period.
FRIDAY	The period beginning bound is the first Friday in the specified period.
SATURDAY	The period beginning bound is the first Saturday in the specified period.
SUNDAY	The period beginning bound is the first Sunday in the specified period.

## WEEK\_BEGIN Anchor Name and WeekStart Anchor Expansion

The WEEK\_BEGIN anchor name corresponds to an anchor expansion by WeekStart.

For example, if the business calendar for the session is ISO and you specify expansion by WEEK\_BEGIN, Teradata Database translates the WEEK\_BEGIN expansion to EXPAND ON ....BY ANCHOR MONDAY because the week begins on Monday.

## Examples

### Example: Expansion on an Anchor Point Using WEEK\_DAY

Create a table named tdate with the following definition.

```
CREATE SET TABLE tdate (
  id      INTEGER,
  quantity INTEGER,
  pd      PERIOD(DATE))
PRIMARY INDEX (id);
```

You insert two rows into tdate so its contents are as follows.

id	quantity	pd
11	110	2005-02-03, 2005-06-20
10	100	2004-01-03, 2004-05-20



Submit a SELECT statement against *tdate* that specifies an EXPAND ON clause anchored by a day of the week, Monday, so that each expanded row begins from a Monday, as specified in the statement, and the duration of each expanded period is seven days.

```
SELECT id, BEGIN(bg)
FROM tdate
EXPAND ON pd AS bg BY ANCHOR MONDAY;
```

Teradata Database returns *tdate* details for each week of a given period, beginning on the first Monday from the eligible data, as you specified in the BY ANCHOR clause of the statement.

Because the first row in *tdate* starts on a Thursday, not a Monday, the expanded row starts on the *next* sequential Monday date, which is February 7, and then continues in weekly granular increments.

id	begin(bg)
--	-----
11	2005-02-07
11	2005-02-14
11	2005-02-21
11	2005-02-28
11	2005-03-07
11	2005-03-14
11	2005-03-21
11	2005-03-28
11	2005-04-04
11	2005-04-11
11	2005-04-18
11	2005-04-25
11	2005-05-02
11	2005-05-09
11	2005-05-16
11	2005-05-23
11	2005-05-30
11	2005-06-06
11	2005-06-13
10	2004-01-05

10	2004-01-12
10	2004-01-12
10	2004-01-19
10	2004-01-26
10	2004-02-02
10	2004-02-09
10	2004-02-16
10	2004-02-23
10	2004-03-01
10	2004-03-08
10	2004-03-15
10	2004-03-22
10	2004-03-29
10	2004-04-05
10	2004-04-12
10	2004-04-19
10	2004-04-26
10	2004-05-03
10	2004-05-10
10	2004-05-17

### Example: Expansion on an Anchor Point Using ANCHOR\_SECOND

Create a table named t2 with the following definition.

```
CREATE TABLE t2 (
  id      INTEGER
  quantity INTEGER
  pd      PERIOD(DATE))
PRIMARY INDEX (id);
```

The input row for the example statement is of type TIMESTAMP and contains the following timestamp period data. (2011-01-01 10:15:20.000001, 2011-01-01 10:15:25.000009).

Submit the following SELECT statement on table *t2*.

```
SELECT BEGIN(expd)
FROM t2
EXPAND ON pd AS expd BY ANCHOR ANCHOR_SECOND;
```

The result set contains 5 rows because the input row has a period of 5 seconds, beginning with 10:15:**20**.000001 and ending with 10:15:**25**.000009.

begin expd
2011-01-01 10:15:21.000000
2011-01-01 10:15:22.000000
2011-01-01 10:15:23.000000
2011-01-01 10:15:24.000000
2011-01-01 10:15:25.000000

## Example: Expansion Over a UNION Operator

Suppose you create a table named *tdate1* with the following definition.

```
CREATE SET TABLE tdate1 (
  id      INTEGER,
  quantity INTEGER,
  pd      PERIOD(DATE))
PRIMARY INDEX (id);
```

Table *tdate1* contains the following rows.

id	quantity	pd
12	120	2006-02-03, 2006-06-20
13	130	2005-01-03, 2005-05-20

You now submit the following unioned SELECT statement against *tdate*, as defined in [Example: Expansion on an Anchor Point Using WEEK\\_DAY](#), and *tdate1* that specifies an EXPAND ON clause for both statements, each having a one month interval granularity.

```
SELECT id, quantity, expd
FROM tdate
EXPAND ON pd AS expd BY INTERVAL '1' MONTH
UNION
SELECT id, quantity, expd
```

```
FROM tdate1
EXPAND ON pd AS expd BY INTERVAL '1' MONTH;
```

In this example, Teradata Database first expands the rows in tables tdate and tdate1 and then unions the resulting rows from the queries on the expanded results.

Teradata Database returns a warning that some rows in the expanded result might have an expanded period duration that is less than the duration of the specified interval.

Teradata Database returns tdate1 details for each month of a given period for these two queries and then unions the result rows as follows.

id	quantity	expd
11	110	2005-02-03, 2005-03-03
11	110	2005-03-03, 2005-04-03
11	110	2005-04-03, 2005-05-03
11	110	2005-05-03, 2005-06-03
11	110	2005-06-03, 2005-06-20
10	100	2004-01-03, 2004-02-03
10	100	2004-02-03, 2004-03-03
10	100	2004-03-03, 2004-04-03
10	100	2004-04-03, 2004-05-03
10	100	2004-05-03, 2004-05-20
12	120	2006-02-03, 2006-03-03
12	120	2006-03-03, 2006-04-03
12	120	2006-04-03, 2006-05-03
12	120	2006-05-03, 2006-06-03
12	120	2006-06-03, 2006-06-20
13	130	2005-01-03, 2005-02-03
13	130	2005-02-03, 2005-03-03
13	130	2005-03-03, 2005-04-03
13	130	2005-04-03, 2005-05-03
13	130	2005-05-03, 2005-05-20

## Example: EXPAND ON MONTH\_BEGIN and MONTH\_END

This example shows how to use MONTH\_BEGIN and MONTH\_END in an EXPAND ON clause.

First create the timestamp table as follows.

```
CREATE SET TABLE ttimestamp (
  id      INTEGER,
  quantity INTEGER,
  pd      PERIOD(TIMESTAMP(0)))
PRIMARY INDEX (id);
```

Table ttimestamp contains the following rows.

id	quantity	pd
11	110	2005-02-01 01:10:40, 2005-06-20 05:20:50
10	100	2004-01-03 01:10:40, 2004-05-31 20:20:50

When you specify an EXPAND ON clause by MONTH\_BEGIN or MONTH\_END, every expanded row starts from either the MONTH\_BEGIN value or from the MONTH\_END value for that month, and the granularity of each expanded period is one month. In this example, the table data is shown with the default session time zone set to INTERVAL '00:00' HOUR TO MINUTE.

```
SET TIME ZONE INTERVAL '00:00' HOUR TO MINUTE;
```

The following SELECT statement specifies an EXPAND ... BY MONTH\_BEGIN.

```
SELECT id, quantity, BEGIN(bg)
FROM ttimestamp
EXPAND ON pd AS bg BY ANCHOR MONTH_BEGIN;
```

Each row is expanded at the default time literal value 00:00:00+00:00 for each MONTH\_BEGIN value.

id	qty	bg
11	110	2005-03-01 00:00:00
11	110	2005-04-01 00:00:00
11	110	2005-05-01 00:00:00
11	110	2005-06-01 00:00:00
10	100	2004-02-01 00:00:00
10	100	2004-03-01 00:00:00
10	100	2004-04-01 00:00:00

10	100	2004-05-01 00:00:00
----	-----	---------------------

The following SELECT statement specifies an EXPAND ... BY ANCHOR MONTH\_END, but is otherwise identical to the previous statement.

```
SELECT id, quantity, BEGIN(bg)
FROM ttimestamp
EXPAND ON pd AS bg BY ANCHOR MONTH_END;
```

Each row is expanded at the default time literal value 23:59:59+00:00 at each month end.

id	quantity	bg
11	110	2005-02-28 23:59:59
11	110	2005-03-31 23:59:59
11	110	2005-04-30 23:59:59
11	110	2005-05-31 23:59:59
10	100	2004-01-31 23:59:59
10	100	2004-02-29 23:59:59
10	100	2004-03-31 23:59:59
10	100	2004-04-30 23:59:59

## Example: EXPAND ON and DISTINCT

This example shows how Teradata Database performs the DISTINCT operation after expansion occurs.

```
CREATE SET TABLE products (
  product_id      INTEGER,
  product_price   DECIMAL(5,2),
  product_duration PERIOD(DATE))
PRIMARY INDEX (product_id);
```

Assume that you have the following rows in products.

product_id -----	product_price -----	product_duration -----
1000	100.00	2007-02-15, 2007-08-11
1001	99.99	2007-03-04, 2007-05-01
1001	101.10	2008-05-10, 2009-05-10

1001	1-4.10	2007-07-16, 2008-10-09
------	--------	------------------------

When you specify a DISTINCT operator in the query expression and the expanded column in the select list of your query, Teradata Database performs the DISTINCT operation after expansion and removes the duplicate rows from the expanded result (see the example below).

```
SELECT DISTINCT product_id, pd
FROM products
EXPAND ON product_duration AS pd BY ANCHOR PERIOD MONTH_BEGIN;
```

This SELECT statement returns the following response set.

product_id	pd
1000	2007-02-01, 2007-03-01
1000	2007-03-01, 2007-04-01
1000	2007-04-01, 2007-05-01
1000	2007-05-01, 2007-06-01
1000	2007-06-01, 2007-07-01
1000	2007-07-01, 2007-08-01
1000	2007-08-01, 2007-09-01
1001	2007-03-01, 2007-04-01
1001	2007-04-01, 2007-05-01
1001	2008-05-01, 2008-06-01
1001	2008-06-01, 2008-07-01
1001	2008-07-01, 2008-08-01
1001	2008-08-01, 2008-09-01
1001	2008-09-01, 2008-10-01
1001	2008-10-01, 2008-11-01
1001	2008-11-01, 2008-12-01
1001	2008-12-01, 2009-01-01
1001	2009-01-01, 2009-02-01
1001	2009-02-01, 2009-03-01
1001	2009-03-01, 2009-04-01
1001	2009-04-01, 2009-05-01

1001	2009-05-01, 2009-06-01
1001	2007-07-01, 2007-08-01
1001	2007-08-01, 2007-09-01
1001	2007-09-01, 2007-10-01
1001	2007-10-01, 2007-11-01
1001	2007-11-01, 2007-12-01
1001	2007-12-01, 2008-01-01
1001	2008-01-01, 2008-02-01
1001	2008-02-01, 2008-03-01
1001	2008-03-01, 2008-04-01
1001	2008-04-01, 2008-05-01

## Example: Same Expansion in Two Different Sessions Using Different Time Zone Intervals

The beginning bound of the expanding period value is adjusted to start at the time specified by *time\_literal* before it expands the rows. The anchor point, for example MONTH\_BEGIN, is computed based on the session time zone. As a result, the output can be different for two sessions that are at two different time zones.

In this example the time zone for the first session is set to INTERVAL '-01:00' HOUR TO MINUTE, and the time zone for the second session is set to INTERVAL '02:00' HOUR TO MINUTE.

You set the time zone for the first session as follows and then submit the indicated SELECT statement anchored on MONTH\_BEGIN:

```
SET TIME ZONE INTERVAL '-01:00' HOUR TO MINUTE;
SELECT id, quantity, BEGIN(pd) AS bg
FROM ttimestamp
EXPAND ON PERIOD bg BY ANCHOR MONTH_BEGIN;
```

id	quantity	bg
11	110	2005-03-01 00:00:00
11	110	2005-04-01 00:00:00
11	110	2005-05-01 00:00:00
11	110	2005-06-01 00:00:00
10	100	2004-02-01 00:00:00



10	100	2004-03-01 00:00:00
10	100	2004-04-01 00:00:00
10	100	2004-05-01 00:00:00

The output of the same SELECT statement submitted in the second session returning one additional row (shaded in orange):

```
SET TIME ZONE INTERVAL '02:00' HOUR TO MINUTE;
SELECT id, quantity, BEGIN(pd) AS bg
FROM ttimestamp
EXPAND ON PERIOD bg BY ANCHOR MONTH_BEGIN;
```

id	quantity	bg
11	110	2005-02-01 00:00:00
11	110	2005-03-01 00:00:00
11	110	2005-04-01 00:00:00
11	110	2005-05-01 00:00:00
11	110	2005-06-01 00:00:00
10	100	2004-02-01 00:00:00
10	100	2004-03-01 00:00:00
10	100	2004-04-01 00:00:00
10	100	2004-05-01 00:00:00E

## Example: EXPAND ON and OLAP Functions

This example shows the use of the EXPAND ON clause in a SELECT statement that also specifies an OLAP function. In this case, the OLAP function specified is RANK. For details about the RANK function, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

First create the player\_history table as follows.

```
CREATE SET TABLE player_history (
  player_id INTEGER,
  duration PERIOD(DATE),
  grade CHARACTER(1))
PRIMARY INDEX (player_id);
```

The player\_history table contains the following rows.

player_id	duration	grade
1000	2007-06-03, 2007-07-03	A
1000	2007-07-03, 2007-08-03	B
1000	2007-08-03, 2007-09-03	C
1001	2007-07-03, 2007-08-03	A
1001	2007-08-03, 2007-09-03	D
1001	2007-09-03, 2007-10-03	E

The following SELECT statement specifies the RANK OLAP function in the select list.

```
SELECT playerid, BEGIN(expd), RANK(grade ASC) AS a, grade
FROM player_history
WHERE player_id = 1000 QUALIFY a < 3
EXPAND ON duration AS expd BY ANCHOR MONDAY;
```

The query returns the following response set.

player_id	BEGIN(expd)	a	grade
1000	2007-06-04	1	A
1000	2007-06-11	1	A
1000	2007-06-18	1	A
1000	2007-06-25	1	A
1000	2007-07-02	1	A
1000	2007-07-09	2	B
1000	2007-07-16	2	B
1000	2007-07-23	2	B
1000	2007-07-30	2	B

## Example: Same Expansion in Two Different Sessions in Different Time Zones

The beginning bound of the expanding period value is adjusted to start at the time specified by *time\_literal* before it expands the rows. The anchor point, for example MONTH\_BEGIN, is computed based on the session time zone. As a result, the output can be different for two sessions that are at two different time zones.

In the following example, the time literal defaults to 00:00:00 at the session time zone because the EXPAND ON clause input row does not specify a time zone, because the value of duration has a PERIOD(TIMESTAMP) data type. After the time literal is converted to UTC, it is the previous day. Therefore, the previous day-to-month begin is checked with the row, and when it is returned, Teradata Database adds the session time.

First you create the following table.

```
CREATE SET TABLE test (
  testid  INTEGER,
  duration PERIOD(TIMESTAMP))
PRIMARY INDEX (testid);
```

Table test contains the following row.

testid	duration (at UTC)
ABC	2002-01-31 15:30:00, 2002-05-31 15:00:00

You then perform the following anchor point expansion by MONTH\_BEGIN with a default time zone literal.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT BEGIN(xyz)
FROM test
EXPAND ON duration AS xyz BY ANCHOR MONTH_BEGIN;
```

This statement returns the following rows:

BEGIN(xyz)
2002-03-01 00:00:00
2002-04-01 00:00:00
2002-05-01 00:00:00

In the following example, the time literal is 20:00:00 at session time zone because the input row does not specify a time zone, because the value of duration has a PERIOD(TIMESTAMP) data type. After the time literal is converted to UTC, it is the next day. Therefore, Teradata Database checks the next day to month end with the row, and when it returns the row, it adds the session time.

You have the following table:

```
CREATE SET TABLE test1 (
  testid  INTEGER,
  duration PERIOD(TIMESTAMP))
PRIMARY INDEX (testid);
```

Table test1 contains the following row:

testid	duration (at UTC)
ABC	2005-12-03 04:30:00, 2006-04-01

You perform the following anchor period expansion by MONTH\_END with a time literal and with a default session time zone.

```
SET TIME ZONE INTERVAL -'07:00' HOUR TO MINUTE;
SELECT BEGIN(xyz)
FROM test1
EXPAND ON duration AS xyz BY ANCHOR MONTH_END AT TIME '20:00:00';
```

BEGIN(XYZ)
2005-12-31 20:00:00
2006-01-31 20:00:00
2006-02-28 20:00:00

In the next example, the time literal value is 07:00:00 at time zone +10:00. After the time literal is converted to UTC, the time is 21:00 on the previous day. Therefore, Teradata Database checks the previous day to month end value with the row and, when the time series value is returned by the statement, Teradata Database adds the session time, which is 00:00.

You have the following table.

```
CREATE SET TABLE test2 (
  testid  INTEGER,
  duration PERIOD(TIMESTAMP))
PRIMARY INDEX (testid);
```

Table test2 contains the following row.

testid	duration (at UTC)
timeseries	2005-12-30 22:30:00, 2006-04-29 18:00:00

You perform the following anchor period expansion by MONTH\_END, specifying both a time literal and a time zone.

```
SET TIME ZONE INTERVAL '00:00' HOUR TO MINUTE;
SELECT timeseries
FROM test2
EXPAND ON duration AS timeseries BY ANCHOR PERIOD MONTH_END
AT TIME '07:00:00+10:00';
```

This statement returns the following row set.

timeseries
2005-12-30 21:00:00, 2006-01-30 21:00:00
2006-01-30 21:00:00, 2006-02-27 21:00:00
2006-02-27 21:00:00, 2006-03-30 21:00:00
2006-03-30 21:00:00, 2006-04-29 21:00:00

## Example: EXPAND ON With a Join

This example shows using a join with the EXPAND ON clause.

First create the student table.

```
CREATE SET TABLE student (
  id          INTEGER,
  name        CHARACTER(10) CHARACTER SET LATIN
              NOT CASESPECIFIC,
  duration_begin_end PERIOD(DATE))
PRIMARY INDEX (id);
```

The student table contains the following row.

id	name	duration_begin_end
101	ABC	2004-01-01, 2004-12-31

Now create the course table.

```
CREATE SET TABLE course (
  name        CHARACTER(10) CHARACTER SET LATIN
              NOT CASESPECIFIC,
  student_id   INTEGER,
  course_begin_end PERIOD(DATE))
PRIMARY INDEX (name);
```

The course table contains the following rows.

name	student_id	course_begin_end
CPP	101	2004-08-01,2004-08-30
Java	101	2004-07-01,2004-07-30
C	101	2004-04-01,2004-06-30

Submit a SELECT statement that returns the month a student was enrolled in a particular course.

This statement joins the student table (expanded on an interval literal and aliased as dt) with the course table using a mix of equality and inequality predicates in its WHERE clause.

```
SELECT course.name, EXTRACT(MONTH FROM BEGIN(expd))
FROM (SELECT student_id, expd
      FROM student
      EXPAND ON duration_begin_end AS expd BY INTERVAL '1' MONTH)
      AS dt, course AS c
WHERE c.student_id = dt.id
AND   (BEGIN(c.course_begin_end) < END(expd))
AND   BEGIN(expd) < END(c.course_begin_end)
AND   dt.id = 101;
```

This statement returns the following five rows.

course-name	extract(month from expd)
C	4
C	5
C	6
CPP	8
Java	7

Teradata Database also returns a 9308 warning message for this statement.

## Example: EXPAND ON For an Anchored Interval

This example shows the use of an anchored interval for doing anchor period and anchor point expansions. For an anchor period expansion, the expanded period value must overlap the expanding period, while for an anchor point expansion, the begin value of the expanded period value must be contained in the expanding period, which is a more restrictive condition.

First create the sold\_products table.

```
CREATE SET TABLE sold_products, NO FALLBACK (
  product_id      INTEGER,
  product_price    DECIMAL(10,2),
  product_duration PERIOD(DATE))
PRIMARY INDEX (product_id);
```

The sold\_products table contains the following rows.

product_id	product_price	product_duration
1000	100.00	2007-02-15, 2007-08-11
1001	99.99	2007-03-04, 2007-05-01

The following SELECT statement specifies an anchor period of MONTH\_BEGIN. This is an anchor period expansion.

```
SELECT product_id, product_price, product_duration, expd
FROM sold_products
EXPAND ON product_duration AS expd BY ANCHOR PERIOD MONTH_BEGIN;
```

The statement returns the following nine rows, with the original two rows highlighted in red:

product_id	product_price	product_duration	expd
1000	100.00	2007-02-15,2007-08-11	2007-02-01,2007-03-01
1000	100.00	2007-02-15,2007-08-11	2007-03-01,2007-04-01
1000	100.00	2007-02-15,2007-08-11	2007-04-01,2007-05-01
1000	100.00	2007-02-15,2007-08-11	2007-05-01,2007-06-01
1000	100.00	2007-02-15,2007-08-11	2007-06-01,2007-07-01
1000	100.00	2007-02-15,2007-08-11	2007-07-01,2007-08-01
1000	100.00	2007-02-15,2007-08-11	2007-08-01,2007-09-01
1001	99.99	2007-03-04,2007-05-01	2007-03-01,2007-04-01
1001	99.99	2007-03-04,2007-05-01	2007-04-01,2007-05-01

For an anchor point expansion done on the same data, the shaded rows would not appear, as the following example shows.

The following table describes the difference between anchor point and anchor period expansions.

Expansion Type	Description
Anchor period expansion	Expanded period value must overlap the expanding period.
Anchor point expansion	Begin value of the expanded period value must be contained within the expanding period.

Submit the following SELECT statement, which differs from the previous statement only in specifying the BEGIN bound function on product\_duration instead of simply specifying the column name. This is an anchor point expansion done on the same data as the previous anchor period expansion.

```
SELECT product_id, product_price, product_duration, BEGIN(expd)
FROM sold_products
EXPAND ON product_duration AS expd BY ANCHOR MONTH_BEGIN;
```

This statement returns seven rows, rather than nine, with the rows shaded in red from the previous example not appearing in the result set.

product_id	product_price	product_duration	begin(expd)
1000	100.00	2007-02-15,2007-08-11	2007-03-01
1000	100.00	2007-02-15,2007-08-11	2007-04-01
1000	100.00	2007-02-15,2007-08-11	2007-05-01
1000	100.00	2007-02-15,2007-08-11	2007-06-01
1000	100.00	2007-02-15,2007-08-11	2007-07-01
1000	100.00	2007-02-15,2007-08-11	2007-08-01
1001	99.99	2007-03-04,2007-05-01	2007-04-01

## Example: EXPAND ON and Span Grouping

This example shows the use of the EXPAND ON clause with grouping on a span of entries from the select list.

First create the stock table.

```
CREATE SET TABLE stock (
  stock_id      INTEGER,
  stock_quantity INTEGER,
  begin_end_date PERIOD(DATE))
PRIMARY INDEX (stockid);
```

The stock table contains the following rows.

stock_id	stock_quantity	begin_end_date
100	200	2005-10-10,2005-11-15
101	20	2005-06-01,2005-08-31

This example shows how you can compute a weighted average for the stock\_quantity column on a monthly basis.



Assume that `udf_agspan` is an aggregate UDF that adds the stock quantity for given month of a year and then divides the sum by the number of days in that month. This provides a different result when compared to the `AVG` function when the row is not spanning the whole month.

```
SELECT udf_agspan(stock_quantity,
  EXTRACT(YEAR FROM BEGIN(expdcol)),
  EXTRACT(MONTH FROM BEGIN(expdcol)))
  (FORMAT '-----9.999') AS wavg,
  EXTRACT(YEAR FROM BEGIN(expdcol) AS yr,
  EXTRACT(MONTH FROM BEGIN(expdcol) AS mn,
  stock_id
FROM (SELECT stock.*, expdcol
  FROM stock
  EXPAND ON begin_end_date AS expdcol BY INTERVAL '1'DAY) AS dt
GROUP BY 2,3,4;
```

This statement returns the following rows.

wavg	yr	mn	stock_id
141.935	2005	10	100
93.333	2005	11	100
20.000	2005	06	101
20.000	2005	07	101
19.355	2005	08	101

## Example: EXPAND ON for a Moving Average

This example shows how to create a moving average, which is a common method of smoothing time series data, on the data in the price column.

First create the `stk` table.

```
CREATE SET TABLE stk (
  stock_id INTEGER,
  price     FLOAT,
  validity  PERIOD(TIMESTAMP))
PRIMARY INDEX (stock_id);
```

The `stk` table contains the following set of 30 rows.

stock_id	price	validity
1000	10.00	2006-01-01 09:00:00 , 2006-01-01 12:00:00

1000	11.00	2006-01-01 12:00:00 , 2006-01-01 15:00:00
1000	9.00	2006-01-01 15:00:00 , 2006-01-01 18:00:00
1000	12.00	2006-01-02 09:00:00 , 2006-01-02 12:00:00
1000	13.00	2006-01-02 12:00:00 , 2006-01-02 15:00:00
1000	14.00	2006-01-02 15:00:00 , 2006-01-02 18:00:00
1000	8.00	2006-01-03 09:00:00 , 2006-01-03 12:00:00
1000	5.00	2006-01-03 12:00:00 , 2006-01-03 15:00:00
1000	15.00	2006-01-03 15:00:00 , 2006-01-03 18:00:00
1000	19.00	2006-01-04 09:00:00 , 2006-01-04 12:00:00
1000	16.00	2006-01-04 12:00:00 , 2006-01-04 15:00:00
1000	16.00	2006-01-04 15:00:00 , 2006-01-04 18:00:00
1000	16.00	2006-01-05 09:00:00 , 2006-01-05 12:00:00
1000	16.00	2006-01-05 12:00:00 , 2006-01-01 15:00:00
1000	16.00	2006-01-05 15:00:00 , 2006-01-05 18:00:00
1001	20.00	2006-01-01 09:00:00 , 2006-01-01 12:00:00
1001	21.00	2006-01-01 12:00:00 , 2006-01-01 15:00:00
1001	19.00	2006-01-01 15:00:00 , 2006-01-01 18:00:00
1001	22.00	2006-01-02 09:00:00 , 2006-01-02 12:00:00
1001	23.00	2006-01-02 12:00:00 , 2006-01-02 15:00:00
1001	24.00	2006-01-02 15:00:00 , 2006-01-02 18:00:00
1001	18.00	2006-01-03 09:00:00 , 2006-01-03 12:00:00
1001	15.00	2006-01-03 12:00:00 , 2006-01-03 15:00:00
1001	25.00	2006-01-03 15:00:00 , 2006-01-03 18:00:00
1001	29.00	2006-01-04 09:00:00 , 2006-01-04 12:00:00
1001	26.00	2006-01-04 12:00:00 , 2006-01-04 15:00:00
1001	26.00	2006-01-04 15:00:00 , 2006-01-04 18:00:00
1001	26.00	2006-01-05 09:00:00 , 2006-01-05 12:00:00
1001	24.00	2006-01-05 12:00:00 , 2006-01-01 15:00:00
1001	24.00	2006-01-05 15:00:00 , 2006-01-05 18:00:00

This example returns a moving average of stock over a three day period.

```
SELECT stock_id, CAST (p AS DATE), AVG(price)
      OVER (PARTITION BY stock_id
            ORDER BY p ROWS
            2 PRECEDING)
FROM (SELECT stock_id, price, BEGIN(p)
      FROM stk
      EXPAND ON validity AS p
            BY ANCHOR DAY AT TIME '17:59:59'
            FOR PERIOD(TIMESTAMP '2006-01-01 17:59:59',
                      TIMESTAMP '2006-01-05 18:00:00')) AS dt;
```

This statement returns the following 10 rows with a moving average over *price*.

stock_id	CAST(p AS DATE)	AVG(price)
1000	2006-01-01	9.000000000000000E 000
1000	2006-01-02	1.150000000000000E 001
1000	2006-01-03	1.266666666666667E 001
1000	2006-01-04	1.500000000000000E 001
1000	2006-01-05	1.566666666666667E 001
1001	2006-01-01	1.900000000000000E 001
1001	2006-01-02	2.150000000000000E 001
1001	2006-01-03	2.266666666666667E 001
1001	2006-01-04	2.500000000000000E 001
1001	2006-01-05	2.500000000000000E 001

You can produce the same result without using an EXPAND ON clause by joining with a calendar table, as in the following SELECT statement.

```
SELECT stock_id, CAST(CAST(p AS TIMESTAMP) AS DATE),
      AVG(price) OVER (PARTITION BY stock_id
            ORDER BY p ROWS
            2 PRECEDING)
FROM stk, (SELECT (calendar_date (FORMAT 'yyyy-mm-dd')) || ' ' ||
      FROM sys_calendar.calendar
      WHERE calendar_date BETWEEN DATE '2006-01-01'
            AND DATE '2006-01-06') AS dt(p)
WHERE BEGIN(validity) <= p
AND p < END(validity)) AS expnd;
```

## Example: EXPAND ON for a WEEK\_BEGIN Anchor Point

This example expands employee using a WEEK\_BEGIN anchor point.

Assume the following table definition.

```
CREATE SET TABLE employee, NO FALLBACK (
  eid          INTEGER,
  ename        CHARACTER(20) CHARACTER SET LATIN NOT CASESPECIFIC,
  jobperiod    PERIOD(DATE))
PRIMARY INDEX (eid);
```

Table employee contains the following single row.

employee		
eid	ename	jobperiod
1001	Xavier	2008-06-02,2008-06-24

Expand employee by WEEK\_BEGIN.

```
SELECT eid, ename, BEGIN(expd) AS tsp
FROM employee
EXPAND ON jobperiod expd BY ANCHOR WEEK_BEGIN;
```

In this example, each expanded row value starts on a Monday because the week starts on a Monday.

employee		
eid	ename	tsp
1001	Xavier	2008-06-09
1001	Xavier	2008-06-16
1001	Xavier	2008-06-23

## Example: EXPAND ON for a QUARTER\_BEGIN Anchor Period

This example expands employee using a QUARTER\_BEGIN anchor period.

```
SELECT eid, ename, BEGIN(expd) AS tsp
FROM employee
EXPAND ON jobdperiod expd BY ANCHOR PERIOD QUARTER_BEGIN;
```

eid ---	ename -----	tsp ---
1001	Xavier	2008-04-01

## Example: Join Before Expansion

Following are the table definitions for this example:

```
CREATE SET TABLE DR.t3, NO FALLBACK , NO BEFORE JOURNAL,
                    NO AFTER JOURNAL, CHECKSUM = DEFAULT (
  a  INTEGER,
  b  INTEGER,
  pd PERIOD(TIMESTAMP(6)))
PRIMARY INDEX (a);
```

```
CREATE SET TABLE DR.t4, NO FALLBACK, NO BEFORE JOURNAL,
                    NO AFTER JOURNAL, CHECKSUM = DEFAULT (
  x  INTEGER NOT NULL,
  y  INTEGER NOT NULL,
  pd PERIOD(DATE))
PRIMARY INDEX (x);
```

This example shows how Teradata Database joins the tables specified in an EXPAND ON clause when the specified period expression specifies a column from a table that is not specified in the FROM clause .

```
SELECT expd
      FROM t4
      EXPAND ON t3.pd AS expd;
```

An EXPLAIN shows a JOIN with t3.

## Example: Nullified EXPAND Operation

This example shows how when an expanded column is not specified in the select list of a query, but a DISTINCT operator *is* specified, the EXPAND operation is nullified.

```
CREATE SET TABLE df2.t1, NO FALLBACK, NO BEFORE JOURNAL,
                    NO AFTER JOURNAL, CHECKSUM = DEFAULT,
                    DEFAULT MERGEBLOCKRATIO (
  i  INTEGER,
  j  INTEGER,
  pd PERIOD(DATE) FORMAT 'yyyy-mm-dd')
PRIMARY INDEX (i);
```

The first SELECT statement in this example does not specify the DISTINCT operator.

```
EXPLAIN SELECT i,j
          FROM t1
          EXPAND ON pd AS expd BY INTERVAL '1' DAY;
```

An EXPLAIN shows an EXPAND ON t1.pd.

The following SELECT statement specifies the DISTINCT operator.

```
SELECT DISTINCT i,j
          FROM t1
          EXPAND ON pd AS expd BY INTERVAL '1' day;
```

An EXPLAIN shows a SORT and elimination of duplicate rows. The DISTINCT operator makes the EXPAND ON step unnecessary.

## Example: Null Expansion Period Producing a Null Expanded Value

This example shows how a null expansion period produces a null expanded value. In this example, the value for PERIOD(DATE) in column *pd* is null, so the expansion on *pd*, *expd*, is also null.

```
CREATE SET TABLE DF2.t4, NO FALLBACK, NO BEFORE JOURNAL,
                      NO AFTER JOURNAL, CHECKSUM = DEFAULT (
  x  INTEGER NOT NULL,
  y  INTEGER NOT NULL,
  pd PERIOD(DATE))
PRIMARY INDEX (x);
```

First show that column *pd* is null.

```
SELECT *
FROM t4;
*** Query completed. One row found. 3 columns returned.
*** Total elapsed time was 1 second.
x          y          pd
-----
10         30        ?
```

Then show that the expansion on *pd* aliased as *expd*, is also null.

```
SELECT x, expd
FROM t4
EXPAND ON pd AS expd;
*** Query completed. One row found. 2 columns returned.
*** Total elapsed time was 1 second.
```

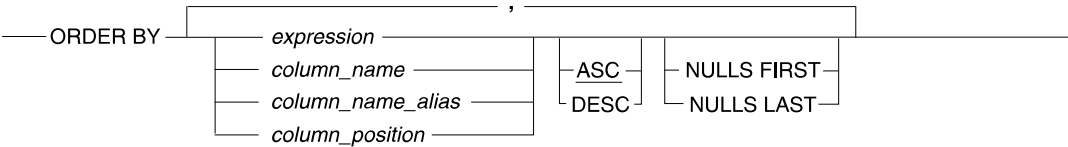
x	expd
-----	
10	?

# ORDER BY Clause

## Purpose

An expression in the SELECT list that specifies how result sets are sorted. If you do not use this clause, result rows are returned unsorted.

## Syntax



## Syntax Elements

### ORDER BY

- Order in which result rows are to be sorted.
- You can specify one or more sort columns on *expression*, *column\_name*, *column\_name\_alias*, or *column\_position*.
- The first sort column you specify determines the column used for the primary (first) sort. Any additional sort columns you specify determines the potential subsequent sorting based on the order in which you specify the columns.
- When multiple sort columns are specified, rows are sorted that have the same values of the previous sort column (or columns).
- You can specify a sort column to be sorted in either ascending or descending order.

### *expression*

- SQL expression on which to sort rows. The expression can specify the numeric position of the expression in the expression list with a name or a constant.
- If the sort field is a character string, the *expression* used in the ORDER BY phrase can include a type modifier to force the sort to be either CASESPECIFIC or NOT CASESPECIFIC.
- You can specify scalar subqueries and scalar UDFs, but you cannot reference a column that has a BLOB, CLOB, ARRAY, or VARRAY data type in the ORDER BY expression list.

### *column\_name*

- the name of a column on which to sort rows. An ordering column does not have to be specified as part of the select expression list.
- The maximum number of columns you can specify is 64.

You can specify a *column\_name\_alias* instead of *column\_name* (see the description of the *column\_name\_alias* element for details). The *column\_name\_alias* must not have the same name as a physical column in the table definition. If it does have the same name, you must specify *column\_position*, not *column\_name\_alias*. See [Example: Ordering on a Column Name Alias](#).

The column you specify cannot have a BLOB, CLOB, ARRAY, or VARRAY data type.

### ***column\_name\_alias***

a column name alias specified in the select expression list of the query for the column on which the result rows are to be sorted.

If you specify a *column\_name\_alias*, that alias cannot match the name of any column that is defined in the table definition for any table referenced in the FROM clause of the query, whether that column is specified in the select list or not. The system references the underlying physical column having the name rather than the column that you attempt to reference using that same name as its alias.

For an improper *column\_name\_alias*, the system returns an error.

The workaround for this is to specify the sort column by its *column\_position* value within the select list for the query. See [Column References and ORDER BY](#).

### ***column\_position***

the numeric position of a column or expression specified in the select expression list on which to sort.

The value you specify must be a positive constant integer literal with a value between 1 and the number of columns specified in the select list, inclusive. Note that Teradata Database treats macro and procedure parameters as expressions, not as the specification of a column position.

If the *column\_name\_alias* specified in the select list is the same as the name of the physical column in the underlying table specified in the FROM clause, then you cannot specify that *column\_name\_alias* as a sort value. You can specify a column position reference to order on the column or expression in the select list. However, you cannot use a *column\_name\_alias* that is the same as the name of some other column specified in the table definition. Substituting *column\_position* for *column\_name\_alias* when the query will not otherwise work is not a good practice, and is not recommended. However, it does work if you must resort to it for a given situation.

The column or expression referenced by its numeric position in the select-list must not have either a BLOB or CLOB data type.

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

## **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.

**NULLS FIRST**

NULL results are to be listed first.

**NULLS LAST**

NULL results are to be listed last.

**ANSI Compliance**

The ORDER BY clause is ANSI SQL:2011-compliant with extensions. The specification of an ORDER BY clause in the definition of an updatable cursor is ANSI SQL:2011-compliant.

The specification of an expression as an ordering argument in the ORDER BY clause is a Teradata extension to the ANSI SQL:2011 standard.

**Usage Notes****Column References and ORDER BY**

Each *column\_name* you specify in an ORDER BY clause must be the name of a column in a relation referenced in the SELECT expression list. The columns named do not have to match the columns in the SELECT expression list.

You can also order on a *column\_name\_alias*.

You can specify up to 64 column names.

You cannot specify BLOB or CLOB columns in the column name list because you cannot sort on LOB values.

The column position you specify in an ORDER BY clause must be an unsigned integer that refers to the sequential position of a column in the SELECT expression list. The column position cannot be passed as a parameter, because the parameter is interpreted as an expression and the data is sorted by a constant. You cannot specify BLOB or CLOB columns in the column position list because you cannot sort on LOB values.

For ORDER BY references to UDT columns, Teradata Database uses the defined ordering functionality for that UDT to order the result set.

You can specify a scalar subquery as a column expression in the ORDER BY clause of a query.

**WITH and ORDER BY**

You can specify an ORDER BY clause before or after a WITH clause.

When you specify both WITH and ORDER BY clauses in the same SELECT request, the clauses function together as follows:

- The WITH clause defines the major sort key.
- The ORDER BY clause defines the minor sort key, regardless of the structure of the query or the number of WITH clauses. See [Example: Combining WITH and ORDER BY Clauses](#).

## PARTITION BY, HASH BY, and ORDER BY

If you have multiple ON clauses, you cannot specify ORDER BY as the only option in an ON clause. You must combine ORDER BY with a PARTITION BY (ANY), HASH BY, or DIMENSION clause.

If you specify an ORDER BY or LOCAL ORDER BY clause with PARTITION BY or HASH BY input, then the following is required:

- All inputs must have the same number of ORDER BY columns.
- The data types of the columns must be the same type or matched using implicit cast.

## Sorting and Default Sort Order Row Length Errors

Before performing the sort operation that orders the rows to be returned, Teradata Database creates a sort key which is appended to the rows. If the length of the sort key exceeds the system row maximum length of 1 MB, the operation returns an error. Depending on the situation, the error message text is one of the following.

- A data row is too long.
- Maximum row length exceeded in *database\_object\_name*.

For explanations of these messages, see *Teradata Vantage™ - Database Messages*, B035-1096.

There are several possible reasons why a data row plus BYNET sort key can unexpectedly exceed the spool row size limit of 1 MB, even without including any updates.

- The value for the MaxDecimal DBS Control field for your system might have been changed to a larger value, which could increase the number of bytes stored for each value in a DECIMAL column to as much as 38 bytes where it might previously have been only 1, 2, 4, or 8 bytes. See *Teradata Vantage™ - Database Design*, B035-1094 and *Teradata Vantage™ Data Types and Literals*, B035-1143.
- Your site has upgraded from a system with a 32-bit aligned row architecture to a system with a 64-bit architecture. This upgrade expands the size of data, both on disk and in memory, for a number of data types. The expansion is due to byte alignment restrictions that require padding to ensure alignment on 8-byte boundaries. These restrictions do not exist for 32-bit architectures. Therefore, the same row can be within the spool row 1 MB size boundary on a 32-bit system, but exceed that limit on a 64-bit system.

This issue is not relevant if your system uses the packed64 row architecture rather than the aligned row format architecture. For more information, see *Teradata Vantage™ - Database Design*, B035-1094.

By default, nulls sort before all other data in a column. In other words, nulls sort low. For example, suppose you want to sort the following set of values: 5, 3, null, 1, 4, and 2.

The values sort as follows:

```

null
1
2
3
4
5

```

By default, the result values of a character expression are sorted in ascending order, using the collating sequence in effect for the session.

## Unexpected Sort Order When Querying DATE Data Type Columns

Using the ORDER BY clause to query columns with the DATE data type may yield results in which the order of the returned values seem to be out of order.

The factors that contribute to this type of unexpected result are:

- DATE time values are stored by the system on disk in UTC time, not local time.
- UTC time values are used to sort the column values, not the local time values.
- DATE time values are affected by the system time zone setting, which can be set using TimeZoneString or the DBSControl general field 16.
- DATE time values that cross the date boundary are adjusted based on the time zone offset for the system time zone (the time zone offset varies depending on the time zone).

### Example of Unexpected Sort Order

This example shows a result set of TIME data type column values that do not seem to be sorted in the order specified in the ORDER BY clause. The local time values inserted into the column are followed by the values returned by the query.

The time zone in this example is Pacific Time, which has a +8 hour time zone offset.

```

CREATE TABLE table_1
(column_1 INTEGER,
column_2 TIME);
INSERT INTO table_1 VALUES ( 0, TIME'00:00:00.000000');
INSERT INTO table_1 VALUES ( 1, TIME'01:00:00.000000');
INSERT INTO table_1 VALUES ( 2, TIME'02:00:00.000000');

```

```

INSERT INTO table_1 VALUES ( 3, TIME'03:00:00.000000');
INSERT INTO table_1 VALUES ( 4, TIME'04:00:00.000000');
INSERT INTO table_1 VALUES ( 5, TIME'05:00:00.000000');
INSERT INTO table_1 VALUES ( 6, TIME'06:00:00.000000');
INSERT INTO table_1 VALUES ( 7, TIME'07:00:00.000000');
INSERT INTO table_1 VALUES ( 8, TIME'08:00:00.000000');
INSERT INTO table_1 VALUES ( 9, TIME'09:00:00.000000');
INSERT INTO table_1 VALUES (10, TIME'10:00:00.000000');
INSERT INTO table_1 VALUES (11, TIME'11:00:00.000000');
INSERT INTO table_1 VALUES (12, TIME'12:00:00.000000');
INSERT INTO table_1 VALUES (13, TIME'13:00:00.000000');
INSERT INTO table_1 VALUES (14, TIME'14:00:00.000000');
INSERT INTO table_1 VALUES (15, TIME'15:00:00.000000');
INSERT INTO table_1 VALUES (16, TIME'16:00:00.000000');
INSERT INTO table_1 VALUES (17, TIME'17:00:00.000000');
INSERT INTO table_1 VALUES (18, TIME'18:00:00.000000');
INSERT INTO table_1 VALUES (19, TIME'19:00:00.000000');
INSERT INTO table_1 VALUES (20, TIME'20:00:00.000000');
INSERT INTO table_1 VALUES (21, TIME'21:00:00.000000');
INSERT INTO table_1 VALUES (22, TIME'22:00:00.000000');
INSERT INTO table_1 VALUES (23, TIME'23:00:00.000000');

```

This SELECT statement is used to query the time values in the TIME data type column (column\_2).

```

SELECT * FROM table_1
ORDER BY column_2;

```

column_1	column_2
16	16:00:00.000000
17	17:00:00.000000
18	18:00:00.000000
19	19:00:00.000000
20	20:00:00.000000
21	21:00:00.000000
22	22:00:00.000000
23	23:00:00.000000
0	00:00:00.000000
1	01:00:00.000000
2	02:00:00.000000
3	03:00:00.000000
4	04:00:00.000000
5	05:00:00.000000

6	06:00:00.000000
7	07:00:00.000000
8	08:00:00.000000
9	09:00:00.000000
10	10:00:00.000000
11	11:00:00.000000
12	12:00:00.000000
13	13:00:00.000000
14	14:00:00.000000
15	15:00:00.000000

Because the sort order is based on UTC time (which is not visible in the column), the time values do not agree with the resulting sort order. Rows 16 through 23 appear at the top of the list because these rows sort at the top of the list according to UTC time.

The UTC equivalents for the returned time values are:

Returned Values	UTC Equivalents (local time + 8 hours)
-----------------	--

column_1	column_2	
-----	-----	
16	16:00:00.000000	00:00:00.000000
17	17:00:00.000000	01:00:00.000000
18	18:00:00.000000	02:00:00.000000
19	19:00:00.000000	03:00:00.000000
20	20:00:00.000000	04:00:00.000000
21	21:00:00.000000	05:00:00.000000
22	22:00:00.000000	06:00:00.000000
23	23:00:00.000000	07:00:00.000000
0	00:00:00.000000	08:00:00.000000
1	01:00:00.000000	09:00:00.000000
2	02:00:00.000000	10:00:00.000000
3	03:00:00.000000	11:00:00.000000
4	04:00:00.000000	12:00:00.000000
5	05:00:00.000000	13:00:00.000000
6	06:00:00.000000	14:00:00.000000
7	07:00:00.000000	15:00:00.000000
8	08:00:00.000000	16:00:00.000000
9	09:00:00.000000	17:00:00.000000
10	10:00:00.000000	18:00:00.000000
11	11:00:00.000000	19:00:00.000000
12	12:00:00.000000	20:00:00.000000
13	13:00:00.000000	21:00:00.000000

14	14:00:00.000000	22:00:00.000000
15	15:00:00.000000	23:00:00.000000

## Workarounds for Unexpected Sort Order When Querying DATE

These workarounds can be used to get expected results:

- Use INTERVAL
- Use CAST to CHAR or VARCHAR
- Use TIMESTAMP instead of TIME

### Use INTERVAL

```
SELECT column_1, column_2, column_2 - interval '8' hour
FROM table_1
ORDER BY 3;
```

column_1	column_2	(column_2- 8)
-----	-----	-----
0	00:00:00.000000	16:00:00.000000
1	01:00:00.000000	17:00:00.000000
2	02:00:00.000000	18:00:00.000000
3	03:00:00.000000	19:00:00.000000
4	04:00:00.000000	20:00:00.000000
5	05:00:00.000000	21:00:00.000000
6	06:00:00.000000	22:00:00.000000
7	07:00:00.000000	23:00:00.000000
8	08:00:00.000000	00:00:00.000000
9	09:00:00.000000	01:00:00.000000
10	10:00:00.000000	02:00:00.000000
11	11:00:00.000000	03:00:00.000000
12	12:00:00.000000	04:00:00.000000
13	13:00:00.000000	05:00:00.000000
14	14:00:00.000000	06:00:00.000000
15	15:00:00.000000	07:00:00.000000
16	16:00:00.000000	08:00:00.000000
17	17:00:00.000000	09:00:00.000000
18	18:00:00.000000	10:00:00.000000
19	19:00:00.000000	11:00:00.000000
20	20:00:00.000000	12:00:00.000000
21	21:00:00.000000	13:00:00.000000
22	22:00:00.000000	14:00:00.000000
23	23:00:00.000000	15:00:00.000000

**Use CAST to CHAR or VARCHAR**

```
SELECT column_1, column_2, CAST(column_2 AS CHAR(15))
FROM table_1
ORDER BY 3;
```

column_1	column_2	column_2
-----	-----	-----
0	00:00:00.000000	00:00:00.000000
1	01:00:00.000000	01:00:00.000000
2	02:00:00.000000	02:00:00.000000
3	03:00:00.000000	03:00:00.000000
4	04:00:00.000000	04:00:00.000000
5	05:00:00.000000	05:00:00.000000
6	06:00:00.000000	06:00:00.000000
7	07:00:00.000000	07:00:00.000000
8	08:00:00.000000	08:00:00.000000
9	09:00:00.000000	09:00:00.000000
10	10:00:00.000000	10:00:00.000000
11	11:00:00.000000	11:00:00.000000
12	12:00:00.000000	12:00:00.000000
13	13:00:00.000000	13:00:00.000000
14	14:00:00.000000	14:00:00.000000
15	15:00:00.000000	15:00:00.000000
16	16:00:00.000000	16:00:00.000000
17	17:00:00.000000	17:00:00.000000
18	18:00:00.000000	18:00:00.000000
19	19:00:00.000000	19:00:00.000000
20	20:00:00.000000	20:00:00.000000
21	21:00:00.000000	21:00:00.000000
22	22:00:00.000000	22:00:00.000000
23	23:00:00.000000	23:00:00.000000

**Use TIMESTAMP instead of TIME**

```
CREATE TABLE table_2
(column_1 INTEGER,
column_2 TIMESTAMP);

INSERT INTO table_2 SELECT * FROM table_1;

SELECT * FROM table_2
ORDER BY column_2;
```

column_1	column_2
-----	-----
0	2011-11-07 00:00:00.000000
1	2011-11-07 01:00:00.000000
2	2011-11-07 02:00:00.000000
3	2011-11-07 03:00:00.000000
4	2011-11-07 04:00:00.000000
5	2011-11-07 05:00:00.000000
6	2011-11-07 06:00:00.000000
7	2011-11-07 07:00:00.000000
8	2011-11-07 08:00:00.000000
9	2011-11-07 09:00:00.000000
10	2011-11-07 10:00:00.000000
11	2011-11-07 11:00:00.000000
12	2011-11-07 12:00:00.000000
13	2011-11-07 13:00:00.000000
14	2011-11-07 14:00:00.000000
15	2011-11-07 15:00:00.000000
16	2011-11-07 16:00:00.000000
17	2011-11-07 17:00:00.000000
18	2011-11-07 18:00:00.000000
19	2011-11-07 19:00:00.000000
20	2011-11-07 20:00:00.000000
21	2011-11-07 21:00:00.000000
22	2011-11-07 22:00:00.000000
23	2011-11-07 23:00:00.000000

## Specifying Collation

Different languages use different collation sequences, and you can alter your system collation to ensure that the ORDER BY clauses you specify in your queries sort correctly.

You can specify the default collation for a user using the CREATE USER or MODIFY USER statement, or you can specify collation for the duration of a session using the SET SESSION COLLATION statement. Otherwise, collation for a session is determined by the logon client system. The direction of the sort can be controlled by including the DESC (descending) or ASC (ascending) sort option in the SQL request.

Teradata Database supports ASCII, EBCDIC, and MULTINATIONAL collating sequences. If MULTINATIONAL is in effect, your collation is one of the European (diacritical) or Kanji sort sequences described in [International Sort Orders](#).

The following topics explain the results of ORDER BY as affected by whether character string expressions have the CASESPECIFIC or NOTCASESPECIFIC attribute.



## Japanese Character Sort Order Considerations

If character strings are to be sorted NOT CASESPECIFIC, only lowercase simple letters, a through z in the Latin alphabet, are converted to uppercase before a comparison or sorting operation is done. NOT CASESPECIFIC is the default in Teradata session mode. See “SET SESSION COLLATION” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

Case differences are ignored in NOT CASESPECIFIC collations. The case of the characters is not critical for NOT CASESPECIFIC collation.

Any non-Latin single-byte character, any multibyte character, and any byte indicating a transition between single-byte characters and multibyte characters is excluded from this function.

If the character strings are to be sorted CASESPECIFIC, which is the default in ANSI session mode, then the case of the characters is critical for collation.

For Kanji1 character data in CASESPECIFIC mode, the letters in any alphabet that uses casing, such as Latin, Greek, or Cyrillic, are considered to be matched only if the letters are identical with the same case.

The system does not consider FULLWIDTH and HALFWIDTH characters to be matched whether in CASESPECIFIC or NOT CASESPECIFIC mode.

Teradata Database uses one of four server character sets to support Japanese characters:

- Unicode
- KanjiSJIS
- Kanji1
- Graphic

### 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.

For Japanese character sites, you can set your session collation as follows:

- For character data stored on the Teradata platform as KanjiSJIS or Unicode, the best way to order the session character set is to use the CHARSET\_COLL collation.  
For character data stored on the Teradata platform as either KanjiSJIS or Unicode, the CHARSET\_COLL collation provides the collation that is closest to sorting on the client.
- The JIS\_COLL collation also provides an adequate collation, and also provides the same collation regardless of the session character set.
- The CHARSET\_COLL and JIS\_COLL collation sequences are not designed to support Kanji1 character data.

For Kanji1 character data, the ASCII collation provides a collation similar to that of the client, assuming the session character set is KanjiSJIS\_0S, KanjiEUC\_0U, or something very similar. However, ASCII collation does not sort like the client if the data is stored on the Teradata platform using the ShiftJIS or Unicode server character sets.

- Users under the KATAKANAEBDIC, KANJIEBCDIC5026\_0I, or KANJIEBCDIC5035\_0I character sets who want to store character data on the Teradata platform as Kanji1, and who want to collate in the session character set should install either KATAKANAEBDIC, KANJIEBCDIC5026\_0I, or KANJIEBCDIC5035\_0I, respectively, at start-up time, and use MULTINATIONAL collation.

Each character set requires a different definition for its MULTINATIONAL collation to collate properly.

Teradata Database handles character data collation for the different Japanese server character sets as follows:

- Under the KanjiEUC character set, the ss3 0x8F is converted to 0xFF. This means that a user-defined KanjiEUC codeset 3 are not ordered properly with respect to other KanjiEUC code sets. The order of other KanjiEUC code sets is proper, that is, ordering is the same as the binary ordering on the client system.

ASCII collation collates Kanji1 data in binary order, but handles case matching of single-byte or HALFWIDTH Latin characters. See *International Character Set Support*. This matches collation on the client for KanjiSJIS\_0S sessions, is close for KanjiEUC\_0U, and is only reasonably close for double-byte data for KanjiEBDIC sessions. KanjiEUC\_0U puts code set 3 after code set 1 rather than before code set 1 as KanjiEBDIC does.

- For Kanji1 data, characters identified as multibyte characters remain in the client encoding and are collated based on their binary values. This explains why ASCII collation works for double-byte characters in KanjiEBDIC sessions.

Multibyte Kanji1 characters do not remain in the client encoding for KanjiEUC\_0U sessions.

For details, see *Teradata Vantage™ NewSQL Engine International Character Set Support*, B035-1125 and *Teradata Vantage™ Data Types and Literals*, B035-1143.

## International Sort Orders

MULTINATIONAL collation is enabled at the user level via the COLLATION option of the CREATE USER or MODIFY USER statement. If a collation is not specified, the default is HOST. The standard sort ordering of the client system is EBCDIC for IBM clients and ASCII for all others.

You can override the default at the session level by issuing the SET SESSION COLLATION statement.

When MULTINATIONAL collation is in effect, the default collation sequence is determined by the collation setting installed at system start-up. Also see *Teradata Vantage™ NewSQL Engine International Character Set Support*, B035-1125.

Each international sort sequence is defined by program constants and no check is made to ensure that collation is compatible with the character set of the current session. The choice of sequence is controlled by your database administrator. The programmed character sequences cannot be changed.

## European Sort Order

The sort order uses a two-level comparison that involves the following rules.

- All lowercase letters are first mapped to their uppercase equivalents unless CASESPECIFIC is specified in the ORDER BY clause or was defined for the column being accessed.

In ANSI session mode, the default is CASESPECIFIC and you must explicitly specify NOT CASESPECIFIC to change this.

- All diacritical forms of the same letter are given the value of the base letter; that is, Ä is given the value of A, Ö is given the value of O, and so forth.
- If two strings produce the same value, the characters are further ordered according to their sort position in their diacritical equivalence class. See the table below.
- Unless the query specifies the DESC sort option, collation is in ascending order.

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

When these rules are applied, the words “abbey,” “Active,” and “adage” are returned in this order,

abbey	Active	adage
-------	--------	-------

and the names Muller, Handl, Böckh, Mueller, Händl, Bohr, Bock, and Müller are ordered as:

Bock	Böckh	Bohr	Handl
Händl	Mueller	Muller	Müller

Equivalence classes and the ordering of diacritical characters in each class are shown in this table. The listed classes are those with characters that have diacritical forms.

European Sort Order					
A	C	E	I	N	O
a A à À á Á â Â ã Ã ä Ä	c C ç Ç	e E è È é É ê Ê ë Ë	i I í Ì ì Í î Î ï Ï	n N ñ Ñ	o O ò Ò ó Ó ô Ô õ Ö ö Ö o O
S	U	Y	AE	O slash	A ring
s S ß	u U ù Ù ú Ú û Û (U tilde)	y Y ÿ Ÿ	æ Æ	ø Ø	å Å

European Sort Order					
A	C	E	I	N	O
	ü Ü				

## Examples

### Example: Ordering on a Column Name

The following example produces a list of employees, sorted by years of work experience. Note that the ORDER BY sort column, *yrs\_exp*, is also specified in the select list of the statement.

```
SELECT name, jobtitle, yrs_exp
FROM employee
ORDER BY yrsexp;
```

### Example: Ordering on a Column Name Not Specified in the Select List

The following example produces a list of employees, sorted by *dept\_no* and *salary*. Note that one of the ORDER BY sort columns, *salary*, is *not* specified in the select list.

```
SELECT name, dept_no
FROM employee
ORDER BY dept_no, salary;
```

### Example: Ordering on Column Position

The following example substitutes the integer 3 for *yrs\_exp* in the ORDER BY clause. The 3 refers to the left-to-right sequential numeric position of *yrs\_exp* in the select expression list.

```
SELECT name, jobtitle, yrs_exp
FROM employee
ORDER BY 3;
```

### Example: Ordering on a Column Name Alias

Define a column named *j* in table *t1*.

The following statement returns an error because *j* is defined to be an alias for the expression `SUM(t1.j)`. However, when it is used in the ORDER BY clause, the system resolves it to the column *j* and not the expression aliased by *j*:

```
SELECT t1.i, SUM(t1.j) AS j, t1.k
FROM t1
GROUP BY 1,3
ORDER BY j;
```

The following statement works because the column name alias *jj* is not the same as the column name *j*, and there is no column named *jj* in the table definition.

```
SELECT t1.i, SUM(t1.j) AS jj, t1.k
FROM t1
GROUP BY 1,3
ORDER BY jj;
```

### Example: Ordering on an Expression

The following example returns a list of each department number and its total population, in the order of lowest population first. The ORDER BY clause specifies a column expression, COUNT(name), to collate the response set.

```
SELECT COUNT(name), dept_no
FROM employee
GROUP BY dept_no,
ORDER BY COUNT(name);
```

### Example: Ordering on Column Name or Column Position Using an Ascending or Descending Sort Sequence

Each of the following statements can be used to list employees by department number, with the highest paid employee listed first and the lowest paid last:

```
SELECT name, dept_no, salary
FROM employee
ORDER BY 2 ASC, 3 DESC;
SELECT name, dept_no, salary
FROM employee
ORDER BY dept_no, salary DESC;
```

### Example: Effects of Default and User-Specified Case Sensitivity on the Ordering of a Result Set

The following statements create and populate table *t*:

```
CREATE TABLE t (  
  a CHAR(4) NOT CASESPECIFIC,  
  b BYTEINT)  
PRIMARY INDEX (a,b);  
INSERT INTO t VALUES ('AAAA', 1);  
INSERT INTO t VALUES ('aaaa', 2);  
INSERT INTO t VALUES ('BBBB', 3);  
INSERT INTO t VALUES ('bbbb', 4);
```

If the default handling of case is allowed, the following statement produces the following results table.

```
SELECT *  
FROM t  
ORDER BY a;  
a      b  
----  ---  
AAAA   1  
aaaa   2  
BBBB   3  
bbbb   4
```

On the other hand, when you specify CASESPECIFIC for the query, the results are one of the results tables immediately following the example SELECT statements, depending on the collation sequence in effect for the current session.

```
SELECT *  
FROM t  
ORDER BY CAST(a AS CASESPECIFIC);
```

or

```
SELECT CAST(a AS CASESPECIFIC), b  
FROM t  
ORDER BY 1;
```

EBCDIC		ASCII		MULTINATIONAL	
A	B	A	B	A	B
----	----	----	----	----	----
aaaa	2	AAAA	1	aaaa	2
bbbb	4	BBBB	3	AAAA	1
AAAA	1	aaaa	2	bbbb	4
BBBB	3	bbbb	4	BBBB	3

## Example: Ordering By a Column Alias Name

You cannot explicitly order by a column alias name. You must specify the column position of the column aliased by the name in the select list.

Suppose you have the following base table definition:

```
CREATE TABLE t1
(i INTEGER,
 j INTEGER,
 k INTEGER);
```

The following query against this table fails because it specifies ordering on the column alias name *j*, which the system interprets as column *j* in table *t1* rather than as the expression `SUM(t1.j)`:

```
SELECT t1.i, SUM(t1.j) AS j, t1.k
FROM t1
GROUP BY 1,3
ORDER BY j;
*** Failure 3504 Selected non-aggregate values must be part of the associated
group.
```

The following rewrite of the query successfully performs the desired ordering:

```
SELECT t1.i, SUM(t1.j) AS j, t1.k
FROM t1
GROUP BY 1,3
ORDER BY 2;
```

## Example: Ordering on a PERIOD Value Expression

The following example shows how you can specify a PERIOD value expression in an ORDER BY clause, where *period\_of\_stay* is the PERIOD value expression.

```
SELECT *
FROM employee
ORDER BY period_of_stay;
```

## Related Topics

For more information about the various ways you can specify character collation in Teradata Database, see:

- *Teradata Vantage™ NewSQL Engine International Character Set Support*, B035-1125 for information about using various international character sets with Teradata Database.
- *Teradata Vantage™ - Database Administration*, B035-1093 for information about viewing and changing international character sets and their defaults.
- *Teradata Vantage™ Data Types and Literals*, B035-1143 for information about the CASESPECIFIC and NOT CASESPECIFIC column data type attributes.
- “CREATE USER” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144 for information about how to create a default collation sequence and a default server character set for a user.
- “MODIFY USER” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144 for information about how to alter the default collation sequence or default server character set for a user.
- “SET SESSION COLLATION” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144 for information about how to establish a default collation sequence for a session.

## WITH Clause

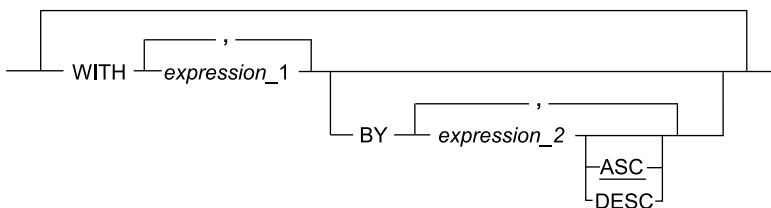
### Purpose

Specifies summary lines and breaks (also known as grouping conditions) that determine how selected results are returned.

The typical use of a WITH clause is with subtotals.

The WITH clause has a different function than the WITH statement modifier. See [WITH Modifier](#).

### Syntax



### Syntax Elements

#### WITH *expression\_1*

An introduction to the condition to be fulfilled by the SELECT statement. Specifies a summary line, such as a total, for the values in a column of the select result. *expression\_1* can contain one or more aggregate expressions that are applied to column values.

You cannot include LOB columns in the WITH *expression\_1* list.

You cannot include a WITH clause with NORMALIZE.



**BY *expression\_2***

One or more result expressions for which *expression\_1* is provided. BY is valid only when used with WITH.

*expression\_2* can refer to an expression in the select expression list either by name or by means of a constant that specifies the numeric position of the expression in the expression list. It can also refer to a scalar subquery.

You cannot include LOB columns in the BY *expression\_2* list.

**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.

**ANSI Compliance**

The WITH clause is a Teradata extension to the ANSI SQL:2011 standard.

**UDT Columns and the WITH Clause**

You cannot specify UDT columns in a WITH clause.

**Scalar Subqueries in the BY Specification of a WITH ... BY clause**

You can specify a scalar subquery as a column expression or parameterized value in the BY specification of a WITH ... BY clause in a query.

Note that you cannot specify a scalar subquery as the argument of a WITH clause because only aggregate expressions are valid as WITH clause arguments. You can, however, specify a scalar subquery as an argument of an aggregate expression in a WITH clause.

**LOB columns and the WITH Clause**

You cannot specify LOB columns in a WITH clause.

**TOP Operator and WITH Statement Modifier**

A SELECT statement that specifies the TOP *n* operator cannot also specify a WITH statement modifier.

**Multiple WITH Clauses in a SELECT Statement**

You can specify more than one WITH clause in a SELECT statement to specify different kinds of summaries. Each succeeding WITH clause refers to an ever broader grouping of rows as follows.

- The BY phrase in the first WITH clause defines the least important sort key.
- The BY phrase in the next WITH clause defines the next-to-least important sort key.
- The final WITH clause defines the major sort key.

See [Example: Specifying Multiple WITH Clauses in a Single SELECT Statement](#).

### ORDER BY Clause and WITH Clause

You can specify an ORDER BY clause before or after any WITH clause. Also see [ORDER BY Clause](#).

WITH and ORDER BY clauses in the same SELECT statement function together as follows:

- The WITH clause defines the major sort key.
- The ORDER BY clause defines the minor sort key.

This is true regardless of the structure of the query or the number of WITH clauses.

See [Example: Combining WITH and ORDER BY Clauses](#).

### WITH and GROUP BY clauses in a SELECT Statement

Do not combine WITH and GROUP BY clauses in a SELECT statement.

Specifying WITH and GROUP BY clauses in the same SELECT statement can produce unintended results.

See [Example: Combining WITH and ORDER BY Clauses](#).

Also see [GROUP BY Clause](#).

### Expressions and the WITH Clause

The *expression\_2* you specify determines where summary lines are generated. For example, BY dept\_no specifies a summary for each value in the dept\_no column; a summary line is generated following a listing of the values for each department number.

If you do not specify a BY phrase, the summary line applies to the entire result as specified by the SELECT expression list.

Like the ORDER BY clause, the values of any expression specified by *expression\_2* can be sorted in either ascending or descending order. For example:

```
WITH SUM(salary) BY divno ASC, dept_no DESC
```

Likewise, *expression\_2* can specify a constant that references an expression by its position in the SELECT expression list. For example:

```
WITH SUM(salary) BY 2 ASC, 3 DESC
```

However, an expression that is specified in *expression\_1* or *expression\_2* need not be specified in the SELECT expression list.

You can specify the following expressions in a WITH clause.

- Expressions operated on by aggregate operators, for example, SUM, AVERAGE, COUNT, MIN, or MAX.

An aggregate operator must be specified directly before each column to which the operator applies, for example, WITH SUM(salary) or MAX(yrsexp).

- Expressions associated with the column values of an expression contained in the BY phrase, for example, WITH dept\_no, SUM(salary) BY dept\_no.

You cannot specify expressions that include LOB or UDT columns in a WITH clause.

### TITLE Phrase and the WITH Clause

You can use a TITLE phrase to specify a title for any valid expression contained in *expression\_1* and the SELECT expression list. The TITLE phrase must be enclosed by parentheses and follow the entire expression to which it applies.

Title is relevant only for FieldMode output for report generation and normally done only via BTEQ.

This clause lists the title Subtotal at each summary row:

```
WITH SUM(salary)(TITLE 'Subtotal')
```

This clause specifies a blank title:

```
WITH SUM(salary)(TITLE ' ')
```

See [Example: Specifying a Detail Line for Each Employee and a Subtotal Line for Each Department](#).

### Example: Specifying a Detail Line for Each Employee and a Subtotal Line for Each Department

You can use the following statement to generate a departmental salary report that contains a detail line for each employee and a subtotal line for each department:

```
SELECT name, dept_no, salary
FROM employee
WITH SUM(salary) BY dept_no;
```

The result returned is as follows:

name	dept_no	salary
----	-----	-----
Peterson J	100	25,000.00
Moffit H	100	35,000.00
Jones M	100	50,000.00
Chin M	100	38,000.00
Greene W	100	32,500.00
		-----
	Sum(Salary)	180,500.00
Leidner P	300	34,000.00
Phan A	300	55,000.00
Russell S	300	65,000.00

```
-----
Sum(Salary) 154,000.00
```

### Example: Specifying Multiple WITH Clauses in a Single SELECT Statement

The following statement generates a report of employee salaries ordered by department, with a summary line of total salaries for each department and a final summary line of total salaries for the entire organization:

```
SELECT name, dep_tno, salary
FROM employee
WITH SUM(salary) BY dept_no
WITH SUM(salary);
```

The result returned is as follows:

```
name      dept_no  salary
-----
Peterson J    100    25,000.00
Moffit H     100    35,000.00
Jones M      100    50,000.00
Chin M       100    38,000.00
Greene W     100    32,000.00
           Sum(Salary) 180,000.00
Leidner P    300    34,000.00
Phan A       300    55,000.00
Russell S    300    65,000.00
           Sum(Salary) 154,000.00
Smith T      500    42,000.00
Clements D   700    38,000.00
           Sum(Salary) 113,000.00
           Sum(Salary) 851,000.00
```

### Example: Combining WITH and ORDER BY Clauses

Both of the following statements use an ORDER BY clause to sort employee names in ascending order in each dept\_no group.

```
SELECT name, dept_no, salary
FROM employee
ORDER BY name
WITH SUM(salary) BY dept_no
WITH SUM(salary);
```

```

SELECT name, dept_no, salary
FROM employee
WITH SUM(salary) BY dept_no
WITH SUM(salary)
ORDER BY name;

```

The result returned is as follows.

name	dept_no	salary
-----	-----	-----
Chin M	100	38,000.00
Greene W	100	32,500.00
Jones M	100	50,000.00
Moffit H	100	35,000.00
Peterson J	100	25,000.00
		-----
	Sum(Salary)	180,500.00
.	.	.
.	.	.
.	.	.
Brangle B	700	30,000.00
Clements D	700	38,000.00
Smith T	700	45,000.00
		-----
	Sum(Salary)	113,000.00
		-----
	Sum(Salary)	851,100.00

If any sort key column contains character data that was entered in mixed case, the results produced from WITH...BY or ORDER BY can be unexpected, depending on whether the CASESPECIFIC option was defined on the column and the collation in effect for the session. See [ORDER BY Clause](#) and *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

You must code your statements carefully to avoid returning unintended result sets from combining GROUP BY and WITH clauses. The next example illustrates a possible problem with careless coding of combined GROUP BY and WITH clauses.

The following statement is intended to generate a report of salary totals by dept\_no with a grand total of employee salaries:

```

SELECT dept_no, SUM(salary)
FROM employee
GROUP BY dept_no
WITH SUM(salary);

```

The result returned is as follows:

dept_no	Sum(Salary)
-----	-----
100	180,500.00
300	143,000.00
500	268,000.00
600	146,600.00
700	113,000.00
-----	-----
Sum(Sa	851,100.00

As would be expected, the WITH clause produces a summary line of total salaries for all departments. The summary title is truncated because of the narrow width of the dept\_no column.

If a WITH clause contains an unnecessary BY phrase, then a redundant summary line is generated following each department salary total as seen in the report following this example query:

```
SELECT dept_no, SUM(salary)
FROM employee
GROUP BY dept_no
WITH SUM(salary) BY dept_no;
```

dept_no	Sum(Salary)
-----	-----
100	180,500.00
	-----
Sum(Sa	180,500.00
	.
	.
	.
700	113,000.00
	-----
Sum(Sa	113,000.00

### Example: Scalar Subquery in the WITH ... BY Clause of a SELECT Statement

The following example specifies a scalar subquery (SELECT prod\_name...) in the BY clause.

```
SELECT SUM(amount)
FROM sales_table AS s
WITH AVG(amount) BY (SELECT prod_name
                      FROM prod_table AS p
                      WHERE p.prod_no = s.prod_no);
```

# Set Operators

## Overview

The SQL set operators manipulate the results sets of two or more queries by combining the results of each individual query into a single results set.

## Teradata SQL Set Operators

Teradata SQL supports the following set operators:

Set Operator	Function
INTERSECT	Returns result rows that appear in all answer sets generated by the individual SELECT statements.
MINUS /EXCEPT	Result is those rows returned by the first SELECT except for those also selected by the second SELECT. MINUS is the same as EXCEPT.
UNION	Combines the results of two or more SELECT statements.

Set operators appear in query expressions. A query expression is a set of queries combined by the set operators INTERSECT, MINUS/EXCEPT, and UNION.

## Syntax for *query\_term*

```

┌ SELECT — statement ─┐
└ ( query_expression ) ─┘

```

### SELECT *statement*

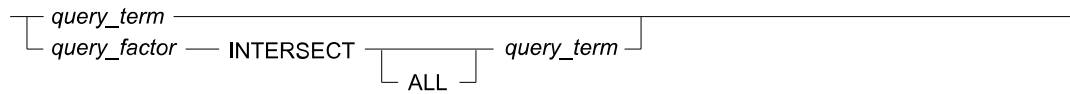
A SELECT statement.

For details, see [SELECT](#).

### *query\_expression*

An expression that might or might not include set operators, other expressions, and an ORDER BY clause.

## Syntax for *query\_factor*



## INTERSECT

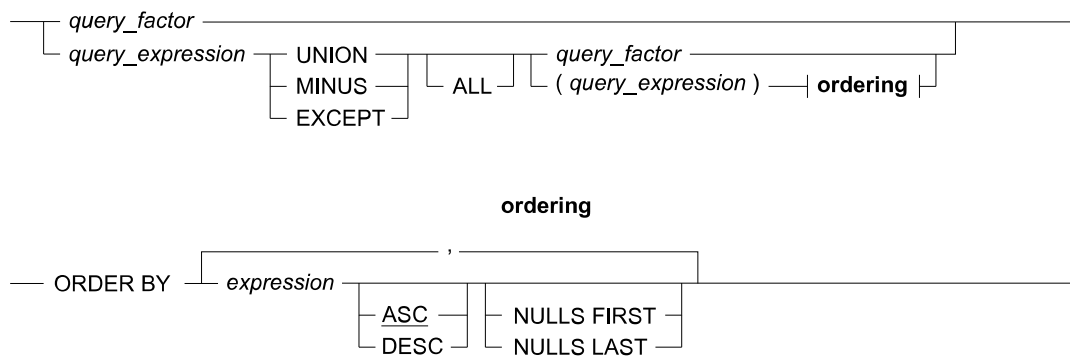
A set operator returning the result rows appearing in all answer sets.

## ALL

an optional keyword, allowing duplicate rows to be returned.

## Syntax for *query\_expression*

### Syntax



## UNION, MINUS, EXCEPT

Set operators specifying how the two or more queries or subqueries are to combine and determine what result rows are required to be returned.

## ALL

Allows duplicate rows to be returned.



## ORDER BY

The ORDER BY clause to order the result rows returned.

### *expression*

An expression used in the ORDER BY clause to determine the sort order of returned rows in the result.

## ASC

That the 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

That the 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.

## NULLS FIRST

NULL results are to be listed first.

## NULLS LAST

NULL results are to be listed last.

## ANSI Compliance

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

## Related Topics

For more information, see:

- For an example, see [Retaining Duplicate Rows Using the ALL Option](#).

## Rules for Set Operators

### Duplicate Rows

By default, duplicate rows are not returned.

To permit duplicate rows to be returned, specify the ALL option.

### Operations That Support Set Operators

You can use set operators within the following operations:

- Simple queries
- Derived tables

---

**Note:**

You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators.

---

- Subqueries
- INSERT ... SELECT clauses
- View definitions

SELECT statements connected by set operators can include all of the normal clause options for SELECT except the WITH clause.

### SELECT AND CONSUME Statement

Set operations do not operate on SELECT AND CONSUME statements.

### Support for ORDER BY Clause

A query expression can include only one ORDER BY specification, at the end.

### Restrictions on the Data Types Involved in Set Operations

The following restrictions apply to CLOB, BLOB, and UDT types involved in set operations.

Data Type	Restrictions
BLOB	You cannot use set operators with CLOB or BLOB types, except in the case of the UNION ALL set operator.
CLOB	
UDT	<ul style="list-style-type: none"> <li>• Multiple UDTs involved in set operations must be identical types because Teradata Database does not perform implicit type conversion on UDTs involved in set operations.</li> </ul>

Data Type	Restrictions
	<p>A workaround for this restriction is to use CREATE CAST to define casts that cast between the UDTs and then explicitly invoke the CAST function within the set operation.</p> <ul style="list-style-type: none"> <li>UDTs involved in set operations must have ordering definitions.</li> </ul> <p>Teradata Database generates ordering functionality for distinct UDTs where the source types are not LOBs. To create an ordering definition for structured UDTs or distinct UDTs where the source types are LOBs, or to replace system-generated ordering functionality, use CREATE ORDERING.</p> <p>For more information on CREATE CAST and CREATE ORDERING, see <i>Teradata Vantage™ SQL Data Definition Language Syntax and Examples</i>, B035-1144.</p>

## Related Topics

For more information, see:

- For an example, see [Retaining Duplicate Rows Using the ALL Option](#).

## Precedence of Set Operators

The precedence for processing set operators is as follows:

1. INTERSECT
2. UNION and MINUS/EXCEPT

The set operators evaluate from left to right if no parentheses explicitly specify another order.

## Example: Select Statement

Consider the following query:

```
SELECT statement_1
UNION
SELECT statement_2
EXCEPT
SELECT statement_3
INTERSECT
SELECT statement_4;
```

The operations are performed in the following order:

1. Intersect the results of statement\_3 and statement\_4.
2. Union the results of statement\_1 and statement\_2.
3. Subtract the intersected rows from the union.

## Using Parentheses to Customize Precedence

To override precedence, use parentheses. Operations in parentheses are performed first.

For example, consider the following form:

```
( ( SELECT statement_1
    UNION
    SELECT statement_2 )
  EXCEPT
  ( SELECT statement_3
    UNION
    SELECT statement_4 )
)
EXCEPT
SELECT statement_5
INTERSECT
SELECT statement_6;
```

The following list explains the precedence of operators for this example.

1. UNION SELECT statement \_1 and SELECT statement \_2.
2. UNION SELECT statement \_3 and SELECT statement \_4.
3. Subtract the result of the second UNION from the result of the first UNION.
4. INTERSECT SELECT statement \_5 and SELECT statement \_6.
5. Subtract the INTERSECT result from the remainder of the UNION operations.

## Related Topics

For more information, see:

- For examples that show how the length of the character type in the first SELECT statement affects the result set, see [Attributes of a Set Result](#).
- For examples that show how the numeric data type in the first SELECT statement affects the result set, see [Example: Effect of the Order of SELECT Statements on Data Type](#).
- For details, see [Syntax for query\\_expression](#).

## Retaining Duplicate Rows Using the ALL Option

Unless you specify the ALL option, duplicate rows are eliminated from the final result. The ALL option retains duplicate rows for the result set to which it is applied.

## Example

The following query returns duplicate rows for each result set, including the final:

```

SELECT statement_1
UNION ALL
SELECT statement_2
MINUS ALL
SELECT statement_3
INTERSECT ALL
SELECT statement_4

```

## Attributes of a Set Result

The data type, title, and format clauses contained in the first SELECT statement determine the data type, title, and format information that appear in the final result.

Attributes for all other SELECT statements in the query are ignored.

## Examples

### Example

```

SELECT level, param, 'GMKSA' (TITLE 'OWNER')
FROM gmksa
WHERE cycle = '03'
UNION
SELECT level, param, 'GMKSA CONTROL'
FROM gmksa_control
WHERE cycle = '03'
ORDER BY 1, 2;

```

The query returns the following results set:

```

***QUERY COMPLETED. 5 ROWS FOUND. 3 COLUMNS RETURNED.
LEVEL  PARAM  OWNER
-----  -----  -----
00      A      GMKSA
00      T      GMKSA
85      X      GMKSA
SF      A      GMKSA
SF      T      GMKSA

```

The first SELECT specifies GMKSA, which is CHAR(5)—that data type is then forced on the second SELECT. As a result, GMKSA\_CONTROL entries are dropped because the first five characters are the same.

Because this query does not specify the ALL option, duplicate rows are dropped.

## Example

In the next query, the SELECT order is reversed:

```
SELECT level, param 'GMKSA CONTROL' (TITLE 'OWNER')
FROM gmksa_control
WHERE cycle = '03'
UNION
SELECT level, param, 'GMKSA'
FROM gmksa
WHERE cycle = '03'
ORDER BY 1, 2;
```

This query returns the following answer set:

```
***QUERY COMPLETED.10 ROWS FOUND. 3 COLUMNS RETURNED.
LEVEL    PARAM    OWNER
-----
00       A       GMKSA
00       A       GMKSA CONTROL
00       T       GMKSA
00       T       GMKSA CONTROL
85       X       GMKSA
85       X       GMKSA CONTROL
SF       A       GMKSA
SF       A       GMKSA CONTROL
SF       T       GMKSA
SF       T       GMKSA CONTROL
```

In this case, because the first SELECT specified 'GMKSA CONTROL', the rows were not duplicates and were included in the answer set.

## Example

This example demonstrates how a poorly formed query can cause truncation of the results.

```
SELECT level, param, 'GMKSA      ' (TITLE 'OWNER')
FROM gmksa
WHERE cycle = '03'
UNION
SELECT level, param, 'GMKSA CONTROL'
FROM gmksa_control
```

```
WHERE cycle = '03'
ORDER BY 1, 2;
```

This query returns the following answer set:

```
***QUERY COMPLETED.10 ROWS FOUND. 3 COLUMNS RETURNED.
LEVEL    PARAM    OWNER
-----
00        A      GMKSA
00        A      GMKSA CONTRO
00        T      GMKSA
00        T      GMKSA CONTRO
85        X      GMKSA
85        X      GMKSA CONTRO
SF        A      GMKSA
SF        A      GMKSA CONTRO
SF        T      GMKSA
SF        T      GMKSA CONTRO
```

This query returned the expected rows; note, however, that because of the way the name was specified in the first SELECT, there was some truncation.

## Example of How the Character Set is Determined for the Query

The character set of the expression in the first SELECT statement determines the character set of the entire query. In the following example, the character set of *<char\_expression\_1>* is used as the character set for the entire query:

```
SELECT <char_expression_1> FROM <table_1>
UNION
SELECT <char_expression_2> FROM <table_2>;
```

If *<char\_expression\_2>* contains characters not included in the character set of *<char\_expression\_1>*, an error can result.

For example, if *<char\_expression\_1>* is CHARACTER SET LATIN, but *<char\_expression\_2>* is CHARACTER SET UNICODE, attempting to translate *<char\_expression\_2>* from UNICODE to LATIN results in an error if multi-byte characters are present in *<char\_expression\_2>*.

## Set Operators With Derived Tables

Derived tables support set operators, as demonstrated in the following example:

## Example

```
SELECT x1
FROM table_1,
(SELECT x2
FROM table_2
UNION
SELECT x3
FROM table_3
) derived_table;
SELECT x1,y1
FROM table_1,
(SELECT *
FROM table_2) derived_table(column_1, column_2)
WHERE column_2 = 1 ;
```

## Restrictions

You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators. The following example returns an error.

## Example

The following table function "add2int" takes two integers as input and returns the two integers and their summation.

```
CREATE TABLE t1 (a1 INTEGER, b1 INTEGER);
CREATE TABLE t2 (a2 INTEGER, b2 INTEGER);
REPLACE FUNCTION add2int
  (a INTEGER,
   b INTEGER)
RETURNS TABLE
  (addend1 INTEGER,
   addend2 INTEGER,
   mysum INTEGER)
SPECIFIC add2int
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
NOT DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'CS!add3int!add2int.c';
/* Query Q1 */
```



```

WITH dt(a1, b1) AS
( SELECT a1, b1
  FROM t1
  UNION ALL
  SELECT a2, b2
  FROM t2
)
SELECT *
FROM TABLE (add2int(dt.a1, dt.b1)
HASH BY b1
LOCAL ORDER BY b1) tf;

```

## Set Operators in Subqueries

Set operators are permitted in subqueries. The following examples demonstrate their correct use.

### Examples

#### Example

```

SELECT x1
FROM table_1
WHERE (x1,y1) IN
(SELECT * FROM table_2
UNION
SELECT * FROM table_3);

```

#### Example

```

SELECT *
FROM table_1
WHERE table_1.x1 IN
(SELECT x2
FROM table_2
UNION
(SELECT x3
FROM table_3
UNION
SELECT x4
FROM table_4));

```

## Example

```
SELECT *  
FROM table_1  
WHERE x1 IN  
(SELECT SUM(x2)  
FROM table_2  
UNION  
SELECT x3  
FROM table_3);
```

## Example

```
SELECT *  
FROM table_1  
WHERE x1 IN  
(SELECT MAX(x2)  
FROM table_2  
UNION  
SELECT MIN(x3)  
FROM table_3);
```

## Example

```
SELECT *  
FROM table_1  
WHERE X1 IN  
(SELECT x2 FROM table_2  
UNION  
SELECT x3 FROM table_3  
UNION  
SELECT x4 FROM table_4);
```

## Example

```
SELECT x1  
FROM table_1  
WHERE x1 IN ANY  
(SELECT x2 FROM table_2  
INTERSECT  
SELECT x3 FROM table_3
```

```
MINUS
SELECT x4 FROM table_4);
```

## Example

```
UPDATE table_1
SET x1=1
WHERE table_1.x1 IN
(SELECT x2
FROM table_2
UNION
SELECT x3
FROM table_3
UNION
SELECT x4
FROM table_4);
```

## Set Operators in INSERT ... SELECT Statements

Set operators are permitted in INSERT ... SELECT statements. The following examples demonstrate their correct use.

### Example: Simple INSERT ... SELECT Using Set Operators

```
INSERT table1 (x1,y1)
SELECT *
FROM table_2
UNION
SELECT x3,y3
FROM table_3;
```

### Example: INSERT ... SELECT from a View that Uses Set Operators

```
REPLACE VIEW v AS
SELECT *
FROM table_1
UNION
SELECT *
FROM table_2;
INSERT table_3(x3,y3)
SELECT *
FROM v;
```

## Example: INSERT ... SELECT from a Derived Table with Set Operators

```
INSERT table_1
SELECT *
FROM
(SELECT x2,y2
FROM table_2
UNION
SELECT *
FROM table_3 DerivedTable
);
```

## Set Operators in View Definitions

Set operators are permitted within view definitions.

For example, the following REPLACE VIEW statement uses UNION within a view definition:

```
REPLACE VIEW view_1 AS
  SELECT x1,y1
  FROM table_1
  UNION
  SELECT x2,y2
  FROM table_2;
```

## Support for the GROUP BY Clause

GROUP BY can be used within views with set operators.

## Restrictions

The following limitations apply to view definitions that specify set operators:

- UPDATE, DELETE, and INSERT are not applicable The following example does not work:

```
REPLACE VIEW V AS
SELECT X
FROM TABLE_1
UNION
SELECT Y FROM
TABLE_1;

UPDATE V
SET X=0;
```

An attempt to perform this sequence of statements produces the following error message:

```
***Failure 3823 VIEW 'v' may not be used for Help Index/
Constraint/Statistics, Update, Delete or Insert.
```

- WITH CHECK OPTION is not applicable The following example does not work:

```
REPLACE VIEW ERRV( c ) AS
SELECT *
FROM TABLE_1
UNION
SELECT *
FROM TABLE_2
WHERE TABLE_2.X=2 WITH CHECK OPTION;
```

An attempt to perform this statement causes the following error message:

```
***Failure 3847 Illegal use of a WITH clause.
```

- Column level privileges cannot be granted The following example does not work:

```
GRANT UPDATE ( c ) ON TABLE_VIEW TO USER_NAME;
```

An attempt to perform this statement causes the following error message:

```
***Failure 3499: GRANT cannot be used on views with set operators.
```

- A view definition that uses set operators cannot specify an ORDER BY clause, but a SELECT statement applied on the view can use ORDER BY.

## Examples

### Example 1

```
REPLACE VIEW v AS
SELECT x1
FROM TABLE_1
UNION
SELECT x2
FROM TABLE_2
UNION;
SELECT x3
FROM TABLE_3;

SELECT * FROM v;
```

## Example 2

```
REPLACE VIEW view_2 AS
  SELECT *
  FROM view_1
  UNION
  SELECT *
  FROM table_3
  UNION
  SELECT *
  FROM table_4;

SELECT *
FROM view_2
ORDER BY 1,2;
```

## Example 3

```
REPLACE VIEW v AS
  SELECT x1
  FROM table_1
  WHERE x1 IN
    (SELECT x2
     FROM table_2
     UNION
     SELECT x3
     FROM table_3
    );
SELECT * FROM v;
```

## Related Topics

For more information, see:

- For a SELECT statement applied on the view that can use ORDER BY, see [GROUP BY and ORDER BY Clauses](#).
- For GROUP BY that can be used within views with set operators, see [GROUP BY and ORDER BY Clauses](#).

## Queries Connected by Set Operators

Certain rules and restrictions apply to SELECT statements connected by set operators that might not apply elsewhere.

### Number of Expressions in SELECT Statements

All SELECT statements must have the same number of expressions.

If the first SELECT statement contains three expressions, all succeeding SELECT statements must contain three expressions.

You can use a null expression in a SELECT statement as a place holder for a missing expression.

In the following example, the second expression is null.

```
SELECT EmpNo, NULL (CHAR(5))
FROM Employee;
```

### WITH Clause

WITH clauses cannot be used in SELECT statements connected by set operators.

### GROUP BY and ORDER BY Clauses

GROUP BY clauses are allowed in individual SELECT statements of a query expression but apply only to that SELECT statement and not to the result set.

ORDER BY clauses are allowed only in the last SELECT statement of a query expression and specify the order of the result set.

ORDER BY clauses can contain only numeric literals.

For example, to order by the first column in your result set, specify ORDER BY 1.

View definitions with set operators can use GROUP BY but cannot use ORDER BY. A SELECT statement applied to a view definition with set operators can use GROUP BY and ORDER BY. The following examples are correct uses of these operations within a view definition:

```
REPLACE VIEW v AS
SELECT x1,y1
FROM table1
UNION
SELECT x2,y2
FROM table2;

SELECT *
FROM v
```

```
ORDER BY 1;

SELECT SUM(x1), y1
FROM v
GROUP BY 2;
```

You can also apply independent GROUP BY operations to each unioned SELECT. The following example demonstrates how to do this:

```
REPLACE VIEW v(column_1,column_2) AS
SELECT MIN(x1),y1
FROM table_1
GROUP BY 2
UNION ALL
SELECT MIN(x2),y2
FROM table_2
GROUP BY 2
UNION ALL
SELECT x3,y3 FROM table_3;

SELECT SUM(v.column_1) (NAMED sum_c1),column_2
GROUP BY 2
ORDER BY 2;

SELECT *
FROM table_1
WHERE (x1,y1) IN
(SELECT SUM(x2), y2
FROM table_2
GROUP BY 2
UNION
SELECT SUM(x3), y3
FROM table_3
GROUP BY 2
);
```

## Table Name in SELECT Statements

Each SELECT statement must identify the table that the data is to come from even if all SELECT statements reference the same table.



## Data Type Compatibility

Corresponding fields in each SELECT statement must have data types that are compatible. For example, if the first field in the first SELECT statement is a character data type, then the first field in each succeeding SELECT statement must be a character data type.

Corresponding numeric types do not have to be the same, but they must be compatible. For example, a field in one SELECT statement can be defined as INTEGER and the corresponding field in another SELECT statement can be defined as SMALLINT.

The data types in the first SELECT statement determine the data types of corresponding columns in the result set.

The following table provides details about data type compatibility.

Data Type	Details
Character	<p>Character types in the first SELECT statement determine the length of character strings in the result set. This can lead to truncation of character strings in the result set if the length of a character type in the first SELECT statement is less than the length of corresponding character types in succeeding SELECT statements.</p> <p>The character set of the expression in the first SELECT statement determines the character set for the entire query.</p>
Numeric	<p>Numeric types in the first SELECT statement determine the size of numeric types in the result set. All corresponding numeric fields in succeeding SELECT statements are converted to the numeric data type in the first SELECT statement. This can lead to a numeric overflow error if the size of a numeric type in the first SELECT statement is smaller than the size of corresponding numeric types in succeeding SELECT statements and the values returned by the succeeding statements do not fit into the smaller data type.</p>
TIME TIMESTAMP PERIOD(TIME) PERIOD(TIMESTAMP)	<p>TIME, TIMESTAMP, PERIOD(TIME), and PERIOD(TIMESTAMP) types in the first SELECT statement determine the precision of corresponding columns in the result set. All corresponding fields in succeeding SELECT statements are implicitly converted to the data type in the first SELECT statement. If a corresponding field does not have a time zone and the data type in the first SELECT statement does, the time zone is set to the current session time zone displacement. If the precision of a corresponding field is lower than the precision of the data type in the first SELECT statement, trailing zeros are appended to the fractional digits as needed. If the precision of corresponding fields in succeeding SELECT statements is higher than the precision of the data type in the first SELECT statement, an error is reported.</p>

## Related Topics

For more information, see:

- For examples that show how the length of the character type in the first SELECT statement affects the result set, see [Attributes of a Set Result](#).

- For examples that show how the numeric data type in the first SELECT statement affects the result set, see [Example: Effect of the Order of SELECT Statements on Data Type](#).
- The character set of the expression in the first SELECT statement determines the character set for the entire query. For more information, see [Example of How the Character Set is Determined for the Query](#).

## INTERSECT Operator

### Purpose

Returns only the rows that exist in the result of both queries.

### Syntax

```
query_expression_1 — INTERSECT ALL query_expression_2 —————▶
```

## Syntax Elements

### *query\_expression\_1*

A complete SELECT statement to be INTERSECTed with *query\_expression\_2*.

For details, see Syntax for *query\_factor*.

### ALL

Allows duplicate rows to be returned.

### *query\_expression\_2*

A complete SELECT statement to be INTERSECTed with *query\_expression\_1*.

## ANSI Compliance

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

## Rules for INTERSECT

The following rules apply to the use of INTERSECT:

- In addition to using INTERSECT within simple queries, you can use INTERSECT within the following operations:
  - Derived tables  
You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators.
  - Subqueries
  - INSERT ... SELECT statements
  - View definitions
- Each query connected by INTERSECT is executed to produce a result consisting of a set of rows. The intersection must include the same number of columns from each table in each SELECT statement (more formally, they must be of the same degree), and the data types of these columns should be compatible.
- INTERSECT cannot be used within the following:
  - SELECT AND CONSUME statements.
  - WITH RECURSIVE clause
  - CREATE RECURSIVE VIEW statements

## Attributes of a Set Result

The data type, title, and format clauses contained in the first SELECT statement in the intersection determine the data type, title, and format information that appear in the final result.

Attributes for all other SELECT statements in the query are ignored.

## Data Type of Nulls

When you specify an explicit NULL for any intersection operation, its data type is INTEGER. For an example of this principle using the UNION operator, see [Example: Effect of Explicit NULLs on Data Type of a UNION](#).

On the other hand, column data defined as NULL has neither value nor data type and evaluates like any other null in a scalar expression.

## Duplicate Row Handling

Unless the ALL option is used, duplicate rows are eliminated from the final result.

If the ALL option is specified, duplicate rows are retained. The ALL option can be specified for as many INTERSECT operators as are used in a multistatement query.

## Example

Assume that two tables contain the following rows:

SPart table			SLocation table	
SuppNo	PartNo		SuppNo	SuppLoc
100	P2		100	London
101	P1		101	London
102	P1		102	Toronto
103	P2		103	Tokyo

To then select supplier number (SuppNo) for suppliers located in London (SuppLoc) who supply part number P1 (PartNo), use the following request:

```
SELECT SuppNo FROM SLocation
WHERE SuppLoc = 'London'
INTERSECT
SELECT SuppNo FROM SPart
WHERE PartNo = 'P1';
```

The result of this request is:

```
SuppNo
-----
101
```

## MINUS/EXCEPT Operator

### Purpose

Returns the results rows that appear in *query\_expression\_1* and not in *query\_expression\_2*.

### Syntax

```
query_expression_1 [ MINUS | EXCEPT | ALL ] query_expression_2 →
```

## Syntax Elements

### *query\_expression\_1*

A complete SELECT statement whose results table is to be MINUSed with *query\_expression\_2*.

## ALL

Allows duplicate rows to be returned.

### *query\_expression\_2*

A complete SELECT statement to be MINUSed from *query\_expression\_1*.

## ANSI Compliance

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

## Usage Notes

Besides simple queries, MINUS or EXCEPT can be used within the following operations:

- Derived tables

---

### Note:

You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators.

---

- Subqueries
- INSERT ... SELECT statements
- View definitions

MINUS and EXCEPT cannot be used within the following operations:

- SELECT AND CONSUME statements.
- WITH RECURSIVE clause
- CREATE RECURSIVE VIEW statements

Each query connected by MINUS or EXCEPT is executed to produce a result consisting of a set of rows. The exception must include the same number of columns from each table in each SELECT statement (more formally, they must be of the same degree), and the data types of these columns should be compatible. All the result sets are then combined into a single result set, which has the data types of the columns specified in the first SELECT statement in the exception.

## MINUS/EXCEPT and NULL

When you specify an explicit NULL for any exception operation, its data type is INTEGER. For an example of this principle using the UNION operator, see [Example: Effect of Explicit NULLs on Data Type of a UNION](#).

On the other hand, column data defined as NULL has neither value nor data type and evaluates like any other null in a scalar expression.

## Duplicate Rows

Unless the ALL option is used, duplicate rows are eliminated from the final result.

If the ALL option is specified, duplicate rows are retained. The ALL option can be specified for as many MINUS operators as are used in a multistatement query.

## UNION Operator

### Purpose

Combines two or more SELECT results tables into a single result.

### Syntax

```
query_expression_1 — UNION — [ ALL ] — query_expression_2 —————▶
```

## Syntax Elements

### *query\_expression\_1*

A complete SELECT statement whose results table is to be MINUSed with *query\_expression\_2*.

For details, see Syntax for *query\_expression*.

### ALL

Duplicate rows are to be retained.

### *query\_expression\_2*

A complete SELECT statement to be MINUSed from *query\_expression\_1*.

## ANSI Compliance

This statement is ANSI SQL:2011 compliant.

## Valid UNION Operations

Besides simple queries, UNION can be used within the following operations:

- Derived tables

---

**Note:**

You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators.

---

- Subqueries
- INSERT ... SELECT statements
- Non-recursive CREATE VIEW statements

UNION ALL is the only valid set operator in a WITH RECURSIVE clause or CREATE RECURSIVE VIEW statement that defines a recursive query.

## Unsupported Operations

UNION cannot be used within the following:

- SELECT AND CONSUME statements.
- WITH RECURSIVE clause (unless the ALL option is also specified)
- CREATE RECURSIVE VIEW statements (unless the ALL option is also specified)

## Description of a UNION Operation

Each query connected by UNION is performed to produce a result consisting of a set of rows. The union must include the same number of columns from each table in each SELECT statement (more formally, they must be of the same degree), and the data types of these columns should be compatible. All the result sets are then combined into a single result set that has the data type of the columns specified in the first SELECT statement in the union.

## UNION and NULL

When you specify an explicit NULL for any union operation, its data type is INTEGER. For an example, see [Example: Effect of Explicit NULLs on Data Type of a UNION](#).

On the other hand, column data defined as NULL has neither value nor data type and evaluates like any other null in a scalar expression.

## Duplicate Rows

Unless the ALL option is used, duplicate rows are eliminated from each result set and from the final result.

If the ALL option is used, duplicate rows are retained for the applicable result set.

You can specify the ALL option for each UNION operator in the query to retain every occurrence of duplicate rows in the final result.

## Unexpected Row Length Errors: Sorting Rows for UNION

Before performing the sort operation used to check for duplicates in some union operations, Teradata Database creates a sort key and appends it to the rows to be sorted. If the length of this temporary data structure exceeds the system limit of 64 KB, the operation fails and returns an error to the requestor. Depending on the situation, the message text is one of the following:

- A data row is too long.
- Maximum row length exceeded in *database\_object\_name*.

See *Teradata Vantage™ - Database Messages*, B035-1096 for explanations of these messages.

## Examples

### Example: Selecting the Name, Project, and Employee Hours

To select the name, project, and the number of hours spent by employees assigned to project OE1-0001, plus the names of employees not assigned to a project, the following query could be used:

```
SELECT Name, Proj_Id, Hours
FROM Employee,Charges
WHERE Employee.Empno = Charges.Empno
AND Proj_Id IN ('OE1-0001')
UNION
SELECT Name, NULL (CHAR (8)), NULL (DECIMAL (4,2))
FROM Employee
WHERE Empno NOT IN
(SELECT Empno
FROM Charges);
```

This query returns the following rows.

Name	Project Id	Hours
Aguilar J	?	?
Brandle B	?	?
Chin M	?	
Clements D	?	?
Kemper R		
Marston A	?	?
Phan A	?	?



Name	Project Id	Hours
Regan R	?	?
Russell S	?	?
Smith T		
Watson L		
Inglis C	0E1-0001	30.0
Inglis C	0E1-001	30.5
Leidner P	0E1-001	10.5
Leidner P	0E1-001	23.0
Moffit H	0E1-001	12.0
Moffit H	0E1-001	35.5

In this example, null expressions are used in columns 2 and 3 of the second SELECT statement. The null expressions are used as place markers so that both SELECT statements in the query contain the same number of expressions.

### Example: Determining the Number and Names of Employees

To determine the department number and names of all employees in departments 500 and 600, the UNION operator could be used as follows:

```
SELECT DeptNo, Name
FROM Employee
WHERE DeptNo = 500
UNION
SELECT DeptNo, Name
FROM Employee
WHERE DeptNo = 600 ;
```

This query returns the following rows.

DeptNo	Name
500	Carter J
500	Inglis C
500	Marston A
500	Omura H

DeptNo	Name
500	Reed C
500	Smith T
500	Watson L
600	Aguilar J
600	Kemper R
600	Newman P
600	Regan R

The same results could have been returned with a simpler query, such as the following:

```
SELECT Name, DeptNo
FROM Employee
WHERE (DeptNo = 500)
OR (DeptNo = 600);
```

The advantage to formulating the query using the UNION operator is that if the DeptNo column is the primary index for the Employee table, then using the UNION operator guarantees that the basic selects are prime key operations. There is no guarantee that a query using the OR operation will make use of the primary index.

## Example: Merging Lists of Values

In addition, the UNION operator is useful if you must merge lists of values taken from two or more tables. For example, if departments 500 and 600 had their own Employee tables, the following query could be used to select data from two different tables and merge that data into a single list:

```
SELECT Name, DeptNo
FROM Employee_dept_500
UNION
SELECT Name, DeptNo
FROM Employee_dept_600 ;
```

## Example: Performing a Union Operation to Find Hours Worked

Suppose you want to know the number of man-hours charged by each employee who is working on a project. In addition, suppose you also wanted the result to include the names of employees who are not working on a project.

To do this, you would have to perform a union operation as illustrated in the following example.

```

SELECT Name, Proj_Id, Hours
FROM Employee, Charges
WHERE Employee.EmpNo = Charges.EmpNo
UNION
SELECT Name, Null (CHAR(8)), Null (DECIMAL(4,2)),
FROM Employee
WHERE EmpNo NOT IN
(SELECT EmpNo
FROM Charges
)
UNION
SELECT Null (VARCHAR(12)), Proj_Id, Hours
FROM Charges
WHERE EmpNo NOT IN
(SELECT EmpNo
FROM Employee
);

```

The first portion of the statement joins the Employee table with the Charges table on the EmpNo column. The second portion accounts for the employees who might be listed in the Employee table, but not the Charges table. The third portion of the statement accounts for the employees who might be listed in the Charges table and not in the Employee table. This ensures that all the information asked for is included in the response.

## UNION Operator and the Outer Join

[Example: Performing a Union Operation to Find Hours Worked](#) does not illustrate an outer join. That operation returns all rows in the joined tables for which there is a match on the join condition and rows from the “left” join table, or the “right” join table, or both tables for which there is no match. Moreover, non-matching rows are extended with NULLs.

It is possible, however, to achieve an outer join using inner joins and the UNION operator, though the union of any two inner joins is not the equivalent of an outer join.

The following example shows how to achieve an outer join using two inner joins and the UNION operator. Notice how the second inner join uses NULLs.

```

SELECT Offering.CourseNo, Offerings.Location, Enrollment.EmpNo
FROM Offerings, Enrollment
WHERE Offerings.CourseNo = Enrollment.CourseNo
UNION
SELECT Offerings.CourseNo, Offerings.Location, NULL
FROM Offerings, Enrollment
WHERE Offerings.CourseNo <> Enrollment.CourseNo;

```

The above UNION operation returns results equivalent to the results of the left outer join example shown above.

O.CourseNo	O.Location	E.EmpNo
C100	El Segundo	235
C100	El Segundo	668
C200	Dayton	?
C400	El Segundo	?

### Example: Effect of Explicit NULLs on Data Type of a UNION

Set operator results evaluate to the data type of the columns defined in the first SELECT statement in the operation. When a column in the first SELECT is defined as an explicit NULL, the data type of the result is not intuitive.

Consider the following two examples, which you might intuitively think would evaluate to the same result but do not.

In the first, an explicit NULL is selected as a column value.

```
SELECT 'p', NULL
FROM TableVM
UNION
SELECT 'q', 145.87
FROM TableVM;
```

BTEQ returns the result as follows.

```
'p'      Null
---  -----
p          ?
q         145
```

The expected value for the second row of the Null column probably differs from what you might expect—a decimal value of 145.87.

What if the order of the two SELECTs in the union is reversed?

```
SELECT 'q', 145.87
FROM TableVM
UNION
SELECT 'p', NULL
FROM TableVM;
```

BTEQ returns the result as follows.

```
'q'      145.87
-----
p          ?
q      145.87
```

The value for q is now reported as its true data type—DECIMAL—and without truncation. Why the difference?

In the first union example, the explicit NULL is specified for the second column in the first SELECT statement. The second column in the second SELECT statement, though specified as a DECIMAL number, evaluates to an integer because in this context, NULL, though having no value, does have the data type INTEGER, and that type is retained for the result of the union.

The second union example carries the data type for the value 145.87—DECIMAL—through to the result. You can confirm the unconverted data type for NULL and 145.87 by performing the following SELECT statement.

```
SELECT TYPE(NULL), TYPE(145.87)
```

BTEQ returns the result as follows.

```
Type(Null)      Type(145.87)
-----
INTEGER          DECIMAL(5,2)
```

## Example: Effect of the Order of SELECT Statements on Data Type

The result of any UNION is always expressed using the data type of the selected value of the first SELECT. This means that SELECT A UNION SELECT B does not always return the same result as SELECT B UNION SELECT A unless you explicitly convert the output data type to ensure the same result in either case.

Consider the following complex unioned queries:

```
SELECT MIN(X8.i1)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1=X1.i1
AND X8.i1 IN
(SELECT COUNT(*)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1=X1.i1
AND X8.i1 = ANY
(SELECT COUNT(*)
FROM t7 X7
```

```

WHERE X7.i1 = ANY
(SELECT AVG(X1.i1)
FROM t1 X1)))
UNION
SELECT AVG(X4.i1)
FROM t4 X4
WHERE X4.i1 = ANY
(SELECT (X8.i1)
FROM t1 X1
RIGHT JOIN t8 X8 ON X8.i1=X1.i1
AND X8.i1 = IN
(SELECT MAX(X8.i1)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1=X1.i1
AND
(SELECT (X4.i1)
FROM t6 X6
RIGHT JOIN t4 X4 ON X6.i1=i1))));

```

The result is the following report.

```

Minimum(i1)
-----
          -2
           0

```

You might intuitively expect that reversing the order of the queries on either side of the UNION would produce the same result. Because the data types of the selected value of the first SELECT can differ, this is not always true, as the following query on the same database demonstrates.

```

SELECT AVG(X4.i1)
FROM t4 X4
WHERE X4.i1 = ANY
(SELECT (X8.i1)
FROM t1 X1
RIGHT JOIN t8 X8 ON X8.i1 = X1.i1
AND X8.i1 = ANY
(SELECT MAX(X8.i1)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1 = X1.i1
AND
(SELECT (X4.i1)
FROM t6 X6
RIGHT JOIN t4 X4 ON X6.i1 = i

```

```

)
)
)
UNION
SELECT MIN(X8.i1)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1 = X1.i1
AND X8.i1 IN
(SELECT COUNT(*)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1 = X1.i1
AND X8.i1 = ANY
(SELECT COUNT(*)
FROM t7 X7
WHERE X7.i1 = ANY
(SELECT AVG(X1.i1)
FROM t1 X1
)
);

```

The result is the following report.

```

Average(i1)
-----
          -2
           1

```

The actual average is < 0.5. Why the difference when the order of SELECTs in the UNION is reversed? The following table explains the seemingly paradoxical results.

WHEN the first SELECT specifies this function ...	The result data type is ...	AND the value returned as the result is ...
AVG	REAL	1
MIN	INTEGER	truncated to 0

## Example: LOB Support in a UNION ALL Query

Previously, columns with CLOB or BLOB data type were not allowed in any set operation query, as it is not possible to hash on a CLOB or BLOB column. UNION ALL does not need to hash on any column (including a LOB), that restriction is removed with the the UNION ALL optimizations feature.

Assume the following tables:

```
CREATE TABLE t_blob1 (a1 INTEGER, b1 BLOB(2097088000));  
CREATE TABLE t_clob2 (a2 INTEGER, b2 CLOB);
```

Consider the following query:

```
-- Prior to Teradata Database 16.0:  
SELECT b1 FROM t_clob1  
UNION ALL  
SELECT b2 FROM t_clob2;  
*** Failure 5690 LOBs are not allowed to be hashed.  
          Statement# 1, Info = 0  
  
-- Teradata Database 16.0:  
SELECT b1 FROM t_clob1  
UNION ALL  
SELECT b2 FROM t_clob2;  
*** Query completed.  8 rows found.
```

## Related Topics

For more information, see:

- For an example, see [Example: Effect of the Order of SELECT Statements on Data Type](#).



# Join Expressions

## Overview

These topics describe joins of tables and views.

For an outer join case study that shows how to write correct outer joins that provide the results you intend, see [Outer Join Case Study](#).

For information on the algorithms used by the Optimizer to perform these joins, see *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

## Joins

A join is an action that projects columns from two or more tables into a new virtual table. Teradata Database supports joins of as many as 128 tables and single-table views per query block.

A join can also be considered an action that retrieves column values from more than one table.

See [Outer Join Relational Algebra](#) for definitions of the relational algebra terms project, restrict, and product.

### Join Varieties

In queries that join the tables, you can specify four basic types of join:

- Natural
- $\Theta$  (theta)
- Inner
- Outer

You can combine join types, for example, make natural inner and outer joins as well as  $\Theta$  inner and outer joins.

### Natural and Theta Joins

The natural join and the  $\Theta$  (theta) join are the two basic types of join.

The natural join is an equality join made over a common column set with matching column names such as a primary index or primary key-foreign key relationship that is expressed in a WHERE clause equality condition. For example,

```
... WHERE a.custnum = b.custnum ...
```

The  $\Theta$  join is a more general type of join where you can use conditions such as greater than ( $>$ ), greater than or equal to ( $\geq$ ), less than ( $<$ ), less than or equal to ( $\leq$ ), in addition to equality ( $=$ ). For example,

```
... WHERE a.custnum > b.custnum ...
```

If the  $\Theta$  operator is equality ( $=$ ), then the join is considered an equijoin. In certain cases, the natural join and the equijoin are identical. Because Teradata SQL does not support the ANSI SQL-2011 NATURAL JOIN syntax, it does not recognize a practical difference between natural and equijoins, which are considered equivalent. While both joins are made over an equality condition, the column names on which tables are joined need not match in an equijoin, while they must match in a natural join.

SQL also supports two other special join cases: the cross join, or Cartesian product, and the self-join.

For more detailed information about these join types, see the following topics:

- [Inner Joins](#)
- [Ordinary Inner Join](#)
- [Cross Join](#)
- [Self-Join](#)

Note that intersection and Cartesian product are special cases of the join operation. See [INTERSECT Operator](#).

## Inner and Outer Joins

The inner join projects only those rows that have specific commonality between the joined tables. Because the inner join does not include rows that have no counterpart in the other table, it is sometimes said to lose that information.

For more information about inner joins, see:

- [Inner Joins](#)
- [Ordinary Inner Join](#)

The outer join is meant to provide a method for regaining the "lost" information the inner join does not report. The outer join is an extended inner join that projects not only those rows that have commonality between the joined tables, but also those rows that have no counterpart in the other relation.

Depending on how you write an outer join, it can project the inner join rows plus any of the following: the nonmatching rows of the left table, the nonmatching rows of the right table, or the nonmatching rows from both. The attributes of the complementary row sets of an outer join are represented in the result set by nulls.

For more information about the outer join and its various types, see:

- [Outer Joins](#)
- [Left Outer Join](#)
- [Right Outer Join](#)
- [Full Outer Join](#)

Outer joins are somewhat controversial for several reasons:

- There are some formal complications involved in deriving outer joins. For example, while it is true that the inner natural join is a projection of the inner equijoin, it is not true that the outer natural join is a projection of the outer equijoin.

- The result of an outer join can be very difficult to interpret, if only because nulls are used to represent two very different things: the standard "value unknown" meaning and the empty set, representing the attributes of the outer row set.

For more information about issues with nulls and the outer join, see *Teradata Vantage™ - Database Design*, B035-1094.

- It is often difficult to code an outer join correctly.

For more information about coding outer joins, see:

- [Coding ON Clauses for Outer Joins](#)
- [Coding ON Clauses With WHERE Clauses for Outer Joins](#)
- [Rules for Coding ON and WHERE Clauses for Outer Joins](#)
- [Outer Join Case Study](#)

## Joining Tables and Views That Have Row-Level Security Constraints

Teradata Database supports joining tables and views that have row-level security constraints if the tables or views have the same row-level security constraints. Otherwise, the system returns an error.

For more information about row-level security constraints, see *Teradata Vantage™ NewSQL Engine Security Administration*, B035-1100 and "CREATE CONSTRAINT" in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

## Inner Joins

The inner join is usually referred to simply as a join. This section includes the following topics:

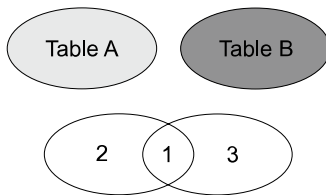
- [Ordinary Inner Join](#)
- [Cross Join](#)
- [Self-Join](#)

## Ordinary Inner Join

A join allows you to select columns and rows from two or more tables and views. You can join as many as 128 tables and views per query block.

An *inner* join projects data from two or more tables or views that meet specific join conditions. Each source must be named and the join condition, the common relationship among the tables or views to be joined, must be specified explicitly in a WHERE clause.

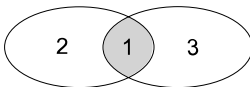
## Components of an Inner Join



Define *table\_A* as the row set containing sections 1 and 2. Refer to *table\_A* as *left\_table*.

Define *table\_B* as the row set containing sections 1 and 3. Refer to *table\_B* as *right\_table*.

The inner join of *table\_A* and *table\_B* is section 1, as indicated by the following Venn diagram.



## The Default Join

Unless explicitly declared as outer joins, all joins are inner joins by default. You can, however, specify an explicit inner join using the reserved word `INNER`.

The following `SELECT` statement has an inner join of the two tables *table\_a* and *table\_b*.

```
SELECT ...
FROM table_a
INNER JOIN table_b ...
```

Because the keyword sequence `INNER JOIN` is optional (but if you specify the keyword `INNER`, you must immediately follow it with the keyword `JOIN`), the following `SELECT` statements are also correct examples of inner joins:

```
SELECT ...
FROM table_a
JOIN table_b ...
SELECT ...
FROM table_a, table_b ...
```

You can specify the join condition using either the `ON` clause or the `WHERE` clause for inner joins. Note that an `ON` clause is mandatory if the keyword `JOIN` is specified for an inner join.

Note that you cannot specify a `SAMPLE` clause in a subquery used as an `ON` clause predicate.

## Example: Default Join

You can determine the department location of employee Marston by joining the *employee* and *department* tables on the column in which there are values common to both. In this case, that column is *deptno*.

```
SELECT Loc
FROM department, employee
WHERE employee.name = 'Marston A'
AND employee.deptno = department.deptno;
```

This query asks two questions:

- What is the number of the department in the *employee* file that contains Marston?
- What is the location code for that department in the *department* file?

The key to answering both questions is the *deptno* column, which has the same values in both tables and thus can be used to make the join relationship (the actual name of the column is irrelevant).

The result of this inner join is the following report.

```
Loc
ATL
```

### Joins on Views Containing an Aggregate

You can join views containing an aggregate column.

In the following example, the *cust\_file* table is joined with the *cust\_prod\_sales* view (which contains a SUM operation) in order to determine which companies purchased more than \$10,000 worth of item 123.

```
CREATE VIEW cust_prod_sales (cust_no, pcode, sales) AS
SELECT cust_no, pcode, SUM(sales)
FROM sales_hist
GROUP BY cust_no, pcode;

SELECT company_name, sales
FROM cust_prod_sales AS a, cust_file AS b
WHERE a.cust_no = b.cust_no
AND a.pcode = 123
AND a.sales > 10000;
```

### Joins on PERIOD Value Expressions

The Optimizer treats an equality operator on a PERIOD value expression in the same way as any other expression for join or access planning. See “Period Value Constructor” in *Teradata Vantage™ SQL Date and Time Functions and Expressions*, B035-1211.

If the join predicate is an inequality on a PERIOD value expression, Teradata Database processes it as a Cartesian product just as it would any other inequality predicate. Row hashing based on a PERIOD value expression during row redistribution considers the internal field size and value for hash value computation.

```
SELECT *
FROM employee AS e, project_details AS pd
ON e.period_of_stay = pd.project_period;
```

## Cross Join

SQL supports unconstrained joins (or cross joins), which are joins for which a WHERE clause relationship between the joined tables is not specified. The result of an unconstrained join is also referred to as a Cartesian product.

The collection of all such ordered pairs formed by multiplying each element of one relation with each element of a second relation is the same result obtained by performing a cross join between two tables: the join is the product of the rows in each table that is joined.

### Reasons for Performing a Cross Join

Concatenating each row of one table with every row of another table rarely produces a useful result. In general, the only reason to request a cross join is to write performance benchmarks. Real world applications of cross joins are essentially nonexistent.

Before performing a cross join, you should address the following considerations:

- Why you need to execute such a wide ranging and uninformative operation
- How expensive (with respect to resource consumption) a cross join can be

For example, a cross join of two tables, `table_a` and `table_b`, each with a cardinality of 1,000 rows, returns a joined table of 1 million rows. Cross joins can easily abort transactions by exceeding user spool space limits.

### Specifying a Cross Join

If you want to return a Cartesian product that is not an outer join of two or more tables, you can write the following SELECT statement:

```
SELECT ...  
FROM table_a  
CROSS JOIN table_b;
```

Because the reserved word sequence `CROSS JOIN` is optional, you could also write the following SELECT statement that accomplishes the same result:

```
SELECT ...  
FROM table_a,table_b;
```

The first form, with the explicit `CROSS JOIN` syntax, more clearly indicates that a cross join is the intent of the query.

If `table_a` contains five rows and `table_b` contains three rows, then the Cartesian product is 3 x 5, or 15. An unconstrained join on these tables results in 15 rows being returned.

## Self-Join

A self-join combines the information from two or more rows of the same table into a single result row, effectively joining the table with itself.

### Example: Self-Join

For example, the following query asks the names of employees who have more years of experience than their managers:

```
SELECT workers.name, workers.yrs_exp, workers.dept_no,
       managers.name, managers.yrsexp
FROM employee AS workers, employee AS managers
WHERE managers.dept_no = workers.dept_no
AND   managers.job_title IN ('Manager', 'Vice Pres')
AND   workers.yrs_exp > managers.yrs_exp;
```

The operation treats the *employee* table as if it were two tables; one named *Workers*, the other named *Managers*. This is accomplished by using table name aliases in the FROM clause.

ANSI calls table aliases correlation names. They are also referred to as range variables.

Because each of these fictitious tables has the same columns (*name*, *yrs\_exp*, and *dept\_no*) each column name must be qualified with the alias name of its table as defined in the FROM clause (for example, “workers.dept\_no”).

The WHERE clause establishes the following things:

- A key to both tables (*dept\_no*)
- Which employees belong in the *managers* table (first AND)
- Which workers and managers should be listed in the tables (second AND)

A possible result of this self-join is as follows:

name	yrs_exp	dept_no	name	yrs_exp
Greene W	15	100	Jones M	13
Carter J	20	500	Watson L	8
Smith T	10	700	Watson L	8
Aguilar J	11	600	Regan R	10
Leidner P	13	300	Phan A	12
Russell S	25	300	Phan A	12

## Outer Joins

The outer join is an extension of the inner join that includes both rows that qualify for a simple inner join as well as a specified set of rows that do not match the join conditions expressed by the query.

Because outer joins are as complicated to code correctly as they are to interpret, this section studies them in detail, including a carefully thought out case study and series of recommendations contributed by a Teradata database administrator.

There are three types of outer joins:

- LEFT OUTER JOIN (see [Left Outer Join](#))
- RIGHT OUTER JOIN (see [Right Outer Join](#))
- FULL OUTER JOIN (see [Full Outer Join](#))

## Definition of the Outer Join

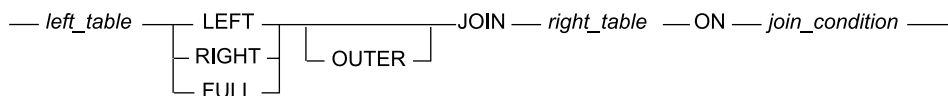
The outer join is an extension of the inner join. The difference between outer joins and inner joins is the existence of an nonmatching row set in the outer join result.

Outer joins of two or more tables perform an inner join of those tables according to a specified join condition and also return rows from the left join table, the right join table, or both, that do not match the inner join condition, extending the results rows with nulls in the nonmatching fields.

The nonmatching rows are for those rows in the outer table that do not have a match in the inner table, while the matching rows in the outer join result are the rows that satisfy the join condition, which are exactly the same as those in the inner join result.

The outer join is generally defined as the algebraic UNION ALL of the components. UNION ALL permits duplicates and so is not a relational operator in the strict sense of the term.

### Syntax



### Syntax Elements

#### *left\_table*

Table reference that appears to the left of the join type keywords.

#### *right\_table*

Table reference that appears to the right of the join type keywords.

#### *join\_condition*

Columns on which the join is made separated by the comparison operator that specifies the comparison type for the join.



The join conditions of an ON clause define the rows in the left table that take part in the match to the right table. At least one join condition is required in the ON clause for each table in the outer join.

You can include multiple join conditions in an ON clause by using the Boolean AND, OR, and NOT operators.

### Example: SELECT with Join

For example, consider the following SELECT statement:

```
SELECT offerings.course_no, offerings.location, enrollment.emp_no
FROM offerings
LEFT OUTER JOIN enrollment
ON offerings.course_no = employee.course_no;
```

where:

Object	Corresponding Syntax Element
offerings	<i>left_table</i>
enrollment	<i>right_table</i>
offerings.course_no=employee.course_no	<i>join_condition</i>

### Rule for Specifying a SAMPLE Clause in an ON Clause Subquery

You cannot specify a SAMPLE clause in a subquery used as an ON clause predicate.

### Simple Outer Join Example

Assume you have the data presented in tables *t1* and *t2*:

t1			t2	
FK			PK	
x1	y1		x2	y2
1	1		1	1
2	2		3	3
3	3		4	4
4	4		5	5

An inner join of these tables is specified by the following query:

```
SELECT x1, y1, x2, y2
FROM t1, t2
WHERE x1=x2;
```

x1	y1	x2	y2
1	1	1	1
3	3	3	3
4	4	4	4

An left outer join of these tables is specified by the following query:

```
SELECT x1, y1, x2, y2
FROM t1 LEFT OUTER JOIN t2 ON x1=x2;
```

The result of this query is as follows:

x1	y1	x2	y2
1	1	1	1
2	2	Null	Null
3	3	3	3
4	4	4	4

### Outer Join Is Equivalent to Inner Join for Selected Foreign Key-Primary Key Relationships

In the original table definitions, x2 is the primary key of t2. Assume the following referential integrity constraint is defined on t1:

```
FOREIGN KEY (x1) REFERENCES t2 (x2);
```

This constraint requires each of the x1 (foreign key) values in t1 to also exist as the primary key for one, and only one, row in table t2 as column x2. Therefore the second row in t1 cannot be valid, which makes the nonmatching row in the outer join result disappear from the outcome. See [Simple Outer Join Example](#). The outer join result converts into the inner join result on the same tables because the referential integrity constraint guarantees there can be no unmatched rows in the join result.

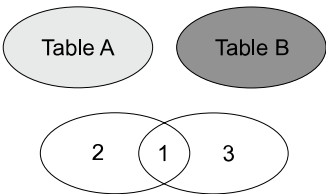
Mathematically, the foreign key-primary key relationship implies that the values in the foreign key column set form a proper subset of the set of values in the primary key column set, expressed mathematically as  $S_{FK} \subset S_{PK}$ , where  $S_{FK}$  represents the foreign key value set and  $S_{PK}$  represents the primary key value set.

Therefore, an equijoin between the parent table primary key column set and a child table foreign key column set always leaves the unmatched row set empty so that an outer join is equivalent to the corresponding

inner join. This property is extremely valuable for query optimization using join indexes. For more information, see *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184 and *Teradata Vantage™ - Database Design*, B035-1094.

Components of an Outer Join

Refer to the following table abstractions and Venn diagram.



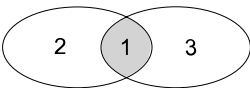
This diagram is the same as [Components of an Inner Join](#). However, the way the components of this diagram are combined is very different.

Define table\_A as the row set contained by sections 1 and 2. Refer to table\_A as left\_table.

Define table\_B as the row set contained by sections 1 and 3. Refer to table\_B as right\_table.

Define the components of a 2-table outer join as follows:

Component	Definition
1	Inner join of the 2 tables as defined by an ON clause with all join conditions applied. These rows also provide the matching rows of an outer join.
2	All rows from left_table not included in section 1 extended with nulls for each nonmatching column from left_table. These are the unmatched rows in a left outer join.
3	All rows from right_table not included in section 1 extended with nulls for each nonmatching column selected from right_table. These are the unmatched rows in a right outer join.



Types of Outer Join

The sections to be unioned are defined by the type of outer join:

Outer Join Type	Definition
LEFT	Section 1 UNION ALL Section 2
RIGHT	Section 1 UNION ALL Section 3
FULL	Section 1 UNION ALL Section 2 UNION ALL Section 3

For each type of outer join, consider the operation as assembling the proper components using the UNION ALL operator. Because UNION ALL permits duplicates, it is not a relational operator in the strict sense of the term.

### Inner Table and Outer Table

The terms inner table and outer table, used frequently when writing about outer joins, are defined in the following table:

Term	Definition
Inner table	The inner table is the table that contributes only matched rows to the outer join result. For an inner join, both tables are inner tables.
Outer table	The outer table is the table that contributes unmatched rows to the outer join result. In this description, unmatched refers to the rows in the left or right (or both) table that are not part of the inner join rows because there are no matching columns, so the rows are extended with nulls in the results table.

This terminology also applies to the result of nested joins and spools.

### Inner/Outer Table Example

Consider the following nested outer join:

```
(table_A
  LEFT OUTER JOIN
    (table_B
      RIGHT OUTER JOIN table_C ON join_condition
    )
  ON join_condition
)
FULL OUTER JOIN table_D ON join_condition
```

The inner and outer tables of this outer join are as follows.

Beginning with the most deeply nested join and working outward, these relationships apply:

1. The table\_c is an outer table with respect to table\_b
2. The table\_a is an outer table with respect to the nested join (table\_b RIGHT OUTER JOIN table\_c ON join\_condition)
3. (table\_a LEFT OUTER JOIN (table\_b RIGHT OUTER JOIN table\_c ON join\_condition) ON join\_condition is an outer table for the full outer join
4. The table\_d is an outer table for the full outer join

### Projected Column List

When you construct an outer join, choose the projected column list carefully to ensure the results are useful and interpretable. In general, you should project the column from the same side as the outer join. [Practical](#)

[Example of a Right Outer Join](#) right outer joins on CourseNo. Therefore, project out the right join column, which is Enrollment.CourseNo.

Similarly, in the full outer join in [Practical Example of a Full Outer Join](#), the example projects both CourseNo columns for this same reason.

## Nulls and the Outer Join

Nulls are a fundamental component of the report produced by a query that contains an outer join. The key feature of the outer join is that in returning rows from the outer table set, the report extends the rows that have no matching values with nulls, as if these unknown “values” came from the corresponding table.

For example, if you want to list courses offered by customer education for which employees have registered and also to include in that list those courses for which no one signed up.

The practical outer join examples in [Left Outer Join](#), [Right Outer Join](#), and [Full Outer Join](#) use these three tables.

- The table offerings shows customer education courses currently being offered and their location. With respect to courses being offered, you can think of offerings as a subset of courses.

offerings		
course_no	beginning_dates	location
C100	01/05/2006	El Segundo
C200	07/02/2006	Dayton
C400	10/07/2006	El Segundo

- The table enrollment shows employees who have registered for courses, some of which may not be offered.

enrollment	
emp_no	course_no
236	C100
236	C300

- The table courses lists all courses developed by customer education, some of which are not currently being offered (for example, C300).

courses	
course_no	name
C100	Recovery Planning
C200	Software Architecture

courses	
C300	Teradata Basics
C400	Introduction to Java Programming

The nulls reported for an outer table set do not represent missing information, but the empty set. For a description of the various uses of nulls in the SQL language, see *Teradata Vantage™ - Database Design*, B035-1094.

### Example: Left Outer Join

The following SELECT statement is an example of a left outer join of two tables, table\_a and table\_b:

```
SELECT column_expression
FROM table_a
LEFT OUTER JOIN table_b ON join_conditions
WHERE join_condition_exclusions;
```

The left table in the above example means the table specified to the left of the keywords OUTER JOIN in the FROM clause. The right table means the table specified to the right of the keywords OUTER JOIN.

The keyword LEFT indicates the source of the nonmatching rows returned in the result of the left outer join.

In addition to performing an inner join of two or more tables according to a join condition, a left outer join, as in the example above, returns nonmatching rows from its left table (table\_a) and extends them with nulls.

A right outer join returns nonmatching rows from its right table and extends the rows with nulls.

A full outer join returns nonmatching rows from both of the join tables and extends the rows with nulls.

The reserved word OUTER is optional so that the preceding SELECT statement could also be written as follows:

```
SELECT ...
FROM table_A
LEFT JOIN table_B ON join_condition;
```

### Example: Scalar Subquery in the ON Clause of a Left Outer Join

You can specify a scalar subquery as an operand of a scalar predicate in the ON clause of an outer join specification.

The following example specifies a scalar subquery (SELECT AVG(price)...) in its ON clause.

```
SELECT category, title, COUNT(*)
FROM movie_titles AS t2 LEFT OUTER JOIN transactions AS txn
ON (SELECT AVG(price)
    FROM movie_titles AS t1
    WHERE t1.category = t2.category)<(SELECT AVG(price)
```

```

                                FROM movie_titles)
AND t2.title = txn.title;

```

## Rules for Using the DEFAULT Function as a Search Condition for an Outer Join ON Clause

The following rules apply to the use of the DEFAULT function as part of the search condition within an outer join ON clause:

- The DEFAULT function takes a single argument that identifies a relation column by name. The function evaluates to a value equal to the current default value for the column. For cases where the default value of the column is specified as a current built-in system function, the DEFAULT function evaluates to the current value of system variables at the time the request is processed.

The resulting data type of the DEFAULT function is the data type of the constant or built-in function specified as the default unless the default is NULL. If the default is NULL, the resulting data type of the DEFAULT function is the same as the data type of the column or expression for which the default is being requested.

- The DEFAULT function has two forms: DEFAULT or DEFAULT (*column\_name*). When no column name is specified, the system derives the column based on context. If the column context cannot be derived, an error is returned to the requestor.
- You can specify a DEFAULT function with a column name argument within a predicate. The system evaluates the DEFAULT function to the default value of the column specified as its argument. Once the system has evaluated the DEFAULT function, the function is calculated as a constant in the predicate.
- You can specify a DEFAULT function without a column name argument within a predicate only if there is one column specification and one DEFAULT function as the terms on each side of the comparison operator within the expression.
- Following existing comparison rules, a condition with a DEFAULT function used with comparison operators other than IS NULL or IS NOT NULL is unknown if the DEFAULT function evaluates to null.

A condition other than IS NULL or IS NOT NULL with a DEFAULT function compared with a null evaluates to unknown.

DEFAULT Function and NULL	Comparison Result
IS NULL	<ul style="list-style-type: none"> <li>• TRUE if the default is null</li> <li>• Otherwise FALSE</li> </ul>
IS NOT NULL	<ul style="list-style-type: none"> <li>• FALSE if the default is null</li> <li>• Otherwise TRUE</li> </ul>

For more information about the DEFAULT function, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

## Use of the DEFAULT Function in an Outer Join ON Condition

The examples assume the following table definitions:

```
CREATE TABLE table15 (
  col1 INTEGER ,
  col2 INTEGER NOT NULL,
  col3 INTEGER NOT NULL DEFAULT NULL,
  col4 INTEGER CHECK (col4 > 10) DEFAULT 9 );
CREATE TABLE table16 (
  col1 INTEGER,
  col2 INTEGER DEFAULT 10,
  col3 INTEGER DEFAULT 20,
  col4 CHARACTER(60) );
```

You can specify a DEFAULT function as a component of the search condition for an outer join. The following examples demonstrate proper use of the DEFAULT function as part of an outer join search condition.

In the following example, the DEFAULT function evaluates to the default value of table16.col2, which is 10.

```
SELECT *
FROM table16
FULL OUTER JOIN table15 ON table15.col1 < DEFAULT(table16.col2);
```

Therefore, the previous example is equivalent to the following example:

```
SELECT *
FROM table16
FULL OUTER JOIN table15 ON table15.col1 < 10;
```

## Outer Join Relational Algebra

The relational algebra and the relational calculus are two different, but equivalent, formal languages for manipulating relations. Algebra is procedural, for internal representations of queries that can be manipulated by query optimizers and database managers, while the calculus is nonprocedural, providing a foundation for user-malleable query languages.

The basic query SELECT-FROM-WHERE, which is the SQL dialect of the generalized nonprocedural relational calculus for a query, can be restated in terms of relational algebra as follows.

```
projection ( restriction ( product ) )
```

where:



Syntax element ...	Specifies ...
projection	the result of applying a PROJECT operator that extracts one or more attributes from one or more relations. This defines an SQL SELECT ... FROM. Projection selects columns from tables.
restriction	the result of applying a RESTRICT (or SELECT) operator that extracts one or more tuples from the projection. Note that the SELECT of relational algebra is different than the SELECT statement of SQL, which essentially performs both a PROJECT operation and a RESTRICT operation. Restriction defines the WHERE, ON, , and QUALIFY clauses of an SQL SELECT statement. Restriction selects qualified rows from tables. When a join condition exists, restriction selects qualified rows from the intermediate results table produced by the join.
product	the result of applying a PRODUCT operator that builds all possible combinations of tuples, one from each specified relation. This defines a join, specifically, an <i>inner</i> join.

For clarity of presentation, restate the original expression as projection (inner join).

### Relational Algebra for the Outer Join

The outer join merges the result of an inner join with the remaining tuples in one (or both) of the joined relations that do not share commonality with the tuples of the inner join result table.

The outer join can be expressed in relational algebra as follows:

```
projection ( inner_join UNION ALL extension )
```

where:

Syntax element ...	Specifies ...
projection	the result of applying a PROJECT operator that extracts one or more attributes from one or more relations.
inner_join	the result of applying the RESTRICT (or SELECT) operator to the PRODUCT of the relations in the projection.
UNION ALL	the UNION ALL operator. Because UNION ALL permits duplicates, it is not a relational operator in the strict sense of the term.
extension	the complement of the result of applying the RESTRICT operator to the PRODUCT of the relations in the projection. Depending on how the query is stated, extension can refer either to the excluded tuples from the left table, the right table, or both. These are, respectively, left, right, and full outer joins.

## Left Outer Join

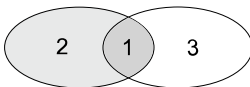
When you perform a left outer join on the Offerings and Enrollment tables, the rows from the left table that are not returned in the result of the inner join of these two tables are returned in the outer join result and extended with nulls.

### Inner/Outer Table Example

The following example uses the explicit table names `inner_table` and `outer_table` to indicate how these terms relate to the way a simple left outer join is constructed in the FROM clause of a SELECT statement. See [Inner Table and Outer Table](#)

The example shows the semantics of inner and outer table references for a left outer join.

```
outer_table LEFT OUTER JOIN inner_table
```



Section 1 represents the inner join  $\cap$  (intersection) of `outer_table` and `inner_table`. Section 2 represents the unmatched rows from the outer table.

The outer join result contains the matching rows from Sections 2 and 3, indicated in the diagram as Section 1, plus unmatched rows from Section 2, noted in the graphic by the more lightly shaded component of the Venn diagram.

In terms of the algebra of sets, the result is

```
(Table_A  $\cap$  Table_B) + (Table_A - Table_B)
```

where:

`Table_A  $\cap$  Table_B` is the set of matched rows from the inner join of `Table_A` and `Table_B`.

`Table_A - Table_B` is the set of unmatched rows from `Table_A`.

### Practical Example of a Left Outer Join

The following SELECT statement yields the results in the table that follows:

```
SELECT offerings.course_no,offerings.location,enrollment.emp_no
FROM offerings
LEFT OUTER JOIN enrollment ON offerings.course_no =
                           enrollment.courseno;
```

o.course_no	o.location	e.emp_no
C100	El Segundo	236
C100	El Segundo	668

C200	Dayton	Null
C400	El Segundo	Null

BTEQ reports represent nulls with the QUESTION MARK character.

These results show that course C100 has two enrolled employees and that courses C200 and C400 have no enrolled employees. In this case, the nulls returned by the outer join of the offerings and enrollment tables provide meaningful information.

The keyword OUTER in the FROM clause is optional, so you can also write the above SELECT statement as follows:

```
SELECT offerings.course_no,offerings.location,enrollment.emp_no
FROM offerings
LEFT JOIN enrollment ON offerings.course_no = employee.course_no;
```

## Right Outer Join

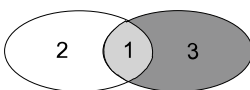
In a right outer join, the rows from the right table that are returned in the result of the inner join are returned in the outer join result and extended with nulls.

### Inner/Outer Table Example

The following example uses the explicit table names `inner_table` and `outer_table` to indicate how these terms relate to the way a simple right outer join is constructed in the FROM clause of a SELECT statement. See [Inner Table and Outer Table](#).

The example shows the semantics of inner and outer table references for a right outer join.

```
inner_table RIGHT OUTER JOIN outer_table
```



Section 1 represents the inner join (intersection) of `outer_table` and `inner_table`. Section 3 represents the unmatched rows from the outer table.

The outer join result contains the matching rows from Sections 2 and 3, indicated in the diagram as Section 1, plus the unmatched rows from Section 3, noted in the graphic by the more darkly shaded component of the Venn diagram.

In terms of the algebra of sets, the result is:

```
(Table_A ∩ Table_B) + (Table_B - Table_A)
```

where:

$\text{Table\_A} \cap \text{Table\_B}$  is the set of matched rows from the inner join of `Table_A` and `Table_B`.

Table\_B - Table\_A is the set of unmatched rows from Table\_B.

### Practical Example of a Right Outer Join

When you perform a right outer join on the offerings and enrollment tables, the rows from the right table that are not returned in the result of the inner join are returned in the outer join result and extended with nulls.

This SELECT statement returns the results in the following table:

```
SELECT offerings.course_no, offerings.location, enrollment.emp_no
FROM offerings
RIGHT OUTER JOIN enrollment
ON offerings.course_no = enrollment.course_no;
```

o.course_no	o.location	e.emp_no
C100	El Segundo	236
C100	El Segundo	668
Null	Null	236

BTEQ reports represent nulls with the QUESTION MARK character.

These results show that course C100 has two employees enrolled in it and that employee 236 has not enrolled in another class. But in this case the nulls returned by the right outer join of the offerings and enrollment tables are deceptive, because we know by inspection of the enrollment table that employee 236 has enrolled for course C300. We also know by inspection of the offerings table that course C300 is not currently being offered.

For more informative results, use the following right outer join:

```
SELECT enrollment.course_no, offerings.location, enrollment.emp_no
FROM offerings
RIGHT OUTER JOIN enrollment
ON offerings.course_no = enrollment.course_no;
```

This query returns the row (C300, Null, 236), not (Null, Null, 236).

### Full Outer Join

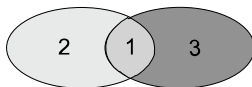
In a full outer join, rows from both tables that have not been returned in the result of the inner join are returned in the outer join result and extended with nulls.

## Inner/Outer Table Example

The following example uses the explicit table names inner table and outer table to indicate how these terms relate to the way a simple full outer join is constructed in the FROM clause of a SELECT statement. See [Inner Table and Outer Table](#).

The example shows the semantics of inner and outer table references for a full outer join.

```
outer_table_1 FULL OUTER JOIN outer_table_2
```



Section 1 represents the inner join (intersection) of outer\_table\_1 and outer\_table\_2. Sections 2 and 3 represent the unmatched rows from both outer tables.

The outer join result contains matched rows from the inner join of table\_a with table\_b plus the unmatched rows from table\_a and table\_b.

In terms of the algebra of sets, the result is

$$(Table\_A \cap Table\_B) + (Table\_A - Table\_B) + (Table\_B - Table\_A)$$

where:

$(Table\_A \cap Table\_B)$  is the set of matched rows from the inner join of table\_a and table\_b.

$(Table\_A - Table\_B)$  is the set of unmatched rows from table\_a.

$(Table\_B - Table\_A)$  is the set of unmatched rows from table\_b.

## Practical Example of a Full Outer Join

Suppose you want to find out if the current course offering by customer education satisfied the employee requests for courses, and include the courses offered and their locations, as well as the classes requested (but not offered) and the employees who requested them.

The example uses the same two tables as the prior example, offerings and enrollment.

offerings			enrollment		
course_no	beginning_dates	location		emp_no	course_no
C100	01/05/94	El Segundo		236	C100
C200	07/02/94	Dayton		236	C300
C400	10/07/94	El Segundo		668	C100

Performing a full outer join on these two tables results in the rows from both tables not returned in the result of the inner join being returned in the outer join and extended with nulls.

The following SELECT statement yields the desired results, as indicated in the results table that follows:

```
SELECT offerings.course_no, offerings.location,
       enrollment.course_no, enrollment.emp_no,
FROM offerings
FULL OUTER JOIN enrollment
ON offerings.course_no = enrollment.course_no;
```

o.course_no	o.location	e.course_no	e.emp_no
C100	El Segundo	C100	236
C100	El Segundo	C100	668
C200	Dayton	Null	Null
C400	El Segundo	Null	Null
Null	Null	C300	236

Note that BTEQ reports represent nulls with the QUESTION MARK character.

These results show that course C100 has two employees enrolled and that courses C200 and C400 have no employees enrolled.

In this case, the nulls returned by the outer join of the offerings and enrollment tables are meaningful information.

You can see that course C300, for which employee 236 registered, is not among those courses offered by customer education, nor does it have a location. Notice the null in the last row of the o.course\_no and the o.location columns.

## Multitable Joins

This topic describes some of the operations that are unique to joins of more than two tables.

### Temporary Derived Table

When joining three or more tables, the join process logically creates a temporary derived table that is defined only for the duration of the SELECT operation. Depending on various factors, the Optimizer might rewrite a request in such a way that it generates a query plan that is very different from the 1:1 correspondence between the coded query and its actual implementation that this simplified description implies.

Such a temporary table is also called a joined table. Teradata refers to this kind of temporary table as a pool. Consider this example:

```
table_r
LEFT OUTER JOIN table_s ON join_condition
RIGHT OUTER JOIN table_t ON join_condition
```

When *table\_r* is left outer joined to *table\_s* according to the first join condition, a derived table is created. It is that derived table, not a persistent base table, that is then right outer joined (as a new derived left table) to *table\_t* according to the second join condition.

### ON Clause Evaluation Order

The result of an inner join of two tables is not changed if rows from *table\_a* are joined to rows from *table\_b*, or if rows from *table\_b* are joined to rows from *table\_a*. In terms of set algebra, inner joins are both commutative and associative. This is true no matter how many tables are (inner) joined.

Because inner joins are both commutative and associative, the Optimizer can select the best join order for inner joins arbitrarily and the end result is always identical.

Outer joins, on the other hand, are rarely either commutative or associative. The Optimizer cannot select an arbitrary best join order for outer joins because neither commutativity nor associativity can be assumed, nor does it have any way to know what specific result you intended to produce with the query you presented to it.

To outer join three tables, you must specify their join order explicitly by placing the ON clause in an appropriate position within the FROM clause to ensure that the join is evaluated correctly.

The Optimizer follows these rules to generate outer joins.

- The first ON clause in the query (reading from left to right) is evaluated first.
- Any ON clause applies to its immediately preceding join operation.

### Example: Outer Join

The two following outer join examples evaluate differently because of the order in which their ON clauses are expressed.

For the first example, consider the following outer join.

```
table_r
LEFT OUTER JOIN table_s ON join_condition
RIGHT OUTER JOIN table_t ON join_condition
```

This request is evaluated according to the following steps:

1. *table\_r* is left outer joined to *table\_s* according to the first join condition.
2. The derived table (the table resulting from the first join operation, a newly derived “left” table with respect to what follows) is right outer joined to *table\_t* according to the next join condition.

Using parentheses to indicate order of evaluation, you could rewrite the example in this way:

```
(
table_r
LEFT OUTER JOIN table_s ON join_condition
)
RIGHT OUTER JOIN table_t ON join_condition
```

You can use parentheses to write a SELECT statement that performs an outer join on three or more tables. The second example places the ON clause in another position, providing different result.

```
table_r
LEFT OUTER JOIN table_s
RIGHT JOIN table_t ON join_condition
ON join_condition
```

This statement is evaluated according to the following steps:

1. *table\_s* is right outer joined to *table\_t* according to the first join condition.
2. The derived table (a newly derived “right” table) is left outer joined to *table\_r* according to the next join condition.

Using parentheses to indicate order of evaluation, you could rewrite the example as in the following.

```
table_r
LEFT OUTER JOIN
(
table_s
RIGHT JOIN table_t ON join_condition
)
ON join_condition
```

### Example: Left Outer Join

Suppose you add a fourth table, *table\_u* to the requests of [Example: Outer Join](#), as in the following join operation:

```
table_r
LEFT OUTER JOIN table_s ON join_condition
JOIN table_t
LEFT OUTER JOIN table_u ON join_condition
ON join_condition
```

This statement is evaluated as follows:

1. *table\_r* is left outer joined to *table\_s* according to the first join condition.
2. *table\_t* is then left outer joined to *table\_u* according to the second join condition.
3. The two derived tables are inner joined according to the third join condition.

Using parentheses to indicate order of evaluation, you could rewrite the example as follows:

```
(
table_r
LEFT OUTER JOIN table_s ON join_condition
)
JOIN
```



```
(
  table_t
  LEFT OUTER JOIN table_u ON join_condition
)
ON join_condition
```

Notice that the join condition specified in the ON clause for the inner join (JOIN in the example) is separated from that operation by the join condition that specifies how the left outer join of *table\_t* and *table\_u* is to be performed.

## Coding ON Clauses for Outer Joins

Different join conditions in a FROM clause can yield very different results.

For example, suppose you wish to find out, using the three tables *courses*, *offerings*, and *enrollment*, whether the current course offerings satisfy the employee requests for courses, including the following information in the query.

- Courses offered
- Course locations
- Courses requested but not offered
- Employees who requested courses

### Example: Full Outer Join and Right Outer Join

In the SELECT statement that follows, the *offerings* table is full outer joined to the *enrollment* table according to the first ON condition (*offerings.course\_no* = *enrollment.course\_no*).

```
SELECT courses.course_no, offerings.course_no, offerings.location,
       enrollment.course_no, enrollment.emp_no
FROM offerings
FULL OUTER JOIN enrollment ON offerings.course_no =
                           enrollment.course_no
RIGHT OUTER JOIN courses ON courses.course_no =
                           offerings.course_no;
```

The result of this intermediate join is shown in the following table:

o.course_no	e.course_no	e.emp_no
C100	C100	236
C100	C100	668
C200	?	?
C400	?	?
?	C300	236

Note that BTEQ reports represent nulls with the QUESTION MARK character.

This intermediate derived table is next right outer joined to the *courses* table, according to the second join condition (*courses.course\_no* = *offerings.course\_no*).

The final results appear in the following table:

c.course_no	o.course_no	e.course_no	e.emp_no
C100	C100	C100	236
C100	C100	C100	668
C200	C200	?	?
C400	C400	?	?
C300	?	?	?

### Example: Right Outer Join of Courses and Enrollment

If the same SELECT statement is written with a different second join condition, as illustrated in the current example, then the *offerings* table is full outer joined to the *enrollment* table, according to the first JOIN condition (*offerings.course\_no* = *enrollment.course\_no*).

```
SELECT courses.course_no, offerings.course_no,
       enrollment.course_no, enrollment.emp_no
FROM offerings
FULL OUTER JOIN enrollment ON offerings.course_no =
                           enrollment.course_no
RIGHT OUTER JOIN courses ON courses.course_no =
                           enrollment.course_no;
```

The result of this intermediate join is shown in the following table:

o.course_no	e.course_no	e.emp_no
C100	C100	236
C100	C100	668
C200	?	?
C400	?	?
?	C300	236

Note that BTEQ reports represent nulls with the QUESTION MARK character.

This intermediate derived table is next right outer joined to the *courses* table, according to the second join condition (*courses.course\_no* = *enrollment.course\_no*).

The final results appear in the following table:

c.course_no	o.course_no	e.course_no	e.emp_no
C100	C100	C100	236
C100	C100	C100	668
C300	?	C300	236
C200	?	?	?
C400	?	?	?

### Comparing the ON Clause Join Conditions of the Examples

Although the following second join conditions might intuitively appear to be equivalent, the results of a SELECT are different using the two different join conditions.

Join Condition	Example
<code>courses.course_no = offerings.course_no</code>	<a href="#">Example: Full Outer Join and Right Outer Join</a>
<code>courses.course_no = enrollment.course_no</code>	<a href="#">Example: Right Outer Join of Courses and Enrollment</a>

Using `courses.course_no=offerings.course_no` as the second join condition, you do not see in the results of the outer join, for example, that employee 236 registered for course C300.

But if you use `courses.course_no=enrollment.course_no` as the second join condition, you do not see in the results of the outer join, for example, that courses C200 and C400 are, in fact, offered.

For information on using WHERE clauses with ON clauses, see [Coding ON Clauses With WHERE Clauses for Outer Joins](#).

## Coding ON Clauses With WHERE Clauses for Outer Joins

Although an ON clause is required for each operation in an outer join in a FROM clause, an outer join can itself also include a WHERE clause.

Any restriction in the WHERE clause is applied only to the table that is the final result of the outer join. The WHERE clause does not define the join condition of the outer join.

### Example: Join with WHERE Clause

Consider the following SELECT statement.

```
SELECT offerings.course_no, enrollment.emp_no
FROM offerings
LEFT JOIN enrollment ON offerings.course_no = enrollment.course_no
WHERE location = 'El Segundo';
```

This query yields the intermediate derived results in the following table:

o.course_no	e.emp_no
C100	236
C100	668
C200	?
C400	?

Note that BTEQ reports represent nulls with the QUESTION MARK character.

The query next applies the location restriction. After the restriction is applied (course C200 is given in Atlanta), the results are as seen in the following table:

o.course_no	e.emp_no
C100	236
C100	668
C400	?

### Example: Outer Join with ON Clause

Suppose you were to put the location restriction in the ON clause, as in the following SELECT statement:

```
SELECT offerings.course_no, enrollment.emp_no
FROM offerings
LEFT OUTER JOIN enrollment ON (location = 'El Segundo')
AND (offerings.course_no = enrollment.course_no);
```

Such a restriction is not a join condition. It is a search condition.

The location restriction is applied as part of the inner join (of the left outer join) rather than after the left outer join has been performed.

When formed this way, the query produces confusing results, as the following table illustrates:

o.course_no	e.emp_no
C100	236
C100	668
C200	?
C400	?

Note that BTEQ reports represent nulls with the QUESTION MARK character.

Although it was applied as part of the inner join (of the left outer join), the location restriction did not eliminate course *C200*, given in Atlanta, from being returned. *C200* was returned as part of the rows that were not returned as a result of the inner join.

### Use Join Conditions in ON Clauses, Not Search Conditions

To avoid obtaining unexpected results, you should generally use only ON clauses that reference columns from two tables that are to be joined. In other words, specify join conditions in ON clauses, and not search conditions. Also see [Placing Search Conditions in a Join](#).

### Example: WHERE Restriction Not Part of the Join Condition

Consider the following two tables, *table\_a* and *table\_b*:

<i>table_a.a</i>	<i>table_a.b</i>		<i>table_b.a</i>
3	1		3
6	6		6

The following SELECT statement yields these results. The left outer join returns the data in the table that follows:

```
SELECT *
FROM table_a
LEFT OUTER JOIN table_b ON table_a.a = table_b.a
WHERE table_a.b > 5;
```

The query yields the intermediate derived results in the following table:

<i>table_a.a</i>	<i>table_a.b</i>		<i>table_b.a</i>
	1		3
6	6		6

When the WHERE restriction is applied, the final results are reported as seen in the following table:

<i>table_a.a</i>	<i>table_a.b</i>		<i>table_b.a</i>
6	6		6

### Example: WHERE Restriction Part of the Join Condition

What happens if you include the WHERE restriction as part of the join condition? Consider the following SELECT statement:

```
SELECT *
FROM table_a
```

```
LEFT OUTER JOIN table_b ON (table_a.b > 5)
AND (table_a.a = table_b.a);
```

The results of this query are confusing, as the following results table shows:

table_a.a	table_a.b		table_b.a
6	6		6
3	1		?

Note that BTEQ reports represent nulls with the QUESTION MARK character.

Notice that the join condition, which specified the inner join of the left outer join, restricts the results to 6, 6, 6. But then the rows that were not returned as a result of the inner join are returned and extended with nulls. Thus, for our second row, you get 3, 1, null, which is not the desired result.

### Using Search Conditions In ON Clauses

Some scenarios require a search condition in the ON clause. For example, to list all courses offered as well as the course requested by employee 236, you might use the following query.

```
SELECT offerings.course_no, enrollment.emp_no
FROM offerings
LEFT OUTER JOIN enrollment ON offerings.course_no =
                           enrollment.course_no
AND enrollment.emp_no = 236;
```

### Placing Search Conditions in a Join

The following table provides guidelines for placing the search condition for outer and inner joins:

Join Type	Clause For Search Condition
Outer	WHERE
Inner	ON

In a left outer join, the outer table is the left table, and the inner table is the right table.

### Rules for Coding ON and WHERE Clauses for Outer Joins

The following rules and recommendations apply to coding ON and WHERE clauses for outer joins:

- One or more join conditions, or connecting terms, are required in the ON clause for each table in an outer join.

These join conditions define the rows in the outer table that take part in the match to the inner table.

However, when a search condition is applied to the *inner* table in a WHERE clause, it should be applied in the ON clause as well.

- The best practice is to use *only* join conditions in ON clauses.

A search condition in the ON clause of the inner table does *not* limit the number of rows in the answer set. It defines the rows that are eligible to take part in the match to the outer table.

- An outer join can also include a WHERE clause. However, the results returned with a WHERE clause may not be obvious or intuitive. See [Outer Join Case Study](#) and the topics that follow.
- Geospatial indexes cannot be used for outer joins. For more information on geospatial data, see SQL Geospatial Types.

To limit the number of qualifying rows in the outer table (and therefore the answer set), the search condition for the outer table must be in the WHERE clause. Note that the Optimizer *logically* applies the WHERE clause condition only after a join has been produced. The *actual* application of conditions always depends on how the Optimizer chooses to implement the query.

If a search condition on the inner table is placed in the WHERE clause, the join is logically equivalent to an inner join, even if you explicitly specify the keywords LEFT/RIGHT/FULL OUTER JOIN in the query. The Optimizer always treats such a join as an inner join to simplify the query, rewriting it to roll the entire complex process into a single step.

## Outer Join Case Study

The outer join is a powerful tool, and its output can be as difficult to interpret as it is to produce. A lack of understanding can produce unexpected, costly, and undesired results. For example, you might erroneously mail promotional fliers to 17 million customers instead of the 17,000 customers that you intended to target.

This case study shows some common pitfalls of the outer join, including how a poorly worded outer join request can result in a simple inner join and how an improperly constructed outer join can return millions of rows that do not answer the business question you are trying to ask.

The study also provides guidelines to help you get a feeling for whether you are answering the business question you think you are asking with your outer joins. The study helps you to understand the complexity of outer joins by using realistic examples and explanations.

As this case study shows, many results are possible, and the correct solution is not necessarily intuitive, especially in a more complex query:

- [Case Study Examples](#)
- [Heuristics for Determining a Reasonable Answer Set](#)
- [Guidelines for Using Outer Joins](#)

This case study was originally developed by Rolf Hanusa and is used with his permission.

## Case Study Examples

These examples represent actual cases from a Teradata customer site. The examples are changed slightly to avoid disclosing any business critical information, but the basic syntax and counts remain accurate.

The example EXPLAIN reports are altered slightly for clarity, replacing site-specific aliases with generic database names. The EXPLAIN reports do not reflect the current EXPLAIN text features. This does not detract from their value as an instructional tool.

#### Defining the Business Question

Before writing a complex query, you must understand the business question it is supposed to answer. The following text is a simple explanation of the business question being asked in the case study.

- What are the customer numbers for all the customers in the *customer* table (which contains over 18 million rows)
- AND
- Who resides in *district* K
- AND
- Who has a *service\_type* of ABC
- OR
- Who has a *service\_type* of XYZ
- AND
- What is their *monthly\_revenue* for July 1997 (using the *revenue* table, which contains over 234 million rows)
- AND
- If the *monthly\_revenue* for a customer is unknown (no revenue rows are found), then report the customer number (from *customer.cust\_num*) with a null (represented by a QUESTION MARK (?) character) to represent its *monthly\_revenue*.

This sounds simple, but when you analyze your answer sets carefully, you might find them to be incorrect. As the following topic demonstrates, the results are correct, but the outer join queries that produce them are not asking the proper questions. This becomes more apparent as the examples are developed and the results analyzed.

The study presents two example queries for determining the reasonableness of the answers of this study, three missteps along the path to getting the correct answer, and a properly coded query that returns the correct answer for the original question that was asked.

## Heuristics for Determining a Reasonable Answer Set

Before you can judge the accuracy of the results returned from an outer join query, you need to estimate a reasonable guess about the likelihood that an answer set is correct. This section provides two simple methods for obtaining the approximate numbers of rows you should expect from a correctly structured outer join query:

- Example 1: A single table SELECT of an outer join used for this study.
- Example 2: An inner join that explains the remaining queries and results in the case study. It starts with the same base of customer rows but matches them with revenue rows for a particular month.



## Single Table Cardinality Estimate

This query is a single table SELECT that provides the cardinality of the customer table as restricted by the conditions used for the example outer join queries.

From the report returned by this query, we know how many rows should be returned by the outer join.

```
SELECT c.cust_num
FROM sampdb.customer c
WHERE c.district = 'K'
AND (c.service_type = 'ABC'
OR c.service_type = 'XYZ')
ORDER BY 1;
```

This query returns 18,034 rows.

## Inner Join Cardinality Estimate

This query is an inner join that helps to explain the example outer join queries and their results. The query starts with the same customer rows found in [Single Table Cardinality Estimate](#), but then matches them with revenue rows for the month of July, 1997.

```
SELECT c.custnum, b.monthly_revenue
FROM sampdb.customer AS c, sampdb2.revenue AS b
WHERE c.custnum = b.custnum
AND c.district = 'K'
AND b.data_year_month = 199707
AND (c.service_type = 'ABC'
OR c.service_type = 'XYZ')
ORDER BY 1;
```

The query returns 13,010 rows.

Note that all *customer* table rows are matched by a *monthly\_revenue* table row.

## Example 1: Outer Join That Is Really an Inner Join

This example 1 makes an explicit outer join request, but its result might surprise you.

```
SELECT c.custnum, b.monthly_revenue
FROM sampdb.customer AS c
LEFT OUTER JOIN sampdb2.revenue AS b ON c.custnum = b.custnum
WHERE c.district = 'K'
AND b.data_date = 199707
AND (c.service_type = 'ABC'
OR c.service_type = 'XYZ')
ORDER BY 1;
```

This query returns 13,010 rows, the same as the simple inner join of [Inner Join Cardinality Estimate](#).

## Analysis of the Result

Although the request specifies a LEFT OUTER JOIN, the Optimizer treats it as a simple inner join because all the selection criteria are in the WHERE clause, so they are logically applied only *after* the outer join processing has been completed. Examples 1 and 3 are logically identical, so they produce the same result.

An EXPLAIN shows that the Optimizer recognizes this query to be an *inner* join and executes it as such (“...joined using a merge join...” rather than “...*left outer* joined using a merge join...”). Therefore, it executes with the speed of an inner join.

The Optimizer selects the best join algorithm for this *inner* join, a merge join, and applies the conditions from the WHERE clause. For information about merge join, see *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

Logically, the WHERE clause terms are applied after the terms in the ON clause of the LEFT OUTER JOIN have been applied, which results in an intermediate step of 18,034 rows. However, the WHERE clause includes `B.DATA_YEAR_MONTH=199707`, eliminating all rows where the value for `B.DATA_YEAR_MONTH` is null and returning the incorrect result of 13,010 rows.

The Optimizer recognizes that the WHERE clause references an inner table that does not evaluate to TRUE for nulls and is therefore equivalent to an inner join. As a result, it generates a plan for a simple inner join, ignoring the request for a left outer join.

The EXPLAIN shows “...SAMPDB.CUSTOMER and SAMPDB2.REVENUE are joined using a merge join...” Had the plan used an outer join, the EXPLAIN text would have said “...SAMPDB.CUSTOMER and SAMPDB2.REVENUE are left outer joined using a merge join...”

The result of the query, 13,010 rows, is not the correct answer to our business question.

## The Coding Error

The SELECT statement errs by placing the outer table in the query (*customer*) in the WHERE clause rather than the ON clause, resulting in the query being processed as a simple inner join rather than the intended left outer join.

You can examine this from two different perspectives, both of which explain why the expected result was not returned.

## Logical explanation

The WHERE clause appears later in the query than the ON clause. Because of this, the WHERE clause predicates are evaluated after the ON clause predicates.

The result table for the ON clause has 18,034 rows. When the WHERE clause predicate `b.data_year_month=199707` is applied, you should expect the ensuing result table to have fewer rows because this condition eliminates any results rows where `b.data_year_month` is null, and an outer join, by definition, produces results nulls on selection predicates.

## Physical explanation

The EXPLAIN text for this query does not follow the logical explanation.

One of the fundamental tasks of the Optimizer is to reorganize queries into an algebraic form that can be analyzed quickly. The Optimizer recognizes that a predicate referencing the inner table of an outer join does not evaluate to TRUE when a column referenced in that predicate expression contains nulls.

The Optimizer recognizes the join to be an inner join, not an outer join. As a result, it generates a plan to perform an inner join. The EXPLAIN includes "...are joined using a merge join..." The wording would have been "...are joined using an *outer* merge join..." if the query had been formulated correctly as an outer join.

While this is the correct interpretation of the SELECT statement as written, the query does not formulate the business question correctly.

## Example 2: Outer Join But Does Not Return the Correct Answer

Example 2 is a correctly formed outer join, but it neither addresses the question posed nor returns the correct answer.

```
SELECT c.custnum, b.monthly_revenue
FROM sampdb.customer AS c
LEFT OUTER JOIN sampdb2.revenue AS b ON c.custnum = b.custnum
AND c.district = 'K'
AND b.data_year_month = 199707
AND (c.service_type = 'ABC'
OR c.service_type = 'XYZ')
ORDER BY 1;
```

This query returns 17713502 rows.

## Analysis of the Result

The query specifies a left outer join without qualifying the results table with a WHERE clause. The Optimizer constructs a plan for a left outer join as specified by the query. The result is a join of every row in the outer table (17,713,502 rows) with the qualified rows of the inner table, those service types of either ABC or XYZ in district K in the month of July, 1997 (13,010 rows).

The cardinality of this result, 17,713,502 rows, is three orders of magnitude larger than the correct answer to the business question.

Without a WHERE clause to qualify the result, an outer join result always contains at least 1 row for every row in the outer table.

Qualifying predicates placed in the ON clause do not eliminate rows from the result. Rather, they are treated as rows that are not matched with the inner table irrespective of whether join terms that reference both the inner and outer tables evaluate to TRUE. In other words, the selection criteria of an ON clause only define the rows to which nulls are to be appended for nonmatching rows. They do not restrict the result to rows matching the conditions they specify.

The Optimizer does qualify the result to a degree, returning only those rows for which the predicate `b.data_year_month = 199707` evaluates to TRUE.

The Optimizer selects the best join algorithm for this left outer join (a merge join) and applies the conditions from the ON clause.

The result of the query, 17,713,502 rows, which is not the correct answer to our business question.

## The Coding Error

The SELECT statement errs by placing all of the selection criteria for the query in the ON clause. The ON clause should only include selection criteria for the *inner* table in the outer join, that is, the criteria that define the nullable nonmatching rows for the outer join.

Selection criteria for the *outer* table (the *customer* table in this case) in the outer join must be placed in the WHERE clause.

## Example 3: Outer Join That Is Really an Inner Join

This example is another query where an explicit outer join is specified, but that join is logically an inner join, so the Optimizer transforms it into a simple inner join before performing the query.

```
SELECT c.custnum, b.monthly_revenue
FROM sampdb.customer c
LEFT OUTER JOIN sampdb2.revenue b ON c.custnum = b.custnum
AND c.district = 'K'
AND (c.service_type = 'ABC'
OR c.service_type = 'XYZ')
WHERE b.data_year_month = 199707
ORDER BY 1;
```

This query returns 13,010 rows.

## Analysis of the Result

This example is similar to Example 1. The Optimizer treats this query as an inner join even though the request explicitly specified an outer join. The WHERE clause on the inner table logically changes this query from an outer join to an inner join.

As in previous examples, the WHERE clause is logically applied after the outer join processing completes, removing all rows that were nulled in the process: nonmatching rows between the left and right tables. The Optimizer knows to execute this as an inner join to improve the performance of the query.

The EXPLAIN text matches the EXPLAIN text for Example 1. As expected, the answer set also matches.

The Optimizer selects the best join algorithm for this *inner* join (a merge join) and applies the conditions from the WHERE clause.

Logically, the WHERE clause term is applied after the terms in the ON clause of the LEFT OUTER JOIN have been applied. That would result in an intermediate step of 18,034 rows. However, the term

`b.data_year_month=199707` from the WHERE clause would then be applied, eliminating all rows where the value for `b.data_year_month` is null, and returning the final, incorrect, result of 13,010 rows.

The Optimizer recognizes that the WHERE clause references an inner table that does not evaluate to TRUE for nulls and is therefore equivalent to an inner join. As a result, it generates a plan for a simple inner join, ignoring the request for a left outer join.

The EXPLAIN text says “...SAMPDB.CUSTOMER and SAMPDB2.REVENUE are joined using a merge join...” Had the plan used an outer join, the EXPLAIN text would have said “...SAMPDB.CUSTOMER and SAMPDB2.REVENUE are left outer joined using a merge join...”

The result of the query, 13,010 rows, which is not the correct answer to our business question.

### The Coding Error

The SELECT statement errs by placing:

```
b.data_date = 199707
```

in the WHERE clause rather than the ON clause, resulting in the query being processed as a simple inner join rather than the left outer join as intended.

### Example 4: Outer Join, the Correct Answer

Finally, we have the correct answer. This example is an outer join that provides the desired answer to the original business question.

```
SELECT c.custnum, b.monthly_revenue
FROM sampdb.customer AS c
LEFT OUTER JOIN sampdb2.revenue AS b ON c.custnum = b.custnum
AND   b.data_year_month = 199707
WHERE c.district = 'K'
AND   (c.service_type = 'ABC'
OR     c.service_type = 'XYZ')
ORDER BY 1;
```

This query returns 18,034 rows.

### Analysis of the Result

The cardinality of this query result, 18,034 rows, reconciles with the expected number of returned rows. See [Single Table Cardinality Estimate](#).

13,010 rows have values (non-nulls) for *monthly\_revenue*.

The EXPLAIN shows that the system performs a left outer join.

The Optimizer selects the best join algorithm for this outer join (a Merge Join) and applies the conditions from the WHERE and ON clauses, respectively.

The left (outer) table is limited by the search conditions in the WHERE clause, and the search condition in the ON clause for the right (inner) table defines the nullable nonmatching rows.

The EXPLAIN text confirms that this is a true outer join (“...SAMPDB.c and SAMPDB2.b are left outer joined...”).

The result of the query, 18,034 rows, the correct answer to our business question.

## Guidelines for Using Outer Joins

Outer joins, when used properly, provide additional information from a single query that would otherwise require multiple queries and steps to achieve. However, the proper use of outer joins requires training and experience because “common sense” reasoning does not always apply to formulating an outer join query in a manner that is not only syntactically correct, but also returns an answer that is correct for the business question.

To ensure the correct answers to your business questions using outer joins, refer to these steps.

1. Understand the question you are trying to answer and know the demographics of your tables.

Form a hypothesis of the answer set you expect before you begin. See [Single Table Cardinality Estimate](#) and [Inner Join Cardinality Estimate](#).

2. Write the query, keeping in mind the proper placement of join and search conditions.

Condition	Clause
Join.	ON
Search condition predicates for inner table.	ON
Search condition predicates for outer table	WHERE

3. Always EXPLAIN the query before performing it.

Look for the words outer join in the EXPLAIN text. If the EXPLAIN text does not include outer join, the Optimizer did not produce an outer join plan.

4. Perform the query and compare the result with your expectations.

Answer Set	Description
Matches your expectations.	Answer set is probably correct.
Does not match your expectations.	Check the placement of the selection criteria predicates in the ON and WHERE clauses of your outer join.

For information about the various types of join methods the Optimizer uses in responding to requests, see “Join Planning and Optimization” in *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

# Statement Syntax

## Overview

These topics describe the SQL data manipulation language (DML) statements, except for these types:

- The forms of DML statements that apply to temporal tables. See *Teradata Vantage™ ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ Temporal Table Support*, B035-1182.
- SELECT, see [SELECT](#).
- Query and workload analysis DML statements, see [Query and Workload Analysis Statements](#).
- DML statements used for:
  - declaring and manipulating cursors
  - multisession programming
  - client-server connectivity
  - dynamic SQL
  - embedded SQL only
  - stored procedures only

See *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148.

## Statement Independence Support

Certain forms of Teradata multistatement DML requests support statement independence. Statement independence enables multistatement requests and iterated INSERT statements to roll back only those statements within the transaction or multistatement request that fail.

---

### Note:

The client software you are using must support statement independence to prevent all multistatement requests and iterated INSERT statements in a transaction or multistatement request from being rolled back. Refer to the documentation for your client software for information on support for statement independence.

---

These forms of multistatement requests support statement independence:

- INSERT; INSERT; INSERT;
- BT; INSERT; INSERT; ET;
- BT; INSERT; INSERT;
- INSERT; INSERT; ET;
- INSERT; COMMIT;
- INSERT; INSERT; COMMIT;

## Related Topics

For information about the INSERT statement, see [INSERT/INSERT ... SELECT](#).

For the rules in regard to using multistatement and iterated requests, see [Multistatement and Iterated INSERT Requests](#).

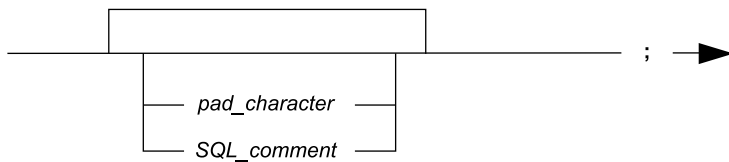
For more information about how transactions are processed and statement independence, see *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

## Null

### Purpose

Provides a method for including a blank line or self-contained comment within an SQL request, such as a multistatement request, macro, stored procedure, and so on.

### Syntax



### Syntax Elements

#### *pad\_character*

Valid pad character, such as a SPACE character.

#### *SQL\_comment*

Valid SQL comment string.

### ANSI Compliance

Null statements, although closely related to the ANSI SQL concept of separators, are a Teradata extension to the ANSI SQL:2011 standard.

### Required Privileges

None.

### Invocation

Nonexecutable.

### Definition: Null Statement

A statement that has no content except for optional pad characters or SQL comments.



**Example: Null Statement**

The semicolon in the following request is a null statement.

```
/* This example shows a comment followed by
   a semicolon used as a null statement */
; UPDATE Pay_Test SET ...
```

**Example: Null Statement and Statement Separator**

The first SEMICOLON in the following request is a null statement. The second SEMICOLON is taken as statement separator.

```
/* This example shows a semicolon used as a null
   statement and as a statement separator */
; UPDATE Payroll_Test SET Name = 'Wedgewood A'
   WHERE Name = 'Wedgewood A'
; SELECT ...
-- This example shows the use of an ANSI component
-- used as a null statement and statement separator ;
```

**Example: Single Null Statement**

A SEMICOLON that precedes the first or only statement of a request is taken as a null statement.

```
; DROP TABLE temp_payroll;
```

## ABORT

**Purpose**

Terminates the current transaction and rolls back its updates.

For information about the temporal form of ABORT, see *Teradata Vantage™ Temporal Table Support*, B035-1182.

**Required Privileges**

None.

**Syntax**

```
ABORT —┬─ abort_message ┬─ FROM option ┬─ WHERE abort_condition ──▶
```

## Syntax Elements

### ***abort\_message***

Text of the message to be returned when the transaction is terminated.

If *abort\_message* is not specified, the message defaults to “user-generated transaction ABORT”.

### **FROM option**

A clause that is required only if the WHERE clause includes subqueries or if the request is preceded with a temporal qualifier.

You can code scalar subqueries as an expression within a derived table in FROM clause conditions in the same way you can specify them for a SELECT request.

See [Scalar Subqueries](#) and [Rules for Using Scalar Subqueries in ABORT Statements](#). Note, however, that you *cannot* code a derived table as a scalar subquery.

The contents of the FROM option are described in [FROM Clause](#).

### **WHERE *abort\_condition***

A clause that introduces a conditional expression whose result must evaluate to TRUE if the transaction is to be rolled back. The expression can specify aggregate operations and scalar subqueries.

If the WHERE clause is omitted, termination is unconditional. See [WHERE Clause](#).

If you specify the value for a row-level security constraint, it must be expressed in its encoded form.

The WHERE condition specifies an expression whose result must evaluate to TRUE if termination is to occur. If the result is FALSE, transaction processing continues.

## ANSI Compliance

ABORT is a Teradata extension to the ANSI SQL:2011 standard. It is a synonym for the ANSI SQL:2011-compliant statement ROLLBACK (see [ROLLBACK](#)).

## Usage Notes

### **Definition and Termination of ANSI Transactions**

In ANSI session mode, the first SQL request in a session initiates a transaction. The transaction is terminated by issuing either a COMMIT or a ABORT/ROLLBACK statement. Request failures do not cause a rollback of the transaction, only of the request that causes them.

### **ABORT Is Explicit**

There are no implicit transactions in ANSI session mode. More accurately, each ANSI transaction is initiated implicitly, but always completed explicitly. A COMMIT or ABORT/ROLLBACK must always be explicitly stated and be the last statement of a transaction in order for the transaction to terminate successfully.

In ANSI session mode, you must issue an ABORT/ROLLBACK (or COMMIT) even when the only statement in a transaction is a SELECT or SELECT AND CONSUME.

For a SELECT statement, there is no difference between issuing a COMMIT or an ABORT/ROLLBACK.

In the case of a SELECT AND CONSUME, there is a difference in the outcomes between issuing a COMMIT or ABORT/ROLLBACK because an ABORT or ROLLBACK statement reinstates the subject queue table of the statement to its former status, containing the rows that were pseudo-consumed by the aborted SELECT AND CONSUME statement.

## ABORT and ROLLBACK Are Synonyms

With the exception of the absence of the WORK keyword, ABORT is a Teradata synonym for the ANSI-compliant ROLLBACK statement.

## Actions Performed by ABORT

ABORT performs the following actions:

1. Backs out changes made to the database as a result of the transaction.
2. Deletes spooled output for the request.
3. Releases locks associated with the transaction.
4. If the transaction is in the form of a macro or a multistatement request, bypasses execution of the remaining statements.
5. Returns a failure response to the user.

## Actions Performed by ABORT With Embedded SQL

ABORT performs the following additional actions when used within an embedded SQL application:

1. See [Actions Performed by ROLLBACK](#).
2. Discards dynamic SQL statements prepared within the current transaction.
3. Closes open cursors.
4. Cancels database changes made by the current transaction.
5. Sets SQLCODE to zero when ROLLBACK is successful.
6. Sets the first SQLERRD element of the SQLCA to 3513 or 3514, depending on whether *abort\_message* is specified.
7. Returns an abort message in the SQLERRM field of the SQLCA if you specify a WHERE clause.
8. Terminates the application program connection (whether explicit or implicit) to Teradata Database if the RELEASE keyword is specified.

Environment	Preprocessor Action when LOGON/CONNECT Request Does Not Precede the Next SQL Request
Mainframe-attached	Attempts to establish an implicit connection.
Workstation-attached	Issues a No Session Established error.

## ABORT With a WHERE Clause

ABORT tests each value separately. Therefore, the WHERE clause should not introduce both an aggregate and a nonaggregate value. The aggregate value becomes, in effect, a GROUP BY value, and the mathematical computation is performed on the grouped aggregate results.

For example, assuming that the following items are true, the ABORT statement that follows incorrectly terminates the transaction.

- The table test contains several rows,
- The sum of test.colA is 188, and
- Only one row contains the value 125 in test.colB

```
ABORT WHERE (SUM(test.colA) <> 188)
          AND (test.colb = 125);
```

The preceding statement is processed first by performing an all-rows scan with the condition (colb=125), which selects a single row and then computes intermediate aggregate results with the condition (SUM(colA) <> 188).

The condition tests true because the value of cola in the selected row is less than 188.

If ABORT ... WHERE is used and the statement requires READ access to an object for execution, the user executing this DML statement must have SELECT right to the data being accessed.

The WHERE clause search condition of an ABORT statement can include scalar subqueries. If so, the subqueries require a FROM clause, and the ABORT should have a FROM clause if it is desired that the scope of a reference in a subquery is the ABORT condition.

## ABORT with a UDT in Its WHERE Clause

ABORT supports comparisons of UDT expressions in a WHERE clause. A UDT expression is any expression that returns a UDT value.

If you specify a UDT comparison, then the UDT must have a defined ordering. See “CREATE ORDERING” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

## Rules for Using Correlated Subqueries in an ABORT Statement

The following rules apply to correlated subqueries used in an ABORT statement:

- A FROM clause is required for an ABORT statement if the ABORT references a table. All tables referenced in an ABORT statement must be defined in a FROM clause.
- If an inner query column specification references an outer FROM clause table, then the column reference must be fully qualified.
- Teradata Database supports the ABORT statement as an extension to ANSI syntax. The equivalent ANSI SQL statement is ROLLBACK. See [ROLLBACK](#).

Correlated subqueries, scalar subqueries, and the EXISTS predicate are supported in the WHERE clause of an ABORT statement.

See [Correlated Subqueries](#).

### Rules for Using Scalar Subqueries in ABORT Statements

You can specify a scalar subquery in the WHERE clause of an ABORT statement in the same way you can specify one for a SELECT statement.

You can also specify an ABORT statement with a scalar subquery in the body of a trigger. However, Teradata Database processes any noncorrelated scalar subquery you specify in the WHERE clause of an ABORT statement in a row trigger as a single-column single-row spool instead of as a parameterized value.

### Rules for Using a Scalar UDF in an ABORT Statement

You can invoke a scalar UDF in the WHERE clause of a UDF if the UDF returns a value expression. However, you can only specify a scalar UDF as a search condition if it is invoked within an expression and returns a value expression.

### Rules For Using ABORT In Embedded SQL

The following rules apply to using ABORT within an embedded SQL program.

- ABORT cannot be performed as a dynamic SQL request.
- ABORT is not valid when you specify the TRANSACT(2PC) option to the embedded SQL preprocessor.

### Multiple ABORT Statements

If a macro or multistatement request contains multiple ABORT statements, those statements are initiated in the order they are specified, even if the expressions could be evaluated immediately by the parser because the request does not access any tables. However, the system can process ABORT statements in parallel.

If the system executes the ABORT statements in parallel, and one of the statements actually does abort, then the system reports that abort regardless of its specified sequence in the macro or multistatement request.

You can examine the EXPLAIN report for the macro or multistatement request to determine whether the system is executing the ABORT statements in parallel or not.

### Two Types of ABORT Statements

There are two categories of ABORT statements, those that can be evaluated by the Parser and do not require access to a table and those that require table access.

If ABORT expressions are processed that do not reference tables, and if their order of execution is not important relative to the other requests in a multistatement request or macro, then they should be placed ahead of any statements that reference tables so that the abort statement can be done at minimum cost.

In the following example, the first two ABORT statements can be evaluated by the parser and do not require access to tables. The third ABORT statement requires access to a table.

```
CREATE MACRO macro_name (
    P1 INTEGER,
    P2 INTEGER)
AS. . .
ABORT 'error' WHERE :p1 < 0;
ABORT 'error' WHERE :p2 < 0;
SELECT. . .
ABORT 'error' WHERE tab.c1 = :p1;
```

## Examples

### Example: ABORT Statement

In the following example, the ABORT statement terminates macro execution if the row for the employee being deleted is not present in the *employee* table.

Statements in the body of the macro are entered as one multistatement request. Therefore, if the WHERE condition of the ABORT statement is met, the entire request is aborted and the accuracy of the count in the *department* table is preserved.

```
CREATE MACRO del_emp (num SMALLINT FORMAT '9(5)',
    dname VARCHAR(12),
    dept SMALLINT FORMAT '999') AS (
    ABORT 'Name does not exist' WHERE :num NOT IN (
    SELECT emp_no
    FROM employee
    WHERE UPPER(name) = UPPER(:dname))
;DELETE FROM employee
    WHERE UPPER(name) = UPPER(:dname)
;UPDATE department
    SET emp_count = emp_count - 1
    WHERE dept_no = :dept; ) ;
```

### Example: Valid ABORT Statements

The following ABORT statements are all syntactically correct.

```
ABORT
WHERE user= 'DBC';

ABORT
FROM table_1
WHERE table_1.x1 = 1;

ABORT
FROM table_11
```

```
WHERE table_1.x1 > 1;

ABORT
FROM table_1,table_2
WHERE table_1.x1 = table_2.x2;
```

### Example: ABORT Statement With WHERE Clause

The following example uses an ABORT statement in the macro *newemp*. *newemp* inserts a row into the *employee* table for each new employee and then executes the SELECT statement to verify that the new information was entered correctly. The ABORT statement ensures that no new employee is inadvertently added to the executive office department, department 300.

```
CREATE MACRO newemp (
    number    INTEGER,
    name      VARCHAR(12),
    dept      INTEGER 100 TO 900,
    position  VARCHAR(12),
    sex       CHARACTER,
    ed_lev    BYTEINT ) AS (
ABORT 'Department number 300 not valid'
WHERE :dept = 300 ;
INSERT INTO employee (emp_no, name, dept_no, job_title, sex,
                      ed_lev)
VALUES (:number, :name, :dept, :position, :sex, :edlev) ;
SELECT *
FROM employee
WHERE emp_no = :number ; ) ;
```

### Example: ABORT With a UDT In The WHERE Clause

The following examples show correct use of UDT expressions in the WHERE clause of an ABORT statement:

```
ABORT WHERE (tab1.euro_col < CAST(0.0 AS euro));

ABORT WHERE (tab1.cir_col.area() < 10.0);
```

### Example: Using an SQL UDF in the WHERE Clause of an ABORT Statement

The following ABORT statement specifies an SQL UDF in its WHERE search condition.

```
ABORT FROM t1
WHERE a1 = test.value_expression(2, 3);
```

## Related Topics

- [ROLLBACK](#)
- [COMMIT](#)
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ Temporal Table Support*, B035-1182

## BEGIN TRANSACTION

### Purpose

Defines the beginning of an explicit logical transaction in Teradata session mode.

An explicit Teradata session mode transaction must always be terminated with an END TRANSACTION statement. You must always specify both BEGIN TRANSACTION and END TRANSACTION statements to define the limits of an explicit transaction in Teradata session mode.

### Syntax

BEGIN TRANSACTION —————▶  
BT —————▶

### ANSI Compliance

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

The statement is valid only in Teradata session mode. If you submit a BEGIN TRANSACTION statement in an ANSI mode session, the system returns an error.

For ANSI session mode transaction control statements, see [COMMIT](#) and [ROLLBACK](#).

Other SQL dialects support similar non-ANSI standard statements with names such as the following:

- BEGIN
- BEGIN WORK

### Required Privileges

None.

## Usage Notes

### Rules for Embedded SQL

The following rules apply to the use of BEGIN TRANSACTION in embedded SQL:



- `BEGIN TRANSACTION` is valid only when you specify the `TRANSACT(BTET)` or `-tr(BTET)` options to the preprocessor. Otherwise, an error is returned and the precompilation fails.
- `BEGIN TRANSACTION` cannot be performed as a dynamic SQL statement.

## Explicit Transactions

An explicit, or user-generated, transaction is a single set of `BEGIN TRANSACTION`/`END TRANSACTION` statements surrounding one or more additional SQL statements.

A `BTEQ .LOGOFF` command following a `BEGIN TRANSACTION` statement and one or more additional statements also forms an explicit transaction as well as terminating the current session.

A `.LOGOFF` command aborts the transaction and rolls back the processing results of all the statements that occurred between `BEGIN TRANSACTION` and `.LOGOFF`.

All other Teradata session mode transactions are implicit.

## Implicit Transactions

An implicit, or system-generated, transaction, is typically one of the following:

- A macro.
- A single-statement request.
- A multistatement request that is not part of an explicit transaction.

## Rules for Transactions Containing DDL Statements

A transaction can be any solitary SQL statement, including a DDL statement.

When a request contains multiple statements, a DDL statement is only allowed in the following situations:

- The transaction is explicit, that is, bracketed by `BEGIN TRANSACTION` and `END TRANSACTION` statements.
- The DDL statement is the last statement in the transaction, that is, immediately followed by the `END TRANSACTION` statement.

## Teradata Database Transaction Handling Protocol

Teradata Database manages transactions to maintain valid, consistent, and available data for all users. A transaction may consist of one or more requests which are sent one at a time and processed as each is received. Locks are applied as needed for each request. Various locks are placed on its associated database objects according to the types of statements contained in the request.

If a statement does not complete successfully or causes processing to time out for some reason, such as a statement error, deadlock, privilege violation, table constraint violation, or premature logoff, the system performs the following transaction management steps.

1. The entire transaction is aborted. Abort processing performs the following actions.
  - a. Performs an implicit END TRANSACTION.
  - b. Backs out any changes made to the database by the transaction.
  - c. Releases any usage locks associated with the transaction.
  - d. Discards any partially accumulated results (spools) generated by the transaction.
2. A failure or time-out response is returned to the requestor.

Most statement errors resulting from multistatement INSERT statements abort the entire transaction.

Statement independence is a method of handling multistatement INSERT errors by only rolling back the individual statements that failed, not the entire transaction. See [Multistatement and Iterated INSERT Requests](#).

---

**Note:**

The client software you are using must support statement independence to prevent all multistatement requests and iterated INSERT statements in a transaction or multistatement request from being rolled back. Refer to the documentation for your client software for information on support for statement independence.

---

## Specifying SQL Request Modifiers With Explicit Transactions

When the Parser receives a BEGIN TRANSACTION statement, it immediately looks for an SQL statement keyword in the SQL text that follows. Consider this when determining the placement of request modifiers such as EXPLAIN, LOCKING, NONTEMPORAL, and USING.

For example, if the first statement in an explicit transaction is associated with a USING request modifier, that USING request modifier must precede the BEGIN TRANSACTION statement.

## Nested BEGIN TRANSACTION/END TRANSACTION Pairs

When BEGIN TRANSACTION/ET pairs are nested, Teradata Database checks to ensure that each BEGIN TRANSACTION statement has a matching END TRANSACTION statement.

The outermost BEGIN TRANSACTION/END TRANSACTION pair defines the explicit transaction. The inner BEGIN TRANSACTION/END TRANSACTION pairs do not effect the transaction because Teradata Database does not support transaction nesting.

Any embedded multistatement requests and macro executions are considered part of the outermost BEGIN TRANSACTION/END TRANSACTION explicit transaction and are not considered to be implicit transactions in this context.

In a multistatement request, there can be no more than one BEGIN TRANSACTION statement, and if you do specify a BEGIN TRANSACTION statement, it must be the first statement in a request.

For example, the following series of requests is treated as one explicit transaction.

```
BEGIN TRANSACTION;
  SELECT ...;
  UPDATE ...
  EXEC a(3,4);
  BEGIN TRANSACTION;
    UPDATE ...;
    INSERT ...
    ;INSERT ...;
  END TRANSACTION;
  INSERT ...;
END TRANSACTION;
```

If an error occurs in the middle of a nested BEGIN TRANSACTION/END TRANSACTION, everything rolls back to the initial BEGIN TRANSACTION.

## Scenarios

The following scenarios illustrate the use of BEGIN TRANSACTION/END TRANSACTION.

### Scenario: Explicit Transaction to Delete

Assuming that the *department* table contains an *emp\_count* column, the following explicit transaction could be used to remove a row from the *employee* table and then decrement a departmental head count in the *department* table.

```
BEGIN TRANSACTION;
  DELETE FROM employee
  WHERE name = 'Reed C';
  UPDATE department
  SET emp_count = emp_count -1
  WHERE dept_no = 500;
END TRANSACTION;
```

### Scenario: Explicit Transaction to Insert

The following example illustrates an explicit transaction in which each INSERT statement is associated with a USING request modifier.

```
BEGIN TRANSACTION ;
  USING ssnumfile (INTEGER)
  INSERT INTO employee (soc_sec_no) VALUES (:ssnumfile) ;
  USING ssnumfile (INTEGER)
```

```

INSERT INTO employee (soc_sec_no) VALUES (:ssnumfile) ;
    USING ssnumfile (INTEGER)
INSERT INTO employee (soc_sec_no) VALUES (:ssnumfile) ;
END TRANSACTION ;

```

### Scenario: Explicit Transaction for a DDL Statement

The following examples illustrate the use a DDL statement in an explicit transaction. These transactions create a volatile table, perform an aggregate operation on the result of another aggregate operation, and then drop the volatile table that was created in the first transaction.

Two transactions are used because a DDL statement must be either the only statement in a transaction or the last statement in a transaction

```

BEGIN TRANSACTION;
CREATE VOLATILE TABLE dept_sum_sal NO LOG (
    dept_no SMALLINT FORMAT '999' BETWEEN 100 AND 900 NOT NULL,
    sum_sal DECIMAL(8,2) FORMAT 'ZZZ,ZZ9.99')
PRIMARY INDEX(dept_no),
ON COMMIT DELETE ROWS;
END TRANSACTION;

BEGIN TRANSACTION;
INSERT INTO dept_sum_sal
SELECT dept_no, SUM(salary)
FROM employee
GROUP BY dept_no;
SELECT AVG(sum_sal)
FROM dept_sum_sal;
DROP VOLATILE TABLE dept_sum_sal;
END TRANSACTION;

```

### Scenario: Implicit Transaction (BTEQ)

The following example is structured as a BTEQ multistatement request, so it is processed as a single implicit transaction.

Note the placement of the USING modifier and the semicolons. With this construct, the failure of any WHERE conditions causes the transaction to abort and all completed insert and update operations to be rolled back.

```

USING var1(CHARACTER),
      var2(CHARACTER),
      var3(CHARACTER)
INSERT INTO test_tab_u (c1) VALUES (:var1)
; INSERT INTO test_tab_u (c1) VALUES (:var2)
; INSERT INTO test_tab_u (c1) VALUES (:var3)

```

```

; UPDATE test_tab_u SET c2 = c1 + 1
  WHERE c1 = :var1
; UPDATE test_tab_u SET c2 = c1 + 1
  WHERE c1 = :var2

```

## Related Topics

- [ABORT](#)
- [END TRANSACTION.](#)
- [COMMIT](#)
- [ROLLBACK](#)

## CALL

### Purpose

Invokes an SQL procedure or external stored procedure.

### Required Privileges

You must have the EXECUTE PROCEDURE privilege on the procedure or the containing database or user.

You do not require any privileges on the database objects referenced by a called procedure. The privileges are checked for the immediate owner of the called procedure.

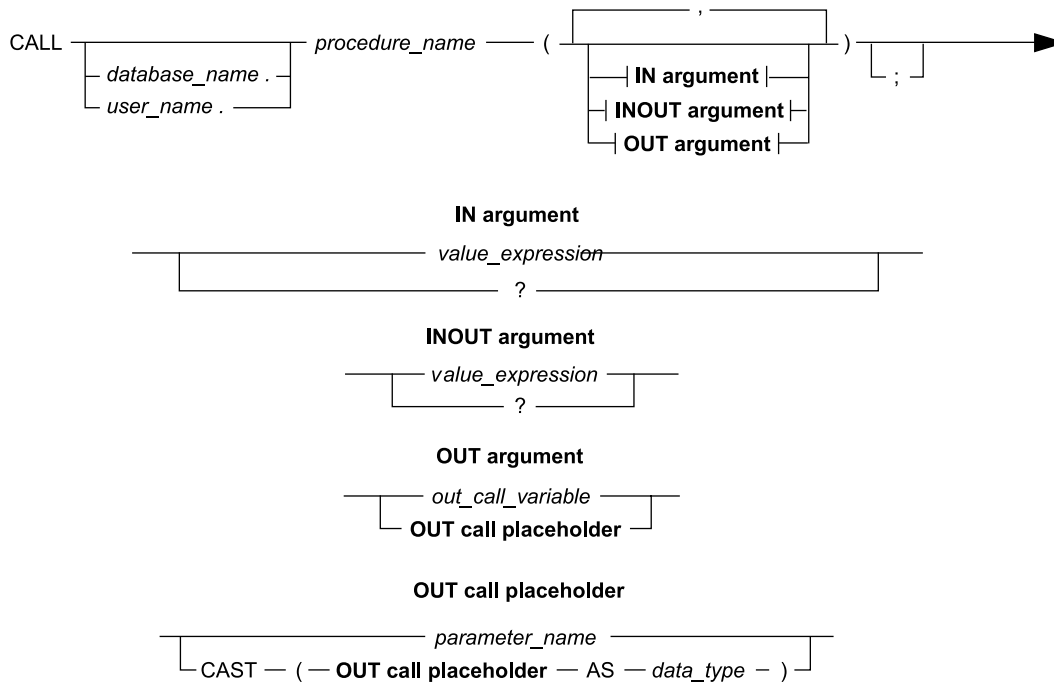
The immediate owner must have the privileges on the referenced objects WITH GRANT OPTION to allow other users to access the object.

For more information, see *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184 and *Teradata Vantage™ - Database Administration*, B035-1093.

### Privileges Granted Automatically

If you call the SQLJ.Install\_Jar procedure to install a JAR file containing classes that implement a Java external stored procedure, the DROP PROCEDURE privilege is automatically granted to the creator of the JAR file, that is, the caller of SQLJ.Install\_Jar.

## Syntax



## Syntax Elements

***database\_name***

***user\_name***

Optional qualifier for the SQL procedure or external procedure to be executed.

If *database\_name* is not specified, the current default database is assumed.

***procedure\_name***

Name of the SQL procedure or external procedure to be executed.

***value\_expression***

Supported arithmetic and string expressions.

The following can be specified in a value expression, subject to client-specific restrictions:

- SQL procedure local variables
- SQL procedure status variables
- IN or INOUT parameters
- FOR loop columns and aliases
- Host variables and macro parameters
- FORMAT, TITLE, and NAMED phrases
- Scalar UDFs
- Scalar subqueries

For client-specific rules, see [Rules For Calling Procedures From Embedded SQL](#).

**?**

A call parameter argument.

A QUESTION MARK character as an input call argument is valid only in SQL DML, ODBC, and JDBC client applications.

#### ***out\_call\_variable***

An identifier prefixed with the COLON (:) character.

Depending on the calling utility, the *out\_call\_variable* can be one of these:

- host variable
- local variable
- IN or INOUT parameter

#### ***out\_call\_placeholder***

A parameter name.

The placeholder provides for nesting of placeholders (parameters or CAST ... AS clauses).

#### ***parameter\_name***

The name of the OUT parameter as defined in the SQL procedure.

#### **CAST ... AS**

The request to convert the data definition of a parameter or another CAST clause to the required type. CAST clauses can be nested.

FORMAT, NAMED and TITLE clauses can be used with the CAST operator.

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

The data definition for the parameter set.

See *Teradata Vantage™ Data Types and Literals*, B035-1143.

### **ANSI Compliance**

CALL is ANSI SQL:2011-compliant.

## **Usage Notes**

### **Invocation**

Interactive SQL, embedded SQL, SQL procedures, and macros.

### **CALL Statements and Multistatement or Iterated Requests**

You cannot include CALL statements in iterated requests or multistatement requests.

## Rules for Executing SQL Procedures and External Stored Procedures

The following rules apply to executing SQL procedures and external procedures:

- SQL procedures and external procedures are platform-specific. A procedure created under one platform can only be run on that platform.
- CALL can be performed only in Teradata or ANSI session modes.

If an SQL procedure is created in Teradata session mode, it cannot be executed in ANSI session mode and vice versa.

CALL cannot be used in 2PC (Two Phase Commit) mode.

- Additionally, CALL can be issued in Prepare or Execute mode using CLI applications.
- LOCKING modifiers cannot be used with a CALL statement.
- You can submit a CALL statement from a macro if the CALL statement is the only statement inside the macro.
- Only one nested procedure call can be an external procedure, the rest must be SQL procedures. For example, an external procedure cannot call an SQL procedure that in turn calls an external procedure.

## Rules For Specifying Input And Output Parameters

Call arguments consisting of input and output parameters must be submitted with a CALL statement. No default parameter values can be defined at the time a procedure is created; the CALL returns an error if the required call arguments are not specified.

The following rules apply to the parameter arguments:

- The number of call arguments in a CALL statement must be equal to the number of parameters in the called procedure.

If the called procedure has no parameters, you cannot specify any call arguments.

- An IN, INOUT or OUT argument must correspond to an IN, INOUT or OUT parameter respectively in the called procedure.
- The default data type for an INOUT parameter input value is determined by the narrowest type that can contain the data passed to the INOUT parameter at run time, not by the default type defined for the parameter in the CREATE PROCEDURE (External or SQL Form) statement that created the procedure.

To ensure that memory overflow errors do not occur when the size of an output value passed to an INOUT parameter exceeds the capacity of the default data type derived from the input value for that parameter, you must take either of the following precautions:

- Explicitly cast the INOUT parameter data type in the CALL statement to ensure that memory overflow errors do not occur.



For example, if you submit the following CALL statement, the system determines that the smallest data type that can contain the input value to the INOUT parameter is SMALLINT, which requires 2 bytes, so it assigns the SMALLINT type to the parameter irrespective of the data type assigned to it at the time the procedure was created.

```
CALL my_proc(32767);
```

However, if the call returns a value greater than or equal to 32,768 to the INOUT parameter, the statement aborts unless you cast its type to the type that was assigned to the parameter at the time the procedure was created, or at minimum to INTEGER. The INTEGER type can contain positive values as large as 2,147,483,647, the BIGINT type can contain positive values as large as 9,223,372,036,854,775,807, the DECIMAL/NUMERIC type can contain far larger numbers, and so on. For details, see *SQL Data Types and Literals* and *Database Design*. For example, you can rewrite the statement as follows:

```
CALL my_proc (CAST('32767' AS INTEGER));
```

- Code the calling application to ensure the appropriate type conversion is made. For a JDBC example, see example 4 in [Example: Preventing Memory Overflow Errors for INOUT Parameters](#).

For details, see “CREATE PROCEDURE (External Form)” or “CREATE PROCEDURE (SQL Form)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

You must ensure either that the default data type for an INOUT parameter can accommodate the value or to know what type to cast it to in order to ensure that a memory overflow error does not occur. For information about how to determine the size of a literal, see “Data Literals” in *Teradata Vantage™ Data Types and Literals*, B035-1143.

- Some additional rules and restrictions apply when CALL is submitted from various client utilities. See the following for the specific rules and restrictions:
  - [Rules For Call Arguments In BTEQ And CLIV2](#)
  - [Rules For Calling Procedures From Embedded SQL](#)
  - [Rules For Call Arguments In ODBC And JDBC](#)
  - [Rules For Call Arguments In Nested Procedures](#)
- IN parameters can only have input, and OUT parameters can only have output. INOUT parameters can have both input and output.
- The data type of a call argument must be compatible with the data type of the corresponding parameter in the called procedure.
- A value expression of NULL can be used to assign NULL to the corresponding IN or INOUT parameter in the called procedure.
- On successful completion of the procedure execution, the ACTIVITY\_COUNT in the success response is set to the following values:

ACTIVITY_COUNT Setting	Procedure Contains
1	Output (INOUT or OUT) parameters.
0	No output parameters.

## Rules For Call Arguments In BTEQ And CLv2

These rules apply to call arguments submitted from applications in BTEQ or CLv2:

- An IN or INOUT argument must be a value expression.
- In a value expression used as IN or INOUT argument, identifiers prefixed by the colon (:), if any, must refer to USING variables associated with a USING clause for the statement containing the CALL. The value of the expression is treated as the input value for the corresponding parameter in the called procedure.
- An OUT argument can be any of the following:
  - An output host variable.
  - A place holder such as a QUESTION MARK character.
  - Any name other than the OUT parameter name specified in the procedure definition.
- These rules apply to using ? parameters.

The following procedure call returns AMBIGUOUS, AMBIGUOUS as the information for the parameters *p1* and *result\_1* with the expectation that during the execution phase, additional information will be forthcoming about the type of the parameter *p1*, which will determine the resulting type of *result\_1* parameter.

```
CALL xsp2 (?, ?);
```

This following procedure call fails because there is no expectation that there will be additional information about what data type the *result\_1* parameter should return.

```
CALL xsp1 (?, ?);
```

These examples show variations in the specification of OUT parameter names in a CALL statement issued from BTEQ. They s are based on the following CREATE PROCEDURE statement:

```
CREATE PROCEDURE sp2 (OUT po1 INTEGER)
BEGIN
  SET :po1 = 20;
END;
```

In the following CALL statement, the OUT argument name *p1* differs from the OUT parameter name *po1* specified in the procedure definition:

```
CALL sp2(p1);
*** Procedure has been executed.
*** Total elapsed time was 1 second.
```

The same CALL statement also works if you specify a placeholder character or host variable instead of the explicit parameter specification, as the two following examples show:

```
CALL sp2(?);
*** Procedure has been executed.
*** Total elapsed time was 1 second.

CALL sp2(:tx1);
*** Procedure has been executed.
*** Total elapsed time was 1 second.
```

## Rules For Call Arguments In ODBC And JDBC

The following additional rules apply to a call argument when the CALL statement is submitted from an ODBC or JDBC application:

- An IN or INOUT argument must be one of the following:

- A value expression.

A value expression must not contain identifiers prefixed by the COLON character. It must be a constant expression.

- A QUESTION MARK (?) character used as an input placeholder.

If you specify ?, the value for the corresponding IN or INOUT parameter of the called procedure must be set using ODBC- or JDBC-specific calls prior to calling the procedure.

There is a 1:1 correspondence between the number of ? markers for IN and INOUT arguments and the number of data items specified in the StatementInfo parcel in the request message. StatementInfo does not contain entries for OUT arguments.

For example, consider the following SQL procedure definition and CALL statement:

```
CREATE PROCEDURE sp3 (
  IN pi1    INTEGER,
  INOUT pio1 INTEGER,
  OUT po1   INTEGER)
BEGIN
  SELECT j INTO :pio1
  FROM tb11
  WHERE i=2;
  SELECT k INTO :po1
  FROM tb11
```

```
WHERE i=2;
END;

CALL sp3 (:?, :?, :?);
```

When this call is made, the StatementInfo parcel contains 2 entries: one each for the IN and INOUT parameters.

- An OUT argument must be an OUT call placeholder.

## Rules For Call Arguments In Nested Procedures

The following additional rules apply to a call argument when the CALL statement is submitted from another stored procedure:

- An IN argument must be a value expression.  
Identifiers in the value expression prefixed by the COLON character (:), if any, must refer to local variables, status variables, IN and INOUT parameters, or for-loop variable qualified columns and aliases of the calling procedure.
- An INOUT argument must be a value expression that is limited to the form of an OUT-call-variable. The identifier must be prefixed with a COLON character and must refer to a local variable or an INOUT parameter of the calling procedure.
- An OUT argument must be an OUT\_call\_variable. The identifier must refer to a local variable or an INOUT or OUT parameter.

## Rules For Calling Procedures From Embedded SQL

The following additional rules apply to a call argument when the CALL statement is submitted from a PP2 embedded SQL application:

- An IN argument must be a value expression.  
Identifiers in the value expression prefixed by the COLON character (:), if any, must refer to host variables. The value of the expression is treated as the input value for the corresponding parameter.
- An INOUT argument must be a value expression that is limited to the form of an OUT\_call\_variable. The identifier must refer to a host variable.
- An OUT argument must be an OUT\_call\_variable. The identifier must refer to a host variable.

## Rules for Calling a Procedure With Dynamic Result Sets From Embedded SQL

To be able to access dynamic result sets returned by a procedure called from embedded SQL, you must ensure that two preprocessor options are set properly.

- Set the SQLCHECK option to FULL so the precompiler can pass the variables that receive the procedure OUT or INOUT parameters to the runtime phase of the preprocessor.

The following table indicates how to set this option for mainframe-attached and workstation-attached environments:

Enable SQL Syntax, Object Reference, and Privilege Checking to FULL	Option
Mainframe-attached	SQLCHECK(FULL)
Workstation-attached	-sc FULL

- If you define an embedded SQL cursor for a called procedure, you must also set the preprocessor TRANSACT option to BTET because CALL statements cannot be executed as part of a multistatement request in ANSI session mode.

The following table indicates how to set this option for mainframe-attached and workstation-attached environments:

Set the Transaction Mode to BTET	Option
Mainframe-attached	TRANSACT(BTET)
Workstation-attached	-tr BTET

See [Example: Calling a Stored Procedure that Returns Dynamic Result Sets from Embedded SQL](#).

## Session Dateform and Called Procedures

Called procedures retain the date and time formats in effect when the procedures were created. Called procedures do not reflect the dateform in effect for the session in which the procedures are called unless the date and time format is the same as the dateform in effect when the procedure was created.

See “SET SESSION DATEFORM” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## Retrieving Values of Output Parameters

The output values of INOUT and OUT parameters are returned after completion of the CALL statement in the following forms:

CALL is Issued from this Type of Application	Output Parameter Values
BTEQ	Result row.
CLIV2	The values for all the INOUT and OUT parameters are returned in one row, irrespective of the number of parameters.

CALL is Issued from this Type of Application	Output Parameter Values
	The value of each parameter is a field in the output.
Embedded SQL	Stored in the corresponding host variable.
Another procedure	stored in the corresponding local variable or parameter of the calling procedure.
ODBC	To be retrieved using an ODBC API.
JDBC	To be retrieved using a JDBC API.

## Status of Unqualified Referenced Objects

If the objects referenced in the called procedure are not qualified by a database name, then they are qualified by the name of the default database at the time the procedure was created.

## Dropped, Renamed, or Replaced Objects

Consider the scenario when a procedure references another procedure that has been dropped or renamed.

If the called procedure *X* references another procedure *Y*, and *Y* is later dropped, renamed, or replaced before executing *X*, the following occurs:

Session Mode	Condition Generated when X Submits an Internal CALL to Execute Y
ANSI	Error
Teradata	Failure

If *Y* is replaced, then its execution generates an error in ANSI session mode or a failure in Teradata session mode if the specified arguments are not compatible with the changed definition of *Y*.

The following cases cause an error or failure:

- The number of call arguments is not equal to the number of parameters in the changed definition of *Y*.
- The data type of the call arguments is incompatible with the changed parameters of *Y*.

These conditions also apply to the attributes of any referenced database objects.

## Java External Procedure-Specific Behavior

The system performs the following actions on the dictionary tables referenced by the SQLJ database during a CALL statement that executes a Java external procedure:

1. Validates the Java procedure to be called.
2. Retrieves the row matching the Java procedure name from the DBC.Routine\_Jar\_Usage table to determine which JAR must be loaded to execute this Java procedure.
3. Verifies that the JAR named in DBC.Routine\_Jar\_Usage exists by finding the corresponding row in the DBC.Jars table.
4. Retrieves any rows from the DBC.Jar\_Jar\_Usage table that indicate whether this JAR includes other JARs, and makes sure those JARs are also loaded.

## Called Procedure Priority

A procedure executes at the priority the user session is running at the time the CALL statement is submitted. If you specify a SET SESSION ACCOUNT at a REQUEST level and the CALL statement is submitted, the procedure along with the SQL statements within the procedure execute at the specified priority.

The account is set to the previous account after the CALL statement is completed.

If the user priority is changed asynchronously during procedure execution, the new priority takes effect for all the subsequent requests that are submitted as part of procedure execution.

If the asynchronous change is at the request level, the priority change applies to all the statements executed as part of the CALL statement.

Consider the following SQL procedure:

```
CREATE PROCEDURE prio2()
BEGIN
  INSERT INTO temp(1, 'stored procedure before prio1') /* STMT1 */;
  CALL prio1() /* STMT2 */;
  INSERT INTO temp(2, 'stored procedure after prio1') /* STMT3 */;
END;
```

## Scenarios

The following three scenarios explain the priority scheduling for the procedure execution:

### Scenario 1

```
LOGON user1/user1, acct1;
CALL prio2() /* this, along with the SQL statements inside */;
```

```

/* the procedure are executed at the */;
/* priority associated with acct1 */;

```

## Scenario 2

```

LOGON user1/user1, acct1;
CALL prio2() /* this, along with the SQL statements inside */;
/* the procedure are executed at the */;
/* priority associated with acct1 */;
SET SESSION ACCOUNT acct2 FOR REQUEST;
CALL prio2() /* this, along with the SQL statements inside */;
/* the procedure are executed at the */;
/* priority associated with acct2 */;
SELECT * FROM temp /* this is executed at the priority */;
/* associated with acct1 */;

```

## Scenario 3

Assume that the priority is changed for user1 to acct2, after executing the STMT 1. The STMT 2 and STMT 3 (along with the SQL requests inside prio1) execute in the changed priority. If the priority is changed at request level, the session priority is restored to that corresponding to acct1 at the end of the execution of prio2.

```

LOGON user1/user1, acct1;
CALL prio2() /* this is executed at the priority associated */;
/* with acct1 */;

```

## Errors and Failures in Procedure Execution

### Errors and Failures in Procedure Execution

An error (ANSI session mode) or failure (Teradata session mode) is reported during procedure execution in any of the following situations:

- The database objects referenced in the stored procedure have been dropped or renamed after the procedure creation.
- The attributes of the referenced objects or of the parameters of the objects have been changed.
- Translation error or failure occurs when the value of an IN or INOUT argument is converted implicitly to the corresponding parameter data type and assigned to the corresponding parameter.

See [Dropped, Renamed, or Replaced Objects](#).

## Errors and Failures in Nested Procedures

If an error or failure occurs in case of nested procedures, the error or failure message text is preceded by the name of the procedure in which the error has occurred.



Assume that procedure *X* references another procedure, *Y*.

Error or Failure Occurs	Error or Failure Text Contains the Following Text as the Procedure Name
<i>Y</i> returns an error or failure during the execution of procedure <i>X</i>	<i>Y</i>
Error or failure occurs in the calling of procedure <i>X</i>	<i>X</i>

## Aborting a CALL Statement

You can use the standard abort facility provided by the client utility or interface to abort a CALL statement. The following action takes place:

You Specify the Abort Request	Action
During stored procedure execution.	Execution stops. The transaction being processed at that time is rolled back, and a failure response is returned.
After the execution completes.	Response of the CALL request is returned to the client.

## Asynchronous and Synchronous Abort Logic

An SQL CLI external stored procedure can issue asynchronous aborts. Normally if a client application issues an asynchronous abort, the system aborts the transaction on the Teradata platform, and an abort success message is sent to the client application; however, the system cannot do that for an SQL CLI external stored procedure. The system must roll back the transaction (if there is one) and return the abort status to the SQL CLI external stored procedure without affecting the client application in any way.

The undesirable side effect is that if the client application had an outstanding transaction, that transaction would no longer exist. As a general rule, you should not write your applications to submit CALL requests inside a transaction unless it is known that the stored procedure will not issue any asynchronous aborts.

This same issue is true for synchronous abort logic in stored procedures.

## Examples

### Example: Input Arguments in BTEQ and CLIV2

Consider the following stored procedure `spSample1` that has three IN parameters, *p1*, *p2*, and *p3* of INTEGER data type. The argument list can be specified in either of the following formats:

```
CALL spSample1 (1, 2, 3);
```

```
CALL spSample1(1, CAST(((2 + 4) * 2) AS FORMAT 'ZZZ9'), 3);
```

The rule of specifying the arguments as a positional list applies to all arguments (IN, INOUT, or OUT).

## Example: Input and Output Arguments in BTEQ and CLIV2

You can use an SQL SHOW PROCEDURE statement to view and verify the parameter types:

```
CREATE PROCEDURE spSample2(OUT p1 INTEGER, INOUT p2 INTEGER,
IN p3 INTEGER)
BEGIN
    SET p1 = p3;
    SET p2 = p2 * p3;
END;
```

The call arguments can be specified in any of the following formats:

### Format 1:

```
CALL spSample2(p1, 20, 3);
```

where the parameter name *p1* is supplied as an argument because it is a placeholder for an OUT parameter. The values 20 and 3 are passed as arguments to the INOUT and IN parameters, *p2* and *p3* respectively.

The CALL statement returns the following response:

p1	p2
-----	-----
3	60

### Format 2:

```
CALL spSample2(p1, 20 * 2, 30 + 40);
```

where the expressions (20 \* 2) and (30 + 40) are passed as arguments to the INOUT and IN parameters, *p2* and *p3*.

The CALL statement returns the following response:

p1	p2
----	----

-----	-----
70	2800

**Format 3:**

```
CALL spSample2(CAST(CAST(p1 AS CHARACTER(10)) AS TITLE 'OutputValue'),
CAST(20 AS SMALLINT), 30 + 40);
```

where the expressions `CAST(20 AS SMALLINT)` and `(30 + 40)` are passed as arguments to the INOUT and IN parameters, *p2* and *p3*.

The CALL statement returns the following response:

OutputValue	20
-----	-----
70	1400

**Format 4:**

This format is used with BTEQ and assumes that data is being imported from a data file.

```
USING (a INTEGER) CALL spSample1(p1, :a, 3);
```

where 3 is the value passed as an argument to the IN parameter *p3*. The value read from the input data file is assigned to the variable *a* and is passed as an argument to the INOUT parameter *p2* via a USING clause.

BTEQ receives the following response from the Teradata platform:

p1	p2
-----	-----
3	30

**Format 5 (Using NAMED and TITLE phrases in the CALL request):**

```
CALL spSample1(CAST (p1 AS NAMED AA TITLE 'OUT VALUE'),
CAST (((20 * 2) + 3) AS TITLE 'INOUT VALUE'), 1);
```

The response looks like this:

OUT VALUE	INOUT VALUE
-----	-----
1	43

## Example: Stored Procedure and Embedded SQL Input Arguments

Consider the stored procedure `spSample2` defined in [Example: Input and Output Arguments in BTEQ and CLIV2](#). The arguments can be specified in any of the formats shown:

### Format 1:

Only literals or constant expressions are specified as arguments for the parameters:

- In a stored procedure:

```
CALL spSample2(1, 2, 3 + 4);
```

- In a C program using embedded SQL:

```
EXEC SQL CALL spSample2(1, 2, 3 + 4);
```

### Format 2:

The application variables contain the values that are passed as arguments:

- In a stored procedure:

```
SET AppVar1 = 10;
SET AppVar2 = 30;
SET AppVar3 = 40;
CALL spSample2(:AppVar1, :AppVar1 + :AppVar2, CAST(:AppVar3 AS
FORMAT 'Z,ZZ9'));
```

- In a C program using embedded SQL:

```
AppVar1 = 10;
AppVar2 = 30;
AppVar3 = 40;
EXEC SQL CALL spSample2(:AppVar1, :AppVar1 + :AppVar2,
CAST(:AppVar3 AS FORMAT 'Z,ZZ9'));
```

### Format 3:

The combination of the application variables (*AppVar1*, *AppVar2*, and *AppVar3*) and values/expressions are specified as arguments:

- In a stored procedure:

```
SET AppVar1 = 10;
SET AppVar2 = 30;
SET AppVar3 = 40;
CALL spSample2(:AppVar1, 3 + :AppVar2, 3 + 4 + :AppVar3);
```

- In a C program using embedded SQL:

```
AppVar1 = 10;
AppVar2 = 30;
AppVar3 = 40;
EXEC SQL CALL spSample2(:AppVar1, 3 + :AppVar2, 3 + 4
                        + :AppVar3);
```

No output parameters are returned from the stored procedure using this format, so the `ACTIVITY_COUNT` is returned as 0 in the success response.

## Example: Stored Procedures and Embedded SQL Input and Output Arguments

Consider the stored procedure `spSample2`. The arguments can be specified in the following format:

Format 1:

- In a stored procedure:

```
SET AppVar2 = 30 + AppVar3;
SET AppVar3 = 40;
CALL spSample1(:AppVar1, :AppVar2, :AppVar3);
```

- In a C program using embedded SQL:

```
AppVar2 = 30 + AppVar3;
AppVar3 = 40;
EXEC SQL CALL spSample1(:AppVar1, :AppVar2, :AppVar3);
```

The values specified for *AppVar2* and *AppVar3* are passed as arguments to *p2* and *p3*, respectively. When the stored procedure execution completes, the output parameter values are returned in *AppVar1* and *AppVar2*. `ACTIVITY_COUNT` is set to 1.

Format 2:

- In a stored procedure:

```
SET AppVar2 = 30 + AppVar3;
SET AppVar3 = 40;
CALL spSample1(:AppVar1, :AppVar2, :AppVar3 + 3);
```

- In a C program using embedded SQL:

```
AppVar2 = 30 + AppVar3;
AppVar3 = 40;
EXEC SQL CALL spSample1(:AppVar1, :AppVar2, :AppVar3 + 3);
```

The values for *p2* and *p3* are *AppVar2* and  $(3 + AppVar3)$ , respectively. When the stored procedure execution completes, the output parameter values are returned in *AppVar1* and *AppVar2*. `ACTIVITY_COUNT` is set to 1.

## Example: Preventing Memory Overflow Errors for INOUT Parameters

If the size of an output value returned to an INOUT parameter is larger than the memory the system had allocated for the input value for that parameter, the CALL request fails and returns an overflow error to the requestor. See [Rules For Specifying Input And Output Parameters](#).

The following examples illustrate this. Suppose you have created an SQL stored procedure named myintz with an INOUT parameter.

### Example 1:

```
BTEQ -- Enter your DBC/SQL request or BTEQ command:
CALL myintz(32767);
```

The smallest data type the system can fit 32,767 into is SMALLINT, so the system allocates 2 bytes for the parameter and sets the type as SMALLINT irrespective of the data type assigned to the INOUT parameter when the procedure was created. If this CALL returns a value of 32,768 or more, the system treats the result as a memory overflow for a SMALLINT variable and returns an error.

### Example 2:

```
BTEQ -- Enter your DBC/SQL request or BTEQ command:
CALL myintz(CAST ('32767' AS INTEGER));
```

In this case, the system recognizes the input value is an INTEGER value because of the explicit casting, so it allocates 4 bytes for the parameter. Then, when the CALL returns 32,768 to the INOUT parameter, the value is within the scope of an INTEGER, the request completes successfully, and no memory allocation error occurs.

```
CALL myintz(cast ('32767' AS INTEGER));
*** Procedure has been executed.
*** Total elapsed time was 42 seconds.
'32767'
-----
      32768
```

Similarly, if the client program code calling a procedure explicitly defines the data type for the parameter, and the specified data type is capable of containing a larger output value than the default type for the input value, the request completes successfully.

The following examples illustrate this using a Java external stored procedure named myint2 called from a Java client application. Note that the basic principle applies for any stored procedure, not just external stored procedures. Assume the value returned to the INOUT parameter is 32,768 in both cases.

### Example 3:

Suppose you invoke the Java external stored procedure myint2 as follows:

```
stmt.executeUpdate("call myint2(32767);");
```

The outcome of this call is a memory overflow error, which is identical to that seen in the first example.

### Example 4:

Suppose a Java client program invokes a Java external procedure as follows:

```
StringBuffer prepCall = new StringBuffer("CALL myint2(?);");
System.out.println(prepareCall.toString());

CallableStatement cStmt = con.prepareCall(prepareCall.toString());

cStmt.setInt(1, integerVar[0]);
cStmt.registerOutParameter(1, Types.INTEGER);

// Making a procedure call
System.out.println("Before executeUpdate()...");

cStmt.executeUpdate();
System.out.println("Value after executeUpdate(): "
                  + cStmt.getInt(1));

integerVar[0] = cStmt.getInt(1);
```

This CALL succeeds because the calling Java program explicitly defined the INOUT parameter as an INTEGER data type, so the system allocates an appropriate amount of memory for an INTEGER input value. As a result, no overflow error occurs when the returned value can be contained within the memory allocated to an INTEGER type.

## Example: Input and Output Arguments in ODBC

### Example: Input and Output Arguments in ODBC

The following example describes the usage of the stored procedures specifying input and output arguments in an ODBC application.

Consider the stored procedure spSample2 defined as follows:

```
CREATE PROCEDURE spSample2(
  OUT p1    INTEGER,
  INOUT p2  INTEGER,
  IN p3     INTEGER)
BEGIN
  SET p1 = p3;
```

```
SET p2 = p2 * p3;
END;
```

The arguments can be specified in the following format:

```
SQLBindParameter(..., 1, SQL_PARAM_INPUT_OUTPUT, ..., SQLINTEGER,
..., ..., AppVar2, sizeof(AppVar2), ...);
SQLBindParameter(..., 2, SQL_PARAM_INPUT, ..., SQLINTEGER, ..., ...,
AppVar3, sizeof(AppVar3), ...);
```

where the second argument in `SQLBindParameter()` is the question mark number ordered sequentially from left to right, starting at 1.

Executing the stored procedure:

```
{
  constchar *request = "CALL spSample2(p1, ?, ?)";
  ...
  ...
  SQLExecDirect(hstmt, request);
  ...
  ...
}
```

Retrieving the output parameter values:

```
SQLBindCol(..., 1, ..., AppVar1, ..., ...);
SQLBindCol(..., 2, ..., AppVar2, ..., ...);
```

where the second argument in the `SQLBindCol()` is the parameter number of result data, ordered sequentially left to right, starting at 1.

## Example: Input and Output Arguments in JDBC

Consider the stored procedure `spSample2`, which can be executed using the following JDBC API calls:

```
CallableStatement cstmt = con.prepareCall("CALL spSample2(p1, ?, ?)");
ResultSet rs = cstmt.executeQuery();
```

The following alternatives can be used for the second line (`ResultSet rs = ...`):

```
boolean bool = cstmt.execute();
```

or

```
int count = cstmt.executeUpdate();
```



The QUESTION MARK character indicates an argument and acts as a placeholder for IN or INOUT parameters in the `prepareCall()` request. The question mark placeholder arguments must be bound with the application local variables and literals using the `CallableStatement.setXXX()` JDBC API calls.

Alternatively, an IN or INOUT argument can be a constant expression.

The INOUT and OUT arguments need to be registered using the following JDBC API call:

```
CallableStatement.registerOutParameter()
```

After execution the parameter values can be retrieved from the response using the `CallableStatement.getXXX()` JDBC API calls.

## Example: Calling a Stored Procedure that Returns Dynamic Result Sets from Embedded SQL

Consider the following stored procedure that returns a limit of four dynamic result sets to a calling embedded SQL application:

```
CREATE PROCEDURE TESTSP1(INOUT p1 INTEGER, OUT p2 INTEGER)
DYNAMIC RESULT SETS 4
BEGIN
  DECLARE tab1_cursor CURSOR WITH RETURN ONLY TO CALLER FOR
    SELECT c1, c2, c3 FROM tab1;
  OPEN tab1_cursor;
  SET p1 = p1 + 1;
  SET p2 = 82;
END;
```

To access the dynamic result set rows returned to the application by this procedure, declare a cursor like the following:

```
EXEC SQL BEGIN DECLARE SECTION;
...
long H1;
long H2;

long M1;
long M2;
long M3;
...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL
  DECLARE TESTCUR CURSOR
    FOR 'CALL TESTSP1(:H1, :H2)';
```

```
EXEC SQL
    OPEN TESTCUR;
```

If the procedure returns one or more rows, the system sets SQLCODE to 3212 and SQLSTATE to 'T3212'. This example does not use SQLSTATE return codes. The SQLSTATE codes equivalent to the SQLCODE values in the example are provided for the sake of completeness.

If the stored procedure returns no rows, the system sets SQLCODE to 0 and SQLSTATE to 'T0000':

```
if (SQLCODE == 3212)
    fetch_rows();
else if (SQLCODE != 0)
    error_check();
```

Fetching rows from a cursor declared for a stored procedure is similar to fetching rows from a cursor defined for an SQL request:

```
void fetch_rows()
do {
    EXEC SQL
        FETCH TESTCUR INTO :M1, :M2, :M3;
    ...
} while (SQLCODE == 0);
```

## Example: Using an SQL UDF as an Argument for an External Stored Procedure

This example passes the SQL UDF *value\_expression* as an argument to the external stored procedure *spAccount*.

```
CALL spAccount(test.value_expression(3,4), outp1);
```

## Example: Specifying a RETURNS or RETURNS STYLE Clause for an OUT Parameter Return Type

The first example shows how to use a RETURNS clause when invoking an external stored procedure. The RETURNS clause explicitly specifies a return type of INTEGER for the *result\_1* OUT parameter.

For this example to be valid, the OUT parameter *result\_1* must be defined with a data type of TD\_ANYTYPE in *xsp\_1*.

```
CALL xsp_1(10.25, result_1 RETURNS INTEGER);
```

The second example shows how to specify a RETURNS STYLE clause when invoking an external stored procedure.

This example refers to table *t1*, which has the following definition.

```
CREATE TABLE t1 (
  int_col      INTEGER,
  varchar_col  VARCHAR(40) CHARACTER SET UNICODE);
```

The RETURNS STYLE clause implicitly specifies a return type of VARCHAR(40) CHARACTER SET UNICODE for the *result\_1* OUT parameter of *xsp\_2* because that is the data type for column *varchar\_col* in table *t1*.

For this example to be valid, the OUT parameter *result\_1* must be defined with a data type of TD\_ANYTYPE in *xsp\_2*.

```
CALL xsp_2(10, 'abc', result_1 RETURNS STYLE t1.varchar_col);
```

## Example: Using RETURNS and RETURNS STYLE Clauses in the Same Procedure Call

This example specifies both RETURNS and RETURNS STYLE clauses in a CALL request using the following partial procedure definition.

```
CREATE PROCEDURE XSP_1(
  IN A INT,
  OUT B TD_ANYTYPE,
  OUT C INT,
  OUT D TD_ANYTYPE,
  OUT E INT,
  OUT F TD_ANYTYPE)
...;
```

You can call *xsp\_1* using the following parameter specifications, mixing both RETURNS and RETURNS STYLE OUT parameter return types.

```
CALL myXSP1(10, RESULT_B RETURNS INTEGER, RESULT_C,
  RESULT_D RETURNS STYLE t1.int_col,
  RESULT_E, RESULT_F RETURNS INTEGER);
```

## Related Topics

See *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184 for information on these statements:

- ALTER PROCEDURE (External Form)
- ALTER PROCEDURE (SQL Form)
- CREATE PROCEDURE (External Form)

- CREATE PROCEDURE (SQL Form)
- HELP PROCEDURE

For information about the TD\_ANYTYPE data type and how to code external routines for this data type, see *Teradata Vantage™ Data Types and Literals*, B035-1143 and *Teradata Vantage™ SQL External Routine Programming*, B035-1147.

## CHECKPOINT

### Purpose

Places a flag in a journal table that can be used to coordinate transaction recovery.

### Required Privileges

To checkpoint a journal table, you must meet at least one of the following criteria:

- Have CHECKPOINT privilege on the journal table.
- Have CHECKPOINT privilege on the database containing the journal table.
- Be an owner of the database containing the journal table.
- Be an immediate or indirect owner of the journal table.

### Syntax - Interactive Form

```
CHECKPOINT table_name [ , NAMED checkpoint_name ] ;
```

### Syntax Elements

#### *table\_name*

The journal table that is to be marked with the checkpoint entry.

#### *checkpoint\_name*

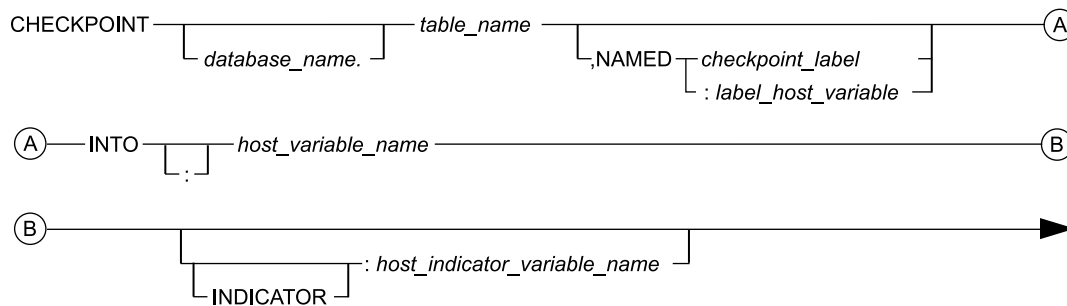
A label that can be used to reference the checkpoint entry in database recovery activities.

*checkpoint\_name* should be unique.

If *checkpoint\_name* duplicates an existing entry in the journal, then you can qualify it with the system-assigned event number.

If you do not specify *checkpoint\_name*, then the checkpoint entry must be referenced in recovery activities by its event number.

## Syntax - Embedded SQL and Stored Procedure Form



## Syntax Elements

### *database\_name*

Name of the database for which a checkpoint flag is to be created. The PERIOD character after the name is required.

### *table\_name*

Name of the table for which a checkpoint flag is to be created.

### *checkpoint\_label*

An SQL identifier that labels the checkpoint.

### *label\_variable*

Host variable that contains a label for the checkpoint.

The colon is required.

### *host\_variable\_name*

Name of a host variable to contain the checkpoint flag.

The preceding colon is optional.

### *host\_indicator\_variable\_name*

Name of an optional host indicator variable to receive nulls.

## ANSI Compliance

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

## General Usage Notes

CHECKPOINT causes the system to place a READ lock on all data tables that write journal images to the table named in the CHECKPOINT request. This lock causes any new transactions to wait until the checkpoint operation is complete. It also causes the checkpoint operation to await the completion of outstanding update transactions.

This action guarantees that the checkpoint saved in the journal represents a clean point in the transaction environment. When the checkpoint operation completes, the system releases the locks.

The system assigns an event number to each checkpoint entry. This number can be returned as a result of CHECKPOINT request processing; it is also stored, along with other information about request execution, in a data dictionary table. You can review the table data through the DBC.EventsV system view.

If an explicit transaction or a multistatement request contains a CHECKPOINT request, then that CHECKPOINT request must precede any INSERT, UPDATE, or DELETE requests in the transaction or request.

## Usage Notes for Stored Procedures and Embedded SQL

The following rules apply to CHECKPOINT:

- The stored procedure and embedded SQL form of CHECKPOINT is a data returning request.
- Although CHECKPOINT is a data returning request, it cannot be associated with a selection cursor.
- Whether specified as *checkpoint\_label* or as *label\_host\_variable*, the checkpoint label must be a valid SQL identifier.
- If you specify *label\_host\_variable*, the host variable must follow the rules for SQL strings for the client language and must be preceded by a colon. For details, see *SQL Stored Procedures and Embedded SQL*.
- The main host variable identified by *host\_variable\_name* must be a type that conforms to INTEGER.
- CHECKPOINT cannot be performed as a dynamic SQL statement.

CHECKPOINT causes a synchronization point to be generated and recorded in the journal table specified by *table\_name*.

If you specify the NAMED clause, the checkpoint label is associated with the synchronization point. A 32-bit integer that uniquely identifies the synchronization point is returned into the main host variable defined in the host variable specification.

CHECKPOINT is not valid in embedded SQL when you specify the TRANSACT(2PC) option to Preprocessor2.

## Example: Place a Checkpoint Entry into a Journal Table

The following request can be used to place a checkpoint entry into a journal table:

```
CHECKPOINT account_db.jnl_table, NAMED daily_pay_maint;
```

When this request is executed, all update activity is ceased on tables that write journal images to the journal table, (*jnl\_table*) and an event number is returned.

A row containing the daily\_pay\_maint label is placed into the journal table. The label can then be used in rollforward or rollback operations.

## COMMENT (Comment-Retrieving Form)

### Purpose

Retrieves a previously placed comment on a database object or definition.

For information about the comment-placing form of this SQL statement, see “COMMENT (Comment-Placing Form)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

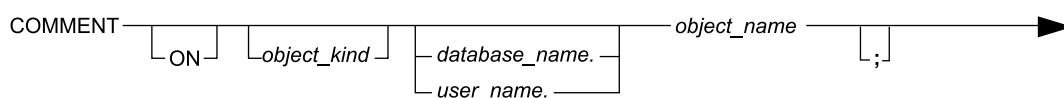
## Required Privileges

None.

## Privileges Granted Automatically

None.

## Syntax



## Syntax Elements

### *object\_kind\_1*

Mandatory database object kind specification.

The valid specifications for *object\_kind\_1* are not all database objects. Triggers and views, for example, are definitions of actions or conceptual groups rather than database objects.

You *must* specify the following database object kinds to retrieve a comment for the kind of object they represent.

- COLUMN
- FUNCTION
- GLOP SET
- GROUP
- MACRO
- METHOD
- PROCEDURE
- PROFILE
- ROLE
- TRIGGER
- TYPE
- VIEW

### *object\_kind\_2*

Optional database object kind specification.

You *can* specify the following database object kinds to retrieve a comment for the kind of object they represent, but they are optional.

- DATABASE
- TABLE
- USER
- FILE

The semantics of the COMMENT statement require that if you specify an object kind, you must specify the keyword TABLE for any hash or join index whose comment you want to retrieve; however, because TABLE is not a mandatory object kind specification, you are not required to specify it.

If you omit the `object_kind` when you retrieve a comment for a file, the `object_name` must contain the whole path to the file, for example, `dbname.uifname`.

#### ***database\_name***

#### ***user\_name***

Containing database or user for *object\_name* if it is not contained by the current database or user.

You cannot specify a database name or user name if *object\_kind* is GROUP.

For the rules to use in naming database objects, see *Teradata Vantage™ SQL Fundamentals*, B035-1141.

#### ***object\_name***

Name of the object for which a comment is to be retrieved, including:

- Parameter in a macro, stored procedure, or user-defined function.
- Column in a user base table, error table, hash index, join index, or view.
- Specific function, macro, profile, role, stored procedure, base table, error table, hash index, join index, trigger, or view name contained by a database or user.
- Database or user.
- UDT. You can retrieve a comment on a particular attribute of a structured UDT by specifying *database\_name.udt\_name.attribute\_name*.
- A method. You must use the *specific* method name.
- A GLOP set.

If you do not precede the object name with an object kind keyword, the system attempts to deduce the object from the level of qualification in the name. Use the fully-qualified name to avoid ambiguity.

Let x.y.z indicate the qualification hierarchy, with x being the highest, or coarsest grained, level and z the lowest, or most finely grained.

If you specify a hierarchy level of x, the specified object is implied to be one of the following:

- Database
- User

If you specify a hierarchy level of x.y, the specified object is implied to be one of the following:

- Base table
- Error table



- GLOP set
- Hash index
- Join index
- Macro
- Profile
- Role
- Stored procedure
- Trigger
- User-defined function
- View within database or user *x*.

If you specify a hierarchy level of *x.y.z*, the specified object is implied to be one of the following:

- Macro parameter
- Stored procedure parameter
- Structured UDT attribute
- Table column
- UDF parameter
- View column within GLOP set, UDF, UDT, macro, stored procedure, profile, role, base table, trigger, or view *y* contained in database or user *x*.

## ANSI Compliance

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

## Rules for Using COMMENT (Comment-Retrieving Form)

These rules apply to using the comment-retrieving form of the COMMENT statement:

- COMMENT (Comment-Retrieving Form) is treated as a DML statement for transaction processing, so it can be used in 2PC session mode. For a brief description of the two-phase commit protocol, see *Teradata Vantage™ - Database Introduction*, B035-1091.

You can specify a COMMENT (Comment-Retrieving Form) request at any point within the boundaries of an explicit transaction in Teradata session mode.

- To retrieve a comment, do *not* specify a comment string following the specification of the object name.

IF you ...	THEN the system ...
specify a comment string	does one of the following things: <ul style="list-style-type: none"> <li>◦ If the specified object has no existing comment, then Teradata Database places the specified comment string for the object in the data dictionary.</li> <li>◦ If the specified object has an existing comment, then Teradata Database replaces the existing comment in the data dictionary with the newly specified string.</li> </ul>
do not specify a comment string	does one of the following things:

IF you ...	THEN the system ...
	<ul style="list-style-type: none"> <li>◦ If the specified object has no existing comment, then Teradata Database returns a null.</li> <li>◦ If the specified object has an existing comment, then Teradata Database returns the text of that comment as it is stored in the data dictionary.</li> </ul>

- You cannot specify a containing database or user name if the object kind you specify is GROUP.

### Example: Retrieving a Comment

Suppose you have placed the following comment describing the *name* column in the *employee* table:

```
COMMENT ON COLUMN employee.name
  IS 'Employee name, last name followed by first initial';
```

The following request retrieves this comment:

```
COMMENT ON COLUMN employee.name;
```

The request returns the comment string as created by the previous request:

```
Employee name, last name followed by first initial
```

Note that because the object kind for this request is COLUMN, you must specify the keyword COLUMN when you code it.

### Example: Placing a Comment

Suppose you have defined a hash index named *ord\_hidx* defined on the *orders* table, which is contained in the *accounting* database.

Because the object kind for this request is TABLE, representing a hash index, you are not required to specify TABLE in the request.

You could type the following comment-placing COMMENT request to define a comment on the hash index *ord\_hidx*:

```
COMMENT accounting.ord_hidx AS 'hash index on Orders';
```

Then if you were to perform the following comment-retrieving COMMENT request, the text that follows would be returned to you:

```
COMMENT accounting.ord_hidx;
```

```
hash index on orders
```

# COMMIT

## Purpose

Terminates the current ANSI session mode SQL transaction, commits all changes made within it, and drops the Transient Journal for the transaction.

See also:

- [ABORT](#)
- [ROLLBACK](#)
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142

## Required Privileges

None.

## Syntax



## Syntax Elements

### WORK

If the SQL Flagger is enabled, the absence of WORK causes the request to be flagged.

For information about the SQL Flagger, see *Teradata Vantage™ SQL Fundamentals*, B035-1141.

### RELEASE

The connection between the client application program and the Teradata Database, whether implicit or explicit, is to be terminated.

This option applies to embedded SQL only and is not ANSI-compliant.

## ANSI Compliance

COMMIT is ANSI SQL:2011-compliant with an extension.

COMMIT is valid only in ANSI session mode. If used in Teradata session mode, it returns a failure and aborts the transaction.

Other SQL dialects support similar non-ANSI standard statements with names such as the following:

- COMMIT TRANSACTION
- COMMIT WORK

## How ANSI Transactions Are Defined and Terminated

In ANSI session mode, the first SQL request in a session initiates a transaction. The transaction is terminated by sending either a COMMIT or a ROLLBACK/ABORT request. Request failures do *not* cause a rollback of the transaction, only of the request that causes them.

### COMMIT Is Explicit

There are no *implicit* transactions in ANSI session mode. More accurately, each ANSI transaction is initiated implicitly, but always completed explicitly. The COMMIT must always be explicitly stated and be the last request of a transaction in order for the transaction to terminate successfully.

In ANSI session mode, you must issue a COMMIT (or ABORT/ROLLBACK) even when the only request in a transaction is a SELECT or SELECT AND CONSUME.

In the case of a SELECT request, it makes no difference whether you issue a COMMIT or an ABORT/ROLLBACK.

In the case of a SELECT AND CONSUME, there *is* a difference in the outcomes between issuing a COMMIT or ABORT/ROLLBACK because an ABORT or ROLLBACK request reinstates the subject queue table of the request to its former status, containing the rows that were pseudo-consumed by the aborted SELECT AND CONSUME request.

### Rules for Embedded SQL

The following rules apply to the COMMIT statement within an embedded SQL application:

- COMMIT cannot be performed as a dynamic SQL statement.
- COMMIT discards dynamic SQL statements prepared within the current transaction.
- COMMIT closes open cursors.
- COMMIT is valid only when you specify the TRANSACT(COMMIT), -tr(COMMIT), TRANSACT(ANSI), or -tr(ANSI) options to the preprocessor.

Its use causes an error if the program is precompiled with the TRANSACT(BTET), -tr(BTET), or the TRANSACT(2PC) preprocessor options.

- If you specify the RELEASE option, then the application program connection to Teradata Database (whether explicit or implicit) is terminated.

If additional SQL requests (other than CONNECT or LOGON) are then performed, an implicit connection is attempted.

- RELEASE is not valid when you specify the TRANSACT(ANSI) or -tr(ANSI) options to the preprocessor.

### Relation to ABORT and ROLLBACK

The ABORT and ROLLBACK statements also cause the current transaction to be terminated, but with rollback rather than commit. See [ABORT](#) and [“ROLLBACK”](#).

## COMMIT and BTEQ

If you use COMMIT in a BTEQ script with either the .SESSION or the .REPEAT command, you must send the COMMIT request along with the repeated SQL request as one request.

If you send the repeated request without the COMMIT, one of the requests is eventually blocked by other sessions and the job hangs because of a deadlock.

The following dummy example illustrates how this should be done:

```
.SESSION TRANS ANSI
.SESSIONS 10
.LOGON TDPID/USER,PASSWD

.IMPORT DATA FILE = data_file_name
.REPEAT I

USING i(INTEGER), j(INTEGER)
INSERT INTO table_name (col1, col2)
VALUES (:1, :j); COMMIT;

.QUIT
```

### Example: INSERT Request

The INSERT request in the following example opens the transaction. COMMIT closes the transaction.

```
INSERT INTO employee (name, empno)
VALUES ('Sinclair P', 101)
WHERE dept = '400';
COMMIT;
```

### Example: UPDATE Followed By COMMIT

The following UPDATE initiates the transaction and COMMIT WORK terminates it:

```
UPDATE parts SET part_num = 20555
WHERE location = 'seoul';
COMMIT WORK;
```

## DELETE

### Purpose

Removes one or more rows from a table.

There are four forms of the DELETE statement.

- **Basic** - removes one or more rows from a table. The basic form of DELETE is ANSI-compliant and you should use it for all delete statements that reference either a single table or reference tables other than the one deleted from only by subqueries in the *condition*.
- **Join Condition** - removes rows from a table when the WHERE condition directly references columns in tables other than the one from which rows are to be deleted; that is, if the WHERE condition includes a subquery or references a derived table.

Use this form when the request specifies a condition that directly references more than one table.

- **Positioned** - is invoked from an embedded SQL application. For details, see “DELETE (Positioned Form)” in *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148.
- **Temporal**. For information about syntax compatible with temporal tables, see *Teradata Vantage™ ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ Temporal Table Support*, B035-1182.

See also:

- “DELETE (Positioned Form)” in *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148
- “DELETE (Temporal Form)” in *Teradata Vantage™ Temporal Table Support*, B035-1182
- “DELETE (ANSI System-Time Form)” in *Teradata Vantage™ ANSI Temporal Table Support*, B035-1186
- “DELETE (ANSI Valid-Time Form)” in *Teradata Vantage™ ANSI Temporal Table Support*, B035-1186
- “DELETE (ANSI Bitemporal Table Form)” in *Teradata Vantage™ ANSI Temporal Table Support*, B035-1186

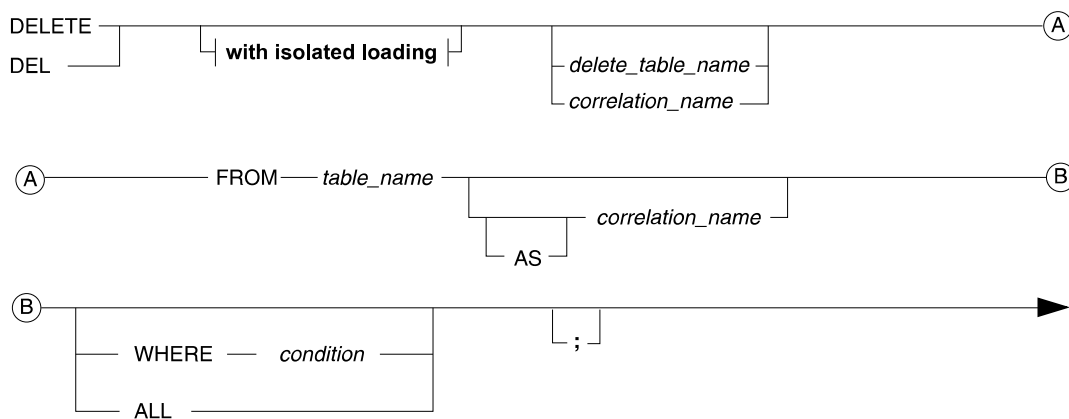
## Required Privileges

You must have the DELETE privilege on the table.

If the DELETE statement specifies WHERE conditions, you must also have the SELECT privilege on all tables and views through which they are referenced.

Use caution when granting the privilege to delete data through a view. Data in fields that might not be visible to the user is also deleted when a row is deleted through a view.

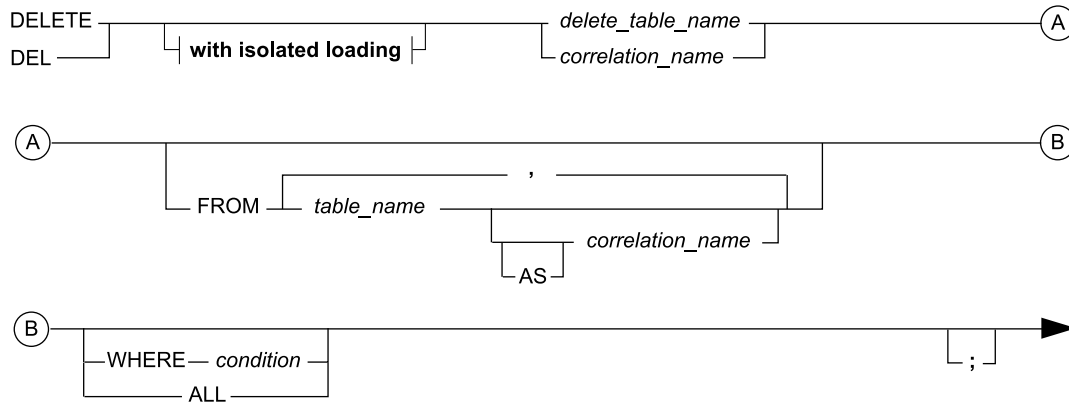
## Syntax - Basic Form



with isolated loading

WITH [NO] [CONCURRENT] ISOLATED LOADING

## Syntax - Join Condition Form



with isolated loading

WITH [NO] [CONCURRENT] ISOLATED LOADING

## ANSI Compliance

The basic form of DELETE is ANSI SQL:2011-compliant with extensions.

The join condition form of DELETE is a Teradata extension to the ANSI SQL:2011 standard.

## Syntax Elements

### Isolated Loading Options

#### WITH ISOLATED LOADING

The DELETE can be performed as a concurrent load isolated operation.

#### NO

The DELETE is not performed as a concurrent load isolated operation.

#### CONCURRENT

Optional keyword that can be included for readability.

## Delete Table Options

### *delete\_table\_name*

The table or queue table from which the DELETE statement is to remove rows.

The *delete\_table\_name* specification is optional when you specify a FROM clause. If you do not specify a *delete\_table\_name*, then the system deletes only the first table specified in the FROM clause.

If you specify a correlation name in the FROM clause, then you must specify that correlation name in place of *delete\_table\_name*.

### *table\_name*

The name of the table, queue table, or view on *delete\_table\_name* from which the DELETE operation is to remove rows.

## FROM Clause

### FROM *table\_name*

Name of a derived table, joined table, or view in the subquery referenced by the predicate of the WHERE clause.

If a row from *delete\_table* is joined with a row from another table in the FROM clause, and the specified WHERE condition for the request evaluates to TRUE for that joined row, then the row in *delete\_table* is deleted; else it is not.

See [Rules for Using Scalar Subqueries in a DELETE Statement](#) for the rules for using a scalar subquery for a derived table.

If you do not specify a FROM clause, then you cannot use correlation names. Compare [Example: Join Condition DELETE With FROM Clause and Correlation Name](#) and [Example: Join Condition DELETE With No FROM Clause](#).

You should also specify the names of all outer tables, including the table from which rows are to be deleted.

### AS *correlation\_name*

Optional table alias name.

You must specify a correlation name for each table specified in a self-join.

ANSI calls table aliases correlation names. They are also referred to as range variables.

## WHERE Clause

### *condition*

Predicate to filter the list of rows to be deleted.



The expression operands can be either constants or references to fields in the specified table or other tables. The predicate also can specify a scalar subquery. See [Scalar Subqueries](#) and [Rules for Using Scalar Subqueries in a DELETE Statement](#) for details.

## ALL

All rows in the table are to be deleted.

This is the default and is used when a WHERE condition is not specified.

The ALL option is a non-ANSI Teradata extension.

## Usage Notes

### Locks and Concurrency

A DELETE statement sets a WRITE lock for the table, partitions, or row. For a nonconcurrent load isolated delete operation on a load isolated table, the delete operation sets an EXCLUSIVE lock.

For a SELECT subquery, the lock depends on the isolation level for the session, the setting of the AccessLockForUncomRead DBS Control field, and whether the subquery is embedded within a SELECT or DELETE statement.

Transaction Isolation Level	DBS Control AccessLockForUncomRead Field Setting	Default Locking Severity for Outer SELECT and Ordinary SELECT Subquery Operations	Default Locking Severity for SELECT Operations Embedded Within an UPDATE Request
SERIALIZABLE	FALSE	READ	READ
	TRUE		READ
READ UNCOMMITTED	FALSE		READ
	TRUE		ACCESS

For more information, see:

- “SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ - Database Utilities*, B035-1102

## DELETE Processing Time

Processing time can vary between different syntaxes used to perform the identical delete operation. Use the EXPLAIN request modifier to determine which syntax form produces optimal processing time.

## Unconstrained Fastpath Delete Processing

To ensure that fastpath delete processing is enabled for unconstrained deletes performed using unconditional DELETE or DELETE ... ALL statements, there must not be any restrictions or conditions on the deletions and the statements must be positioned properly.

## Restrictions on Fastpath Deletions

To enable fastpath delete processing:

- The DELETE or DELETE ... ALL statement must not contain a WHERE clause.
- None of the rows of the target table specified in the statement for deletion can have any row-level security constraints.

## Fastpath Positioning Requirements

To enable fastpath delete processing, statements used to perform unconstrained deletes must be positioned properly. Proper positioning is determined by:

- Position of the statement within the transaction.
- Active session mode of the transaction.
- Type of transaction (ANSI, Implicit, Explicit).

### Position Within the Transaction

The speed of processing of an unconstrained delete depends on its position within the transaction.

When submitted as the last request of a transaction, other than for an implicit transaction, the DELETE or DELETE ... ALL statement must be the last statement that references the target table in the request that contains an END TRANSACTION or COMMIT. When a request is submitted as an implicit transaction, the DELETE or DELETE ... ALL statement must be the last statement that references the target table in the request.

### Proper Positioning For the Session Mode and Transaction Type

Because fastpath delete processing defers the delete until a transaction ends and because it allows rollback and delete processing to be almost instantaneous, make sure that the DELETE or

DELETE ... ALL statements are positioned properly based on the active session mode and the transaction type.

Session mode	Transaction Type	Positioning Requirements
ANSI	ANSI	<p>The DELETE ... ALL or unconditional DELETE must be in the same multistatement request that contains a COMMIT.</p> <p>For example,</p> <pre>DELETE FROM table_name ALL ;COMMIT;</pre>
Teradata	Implicit (such as a multistatement macro or BTEQ request)	<p>The DELETE ... ALL or unconditional DELETE must be the last statement that references the target table in the request.</p>
	Explicit	<p>The DELETE ... ALL or unconditional DELETE must be in the same multistatement request that contains the END TRANSACTION that terminates the currently open transaction.</p> <p>For example,</p> <pre>BEGIN TRANSACTION; DELETE FROM table_name ALL ;END TRANSACTION;</pre> <p>This is not valid for embedded SQL applications because the DELETE and END TRANSACTION statements must fall into the same request, and embedded SQL does not support multistatement requests.</p>

## Fastpath Delete and Join Indexes

Teradata Database uses fastpath delete processing when the target table has a simple or aggregate join index that is one of the following:

- Single-table join index.
- Multitable join index defined with an inner join.
- Multitable join index where the table being deleted is the outer table of an outer join.

This includes the following statements:

- A DELETE ALL or unconditional DELETE statement on a table that has a simple or aggregate join index. Teradata Database uses fastpath delete processing for the base table and its join index.
- A conditional DELETE statement on a table that has a simple or aggregate join index. Teradata Database uses fastpath delete processing only for the join index if the DELETE condition covers the entire index.

## Fastpath Delete for Multiple DELETE Statements

Teradata Database uses fastpath delete processing for multiple DELETE ALL statements in an implicit transaction or in the last request of a transaction.

## Constrained (Slow Path) DELETE Processing

The processing time for a constrained delete (DELETE...WHERE) can increase when:

- The DELETE does not meet the conditions for a fastpath delete processing. See [Unconstrained Fastpath Delete Processing](#).
- The FALLBACK option is specified for the table, because the rows in the secondary copy of the table must also be deleted.
- Secondary indexes are defined for the table, because the secondary index subtable also must be updated to account for the rows deleted.

Processing time for a constrained delete can be shortened by specifying a primary index value, a USI value, or a highly selective NUSI value as the conditional expression of the WHERE clause.

## Duplicate Rows and DELETE Processing

Duplicate rows in a MULTISSET table cannot be distinguished. When a WHERE condition identifies duplicate rows, all duplicate rows are deleted.

## General Rules for Using DELETE

The following rules apply to using DELETE.

- All correlation names must be specified in the FROM clause.
- The activity count in the success response for a DELETE statement reflects the total number of rows deleted.
- A DELETE statement that references objects in multiple databases should use fully-qualified names. Name resolution problems may occur if referenced databases contain tables or views with identical names and these objects are not fully qualified. Name resolution problems can occur even if the identically named objects are not explicitly referenced.
- You cannot delete a row from a base table that causes an insert into a join index that has row partitioning such that a partitioning expression for that index row does not result in a value between 1 and the number of partitions defined for that level. Otherwise, Teradata Database aborts the request (ANSI mode) or the transaction (Teradata mode).
- You cannot delete a row from a base table that causes an update of an index row in a join index with row partitioning such that a partitioning expression for that index row after the update does not result

in a value between 1 and the number of partitions defined for that level. Otherwise, Teradata Database aborts the request (ANSI mode) or the transaction (Teradata mode).

## Deleting Rows Using Views

To delete table rows using a view through which the table is accessed, refer to the following conditions.

- You must have the DELETE privilege on the view. Also, the immediate owner of the view (that is, the database in which the view resides) must have the DELETE privilege on the underlying object (view or base table) whose rows are to be deleted, and the SELECT privilege on all tables that are specified in the WHERE clause.
- Each column of the view must correspond to a column in the underlying table. That is, none of the columns in the view can be derived using an expression.
- The data type definitions for an index column should match in both the view definition and in the base table to which the view refers.

While it is true that you can generally convert the type of a view column (for example, from VARCHAR to CHARACTER), if that converted column is a component of an index, then that index is not used to delete rows from the base table because the data type of the recast column no longer matches the data type definition for that column in the index.

The resulting all-AMP, all-row behavior of the delete circumvents the performance advantages for which the index was designed.

- Any two view columns cannot reference the same table column.
- The view cannot include a column that contains a range constraint.
- The expression used to define a view cannot have a data type specified for any column in the view.

## Subqueries in a DELETE Statement

DELETE statement predicates can include subqueries that reference the delete target table, as well as other tables. The following DELETE statement is an example:

```
DELETE FROM publisher
WHERE 0 = (SELECT COUNT(*)
          FROM book
          WHERE book.pub_num=publisher.pub_num);
```

Two publishers have books in the library and two publishers do not.

The subquery executes once for each row of the outer reference, the *publisher* table. Because two publishers have no books in the library, the two rows that represent those publishers are deleted from the *publisher* table.

To modify this DELETE to use a noncorrelated subquery, change the subquery code to include all tables it references in its FROM clause.

```
DELETE FROM publisher
WHERE 0 = (SELECT COUNT(*)
          FROM book, publisher
          WHERE book.pub_num=publisher.pub_num);
```

When coded this way, the subquery predicate has a local defining reference, so the DELETE statement does not contain a correlated subquery. The count, determined once, is nonzero, so no rows are deleted.

## Rules for Using Scalar Subqueries in a DELETE Statement

You can specify a scalar subquery in the WHERE clause of a DELETE statement in the same way you can specify one for a SELECT statement.

You can also specify a DELETE statement with a scalar subquery in the body of a trigger. However, Teradata Database processes any noncorrelated scalar subquery you specify in the WHERE clause of a DELETE statement in a row trigger as a single-column single-row spool instead of as a parameterized value.

## Rules for Using Correlated Subqueries in a DELETE Statement

The following rules apply to correlated subqueries used in a DELETE statement:

- A DELETE statement requires that if joined tables are specified, all tables referenced in the DELETE statement must be specified in the FROM clause, including the deleted table.  
A *table\_name* must be added to specify the deleted table name in this case.
- All correlation names must be defined in the FROM clause.  
Correlation names are also referred to as range variables and aliases.
- If a correlation name is defined for the deleted table name in the FROM clause, then that correlation name, *not* the original table name, must be used as the *table\_name* that follows the DELETE keyword.
- If an inner query column specification references an outer FROM clause table, then the column reference must be fully qualified.
- The *table\_name* preceding the FROM clause is optional if no joined tables are specified for deletion.

Also see [Correlated Subqueries](#).

## DELETE Statement in the Last Request

If a DELETE statement is in the last request of a transaction and the following conditions are met, the delete is optimized to delete entire row partitions:

- The ANSI mode and Teradata mode conditions for fastpath delete processing are satisfied, with the exception that a deferred partition DELETE can be followed by another statement that references the same target table.
- The target table is not defined with referential integrity.

This avoids transient journaling of each row deleted from those row partitions.

## Deferred Deletion Applies Only to Range Terms

Deferred deletions do not occur for partitioning levels that are not defined using a RANGE\_N function. To qualify for deferred deletion, the test value of a RANGE\_N partitioning expression must be a simple column reference for a partitioning level.

## Deferred Deletion is Supported for LIKE Terms

Deferred deletion is also supported for LIKE terms defined in the form *partitioning\_column* LIKE 'abc%'.

## Deferred Deletion and Join Indexes

Teradata Database uses deferred partition deletion on row-partitioned base tables and row-partitioned join indexes for the following scenarios:

- Row-partitioned tables.
- Row-partitioned join indexes

### Deletion and Row-Partitioned Join Indexes

If you attempt to delete a row from a base table that causes an insert into a row-partitioned join index or an update of an index row in a row-partitioned join index such that:

- any of the partitioning expressions for that join index row evaluate to null, or
- the partitioning expression is an expression that is not CASE\_N or RANGE\_N, it's result is not between 1 and 65535 for the row

the system aborts the request (ANSI mode) or transaction (Teradata mode). It does not perform the insert or update and returns an error or failure, respectively.

Deleting a row from a base table does not always cause the deletion of a join index on that base table. For example, you can specify a WHERE clause in the CREATE JOIN INDEX statement to create a sparse join index for which only those rows that meet the condition of the WHERE clause are inserted into the index, or, for the case of a row in the join index being updated in such a way that it no longer meets the conditions of the WHERE clause after the update, cause that row to be deleted from the index.

The process for this activity is as follows:

1. The system checks the WHERE clause condition for its truth value after the update to the row.

Condition	Description
FALSE	the system deletes the row from the sparse join index.
TRUE	the system retains the row in the sparse join index and proceeds to stage b.

2. The system evaluates the new result of the partitioning expression for the updated row.

Partitioning Expression	Description
<ul style="list-style-type: none"> <li>Evaluates to null, or</li> <li>Expression is not CASE_N or RANGE_N</li> </ul>	<p>Result is not between 1 and 65535 for the row.</p> <p>The system aborts the request (ANSI mode) or transaction (Teradata mode). It does not update the base table or the sparse join index, and returns an error or failure, respectively.</p>
<ul style="list-style-type: none"> <li>Evaluates to a value, and</li> <li>Expression is not CASE_N or RANGE_N</li> </ul>	<p>Result is between 1 and 65535 for the row.</p> <p>The system stores the row in the appropriate partition, which might be different from the partition in which it was stored, and continues processing requests.</p>

Expression evaluation errors, such as divide by zero, can occur during the evaluation of a partitioning expression. The system response to such an error varies depending on the session mode in effect.

Session Mode	Expression Evaluation Errors Roll Back this Work Unit
ANSI	Request that contains the aborted request.
Teradata	Transaction that contains the aborted request.

Define your partitioning expressions to ensure that expression errors do not prevent the insertion of valid rows.

## Collation and Row Deletion

Collation can affect the deletion of rows from tables defined with a character partitioning expression.

- If the collation for a table is either MULTINATIONAL or CHARSET\_COLL and the definition for the collation has changed since the table was created, Teradata Database aborts any request that attempts to delete a row from the table and returns an error to the requestor.
- If a noncompressed join index with a character partitioning expression defined with either an MULTINATIONAL or CHARSET\_COLL collation sequence is defined on a table and the definition for the collation has changed since the join index was created, Teradata Database aborts any request that attempts to delete a row from the table and returns an error to the requestor whether the operation would have deleted rows from the join index or not.
- The session mode and collation at the time the table was created need not match the current session mode and collation for the delete operation to succeed.

Partitioning Expressions and Unicode



If a partitioning expression for a table or noncompressed join index involves Unicode character expressions or literals and the system has been backed down to a release that has Unicode code points that do not match the code points that were in effect when the table or join index was defined, Teradata Database aborts any attempts to delete rows from the table and returns an error to the requestor.

## Column Partitioned Tables and DELETE

For column partitions with ROW format, rows are physically deleted and the space is reclaimed.

## DELETE for NoPI Tables

DELETE is supported for NoPI tables.

DELETE has the following rules and restrictions for NoPI tables:

- A constrained DELETE request must use a full-table scan on a NoPI table unless the table has a secondary, join, or hash index and the Optimizer selects an access path that uses the index. However, column-partition elimination and row-partition elimination may result in an operation that does not require a full-table scan.
- To be eligible for fastpath delete processing, a DELETE ALL or unconstrained DELETE request on a NoPI table must conform to the rules described in [Fastpath Positioning Requirements](#). Deferred deletions for entire row partitions that are deleted reclaims space for any logically deleted data in that row partition.

For more information about NoPI tables, see *Teradata Vantage™ - Database Design*, B035-1094.

## Queue Tables and DELETE

The best practice is to avoid using the DELETE statement on a queue table because the operation requires a full table scan to rebuild the internal queue table cache. You should reserve this statement for exception handling.

A DELETE statement cannot be specified in a multistatement request that contains a SELECT and CONSUME request for the same queue table.

For details on queue tables and the queue table internal cache, see *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

## DELETE Support for External UDT Expressions

External UDT expressions are valid specifications for DELETE processing in the same way as any other valid expression. A UDT expression is any expression that returns a UDT value. See [Example: UDT Support For DELETE Statements](#) for valid examples of using a UDT expression in a DELETE operation.

## DELETE Support for SQL UDT Expressions

You can specify an SQL UDT as part of a WHERE clause condition in a DELETE statement if the SQL UDT returns a value expression. See [Example: Using an SQL UDF in a DELETE Request WHERE Condition](#).

## Embedded SQL and Stored Procedure Error Condition Handling

If DELETE is specified with a WHERE clause and the specified search condition fails because it selects no rows, the value '02000' is assigned to SQLSTATE, +100 is assigned to SQLCODE, and no rows are deleted.

A fastpath delete operation may not occur on load isolated tables due to presence of logically deleted rows.

## DELETE Support for Load Isolated Tables

A nonconcurrent load isolated delete operation on a load isolated table or join index physically deletes the qualified rows.

A concurrent load isolated delete operation on a load isolated table or join index logically deletes the qualified rows. The rows are marked as deleted and the space is not reclaimed until you issue an ALTER TABLE statement with the RELEASE DELETED ROWS option. See *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## Examples

### Example: DELETE Set of Rows

This example deletes all rows with a department number of 500 from the employee table:

```
DELETE FROM employee
WHERE deptno=500;
```

### Example: DELETE All Rows

This example deletes all rows from the employee table (the ALL keyword may be omitted):

```
DELETE FROM employee ALL;
```

## Example: DELETE Single Row

This example deletes the row for employee 10011 from the employee table:

```
DELETE FROM employee
WHERE empno=10011;
```

## Example: Join Condition DELETE

This example deletes rows from the employee table for employees who work in NYC. This operation joins the employee and department tables in the WHERE clause predicate:

```
DELETE FROM employee
WHERE employee.deptno = department.deptno
AND department.location = 'NYC';
```

## Example: Self-Join Condition DELETE

This example deletes rows from the employee table for managers who have less work experience than their employees. This delete performs a self-join on the employee table:

```
DELETE FROM employee AS managers
WHERE managers.deptno = employee.deptno
AND managers.jobtitle IN ('Manager', 'Vice Pres')
AND employee.yrsexp > managers.yrsexp;
```

## Example: Delete Rows with an Equality Constraint on a Partitioning Column

This is an example of deleting rows with an equality constraint on partitioning column. With partition level locking, an all-AMPs partition range lock is placed. The partition range has a single partition pair.

The table definition for this example is as follows:

```
CREATE TABLE HLSDS.SLPPIT1 (PI INT, PC INT, X INT, Y INT)
PRIMARY INDEX (PI)
PARTITION BY (RANGE_N(PC BETWEEN 1 AND 10 EACH 1))
```

An EXPLAIN of the DELETE statement shows the partition lock:

```
Explain DELETE HLSDS.SLPPIT1 WHERE PC = 10;
1) First, we lock HLSDS.SLPPIT1 for write on a reserved rowHash in a
   single partition to prevent global deadlock.
```

- 2) Next, we lock HLSDS.SLPPIT1 for write on a single partition.
- 3) We do an all-AMPs DELETE of a single partition from HLSDS.SLPPIT1 with a condition of ("HLSDS.SLPPIT1.PC = 10"). The size is estimated with no confidence to be 1 row. The estimated time for this step is 0.03 seconds.
- 4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

## Example: Delete Rows with an Indirect Constraint on the Target Table Partitioning Column

This delete operation uses an indirect constraint on target table partitioning column, that is slppit1.pc = srct1.b AND srct1.b = 10, to generate a single partition elimination list so a PartitionRange lock can be placed on table slppit1.

The table definition for this example is as follows:

```
CREATE TABLE HLSDS.SLPPIT1 (PI INT, PC INT, X INT, Y INT)
  PRIMARY INDEX (PI)
  PARTITION BY (RANGE_N(PC BETWEEN 1 AND 10 EACH 1));
CREATE TABLE HLSDS.SRCT1 (A INT, B INT, C INT) PRIMARY INDEX (A);
```

An EXPLAIN of the DELETE statement shows the partition lock:

```
Explain DELETE HLSDS.SLPPIT1 FROM HLSDS.SRCT1
  WHERE SLPPIT1.PC = SRCT1.B AND SRCT1.B = 10;
```

- 1) First, we lock HLSDS.SRCT1 for read on a reserved rowHash to prevent global deadlock.
- 2) Next, we lock HLSDS.SLPPIT1 for write on a reserved rowHash in a single partition to prevent global deadlock, and we lock HLSDS.SLPPIT1 for read on a reserved rowHash in a single partition to prevent global deadlock.
- 3) We lock HLSDS.SRCT1 for read, we lock HLSDS.SLPPIT1 for write on a single partition, and we lock HLSDS.SLPPIT1 for read on a single partition.
- 4) We do an all-AMPs RETRIEVE step from HLSDS.SRCT1 by way of an all-rows scan with a condition of ("(NOT (HLSDS.SRCT1.A IS NULL )) AND (HLSDS.SRCT1.B = 10)") into Spool 2 (all\_amps), which is duplicated on all AMPs. The size of Spool 2 is estimated with no confidence to be 4 rows (68 bytes). The estimated time for this step is 0.07 seconds.
- 5) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an all-rows scan, which is joined to a single partition of HLSDS.SLPPIT1 with a condition of ("HLSDS.SLPPIT1.PC = 10") with

a residual condition of ("HLSDS.SLPPIT1.PC = 10"). Spool 2 and HLSDS.SLPPIT1 are joined using a product join, with a join condition of ("HLSDS.SLPPIT1.PC = B"). The result goes into Spool 1 (all\_amps), which is built locally on the AMPs. Then we do a SORT to partition Spool 1 by rowkey and the sort key in spool field1 eliminating duplicate rows. The size of Spool 1 is estimated with no confidence to be 1 row (18 bytes). The estimated time for this step is 0.06 seconds.

- 6) We do an all-AMPs MERGE DELETE to HLSDS.SLPPIT1 from Spool 1 (Last Use) via the row id. The size is estimated with no confidence to be 1 row. The estimated time for this step is 0.81 seconds.
- 7) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

## Example: Macro For DELETE

The following macro structures the SELECT and DELETE statements as a single multistatement request:

```
CREATE MACRO res_use
  (from_date (DATE,      DEFAULT DATE),
   to_date   (DATE,      DEFAULT DATE),
   from_time (INTEGER,    DEFAULT 0),
   to_time   (INTEGER,    DEFAULT 999999),
   proc      (VARCHAR(4), DEFAULT 'P'),
   secs      (SMALLINT,   DEFAULT 600) )
AS (SELECT
  the_date (TITLE 'Resource//TheDate'),
  the_time (TITLE 'Utilized//TheTime'),
  proc,
  AVERAGE(hits) (TITLE 'Avg//Hits'),
  MAXIMUM(cpu)  (TITLE 'Max//CPU'),
  AVERAGE(cpu) (TITLE 'Avg//CPU'),
  MAXIMUM(disk) (TITLE 'Max//Disk'),
  AVERAGE(disk) (TITLE 'Avg//Disk'),
  MAXIMUM(host) (TITLE 'Max//Host'),
  AVERAGE(host) (TITLE 'Avg//Host'),
  MAXIMUM(chan) (TITLE 'Max//Chan'),
  AVERAGE(chan) (TITLE 'Avg//Chan')
FROM DBC.res_use_view
GROUP BY the_date, the_time, proc
WHERE the_date BETWEEN :from_date AND :to_date
AND   the_time BETWEEN :from_time AND :to_time
AND   proc CONTAINS :proc)
```

```

        AND    secs EQ :secs
        ORDER BY proc, the_date, the_time

;DELETE FROM res_use_view ALL;);

```

If the preceding macro is executed in Teradata session mode, but not within the boundaries of an explicit transaction, fastpath delete processing is used for the DELETE statement. See [Unconstrained Fastpath Delete Processing](#).

In ANSI session mode, the system uses fastpath delete processing for the DELETE statement when a COMMIT is included in the macro or when a COMMIT is specified at the end of the request line, and immediately following the execution of the macro, within the request.

For example, the last line of the preceding macro would read as follows:

```

DELETE FROM res_use_view ALL ;
COMMIT;) ;

```

The system uses slow path processing for the DELETE statement when no COMMIT is stated in the macro itself or within the request that executes the macro.

## Example: DELETE ALL Multistatement Request

The DELETE ... ALL statement in the following BTEQ request invokes fastpath delete processing. The statements are combined into a multistatement request and are processed as an implicit transaction in Teradata session mode. Note that DELETE is the last statement in the request:

```

SELECT *
FROM DBC.log_on_off_v
WHERE log_date = DATE
AND    user_name = 'Administrator'
;SELECT log_time, user_name, event, account_name
FROM DBC.log_on_off_v
WHERE log_date = DATE
AND    user_name NOT IN ('Administrator', 'SecAdmin', 'Oper')
;SELECT log_time, event, logical_host_id, pe_no
FROM DBC.log_on_off_v
WHERE log_date = DATE
AND user_name = 'Oper'
;DELETE FROM DBC.log_on_off_v ALL ;

```

In ANSI session mode, the COMMIT statement must follow the DELETE request, so the last line of the above example would read:

```

DELETE FROM DBC.log_on_off_v ALL ;
COMMIT;

```

## Example: Join Condition DELETE With FROM Clause and Correlation Name

The following example deletes employees from the *employee* table using a join on the *employee* and *department* tables.

```
DELETE employee
FROM department AS d, employee
WHERE employee.dept_no = d.dept_no
AND salary_pool < 50000;
```

## Example: Join Condition DELETE With No FROM Clause

The following example uses different syntax to perform the same action as [Example: Join Condition DELETE With FROM Clause and Correlation Name](#). Because this DELETE statement does not specify a FROM clause, you cannot use correlation names for the tables.

```
DELETE employee
WHERE employee.dept_no = department.dept_no
AND salary_pool < 50000;
```

## Example: Join Condition DELETE With Derived Table Subquery

The following example uses a subquery to perform the same action as [Example: Join Condition DELETE With FROM Clause and Correlation Name](#) and [Example: Join Condition DELETE With No FROM Clause](#).

```
DELETE FROM employee
WHERE dept_no IN (SELECT dept_no
                  FROM department
                  WHERE salary_pool < 50000);
```

## Example: UDT Support For DELETE Statements

The following examples show valid use of UDT expressions in a DELETE statement:

```
DELETE FROM test_table
WHERE circle_udt < NEW circle('1,1,9');

DELETE FROM test_table1, test_table2
WHERE test_table1.mycirc = test_table2.mycirc;
```

## Example: DELETE and NoPI Tables

The following examples are based on this NoPI table definition:

```
CREATE TABLE new_sales,
FALLBACK (
  item_nbr  INTEGER NOT NULL,
  sale_date DATE FORMAT 'MM/DD/YYYY' NOT NULL,
  item_count INTEGER)
NO PRIMARY INDEX;
```

The following DELETE statement requires a full-table scan because *new\_sales* has neither a primary nor a secondary index.

```
DELETE FROM new_sales
WHERE item_nbr = 100;
```

The following DELETE statement uses fastpath delete processing if it is submitted in Teradata session mode as an implicit transaction.

```
DELETE FROM new_sales;
```

The following DELETE statement uses fastpath delete processing when it is submitted in Teradata session mode as a single request.

```
BEGIN TRANSACTION
;DELETE FROM new_sales
;END TRANSACTION;
```

Assume the following single-statement request is submitted in Teradata session mode.

```
BEGIN TRANSACTION;
```

Then the following DELETE statement uses fastpath delete processing when it is submitted with an END TRANSACTION statement in a multistatement request.

```
DELETE FROM new_sales
;END TRANSACTION;
```

The following DELETE statement uses fastpath delete processing when it is submitted with a COMMIT statement in ANSI mode in a multistatement request.

```
DELETE FROM new_sales
;COMMIT;
```



## Example: Using an SQL UDF in a DELETE Request WHERE Condition

You can specify an SQL UDF in a WHERE clause search condition if it returns a value expression.

```
DELETE FROM t1
WHERE a1 = test.value_expression(b1, c1);
```

## Example: Deleting from a Table with an Implicit Isolated Load Operation

For information on defining a load isolated table, see the WITH ISOLATED LOADING option for CREATE TABLE and ALTER TABLE in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

Following is the table definition for the example.

```
CREATE TABLE ldi_table1,
  WITH CONCURRENT ISOLATED LOADING FOR ALL
  (a INTEGER,
   b INTEGER,
   c INTEGER)
PRIMARY INDEX ( a );
```

This statement performs a delete on the load isolated table ldi\_table1 as an implicit concurrent load isolated operation:

```
DELETE WITH ISOLATED LOADING FROM ldi_table1 WHERE a > 10;
```

## Example: Deleting from a Table with an Explicit Isolated Load Operation

For information on defining a load isolated table and performing an explicit isolated load operation, see the WITH ISOLATED LOADING option for CREATE TABLE and ALTER TABLE, in addition to Load Isolation Statements in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

Following is the table definition for the example.

```
CREATE TABLE ldi_table1,
  WITH CONCURRENT ISOLATED LOADING FOR ALL
  (a INTEGER,
   b INTEGER,
   c INTEGER)
PRIMARY INDEX ( a );
```

This statement starts an explicit concurrent load isolated operation on table ldi\_table1:

```
BEGIN ISOLATED LOADING ON ldi_table1
  USING QUERY_BAND 'LDILoadGroup=Load1;';
```

This statement sets the session as an isolated load session:

```
SET QUERY_BAND='LDILoadGroup=Load1;' FOR SESSION;
```

This statement performs an explicit concurrent load isolated delete from table `ldi_table1`:

```
DELETE FROM ldi_table1 WHERE a > 10;
```

This statement ends the explicit concurrent load isolated operation:

```
END ISOLATED LOADING FOR QUERY_BAND 'LDILoadGroup=Load1;';
```

You can use this statement to clear the query band for the next load operation in the same session:

```
SET QUERY_BAND = 'LDILoadGroup=NONE;' FOR SESSION;
```

## Example: Deleting a Row From a Table With Row-Level Security Protection

In this example, assume that the user submitting the request has the privileges required to delete a row from *inventory*.

The following DELETE statement deletes a row from a table named *inventory*.

```
DELETE FROM inventory
WHERE col_1 = 12126;
```

## Example: Row-Level Security DELETE and SELECT Constraints For User Lacking Required Privileges (DELETE)

This example shows how the DELETE and SELECT constraints are applied when a user without the required OVERRIDE privileges attempts to execute a DELETE statement on a table that has the row-level security DELETE and SELECT constraints.

An EXPLAIN request modifier is used to show the steps involved in the execution of the request and the outcome of the application of the constraints.

The statement used to create the table in this example is:

```
CREATE TABLE rls_tbl(
  col1 INT,
  col2 INT,
  classification_levels CONSTRAINT,
  classification_categories CONSTRAINT);
```

The user's session constraint values are:

```
Constraint1Name LEVELS
Constraint1Value 2
Constraint3Name CATEGORIES
Constraint3Value '90000000'xb
```

EXPLAIN Request Modifier with a DELETE Statement

This EXPLAIN request modifier shows the steps performed to execute the DELETE statement and the outcome of the application of the DELETE and SELECT constraints.

```
EXPLAIN DELETE FROM rls_tbl WHERE col1=2;
```

The system returns this EXPLAIN text.

```
*** Help information returned. 10 rows.
*** Total elapsed time was 1 second.
Explanation
-----
1) First, we do a single-AMP DELETE from RS.rls_tbl by way of the
   primary index "RS.rls_tbl.col1 = 2" with a residual condition of (
   "((SYSLIB.DELETELEVEL (RS.rls_tbl.levels ))= 'T') AND
   (((SYSLIB.SELECTLEVEL (2, RS.rls_tbl.levels ))= 'T') AND
   (((SYSLIB.DELETECATEGORIES (RS.rls_tbl.categories ))= 'T') AND
   ((SYSLIB.SELECTCATEGORIES ('90000000'XB, RS.rls_tbl.categories ))=
   'T')))). The size is estimated with no confidence to be 2 rows.
   The estimated time for this step is 0.01 seconds.
-> No rows are returned to the user as the result of statement 1.
   The total estimated time is 0.01 seconds.
```

ECHO

Purpose

Returns a fixed character string to the requestor.

Syntax



## Syntax Elements

### *string*

Text to be returned to the requestor.

### *command*

A BTEQ format command to be returned to the requestor. The format command must be terminated by a semicolon.

## ANSI Compliance

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

## Required Privileges

None.

## ECHO Not Supported for Embedded SQL

The Teradata preprocessor does not support ECHO.

## ECHO Not Supported for SQL Procedures

SQL procedures do not support ECHO.

## How ECHO Works

When the system encounters an ECHO request, it sends an ECHO parcel containing *string* to the client system, where it is then interpreted.

## ECHO and BTEQ

ECHO is used most frequently in macros to perform BTEQ commands during macro execution.

ECHO cannot be used with the following BTEQ commands.

- =
- EXPORT
- IMPORT
- LOGOFF
- LOGON
- QUIT
- REPEAT
- RUN

If *command* is not a BTEQ format command, BTEQ logs the illegal command as an error.

## Example: ECHO

The following ECHO request could be used in a macro that creates a report through BTEQ. When the system encounters the request during macro execution, the width of a printed page is set to 72 characters.

```
... ECHO 'SET WIDTH 72;' ...
```

## END TRANSACTION

### Purpose

Defines the completion of an explicit Teradata session mode transaction, commits the transaction, and drops its Transient Journal.

An explicit Teradata session mode transaction must always start with a BEGIN TRANSACTION statement, and you must always specify both BEGIN TRANSACTION and END TRANSACTION statements to define the limits of an explicit transaction in Teradata session mode.

See [BEGIN TRANSACTION](#) for additional usage information about END TRANSACTION.

See also:

- [ABORT](#)
- [BEGIN TRANSACTION](#)
- [COMMIT](#)
- [ROLLBACK](#)

### Required Privileges

None.

### Syntax

```
END TRANSACTION
ET _____▶
```

### ANSI Compliance

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

The statement is valid only in Teradata session mode. If you submit an END TRANSACTION statement in ANSI session mode, Teradata Database aborts the request and returns an error.

For ANSI session mode transaction control statements, see [COMMIT](#) and [ROLLBACK](#).

## BEGIN TRANSACTION/END TRANSACTION Pairs and Explicit Transactions

When you code an explicit transaction in Teradata session mode, you must initiate the transaction with a BEGIN TRANSACTION statement and complete it with an END TRANSACTION statement. See [BEGIN TRANSACTION](#).

## END TRANSACTION and Multistatement Requests

There can be no more than one END TRANSACTION statement specified in a multistatement request, and that statement, if specified, must be the last statement in the request.

Differences Between the Effects of END TRANSACTION and ABORT or ROLLBACK

While an END TRANSACTION request terminates and commits a transaction and drops its Transient Journal, ABORT and ROLLBACK requests terminate a transaction with rollback. See [ABORT](#) and [ROLLBACK](#).

In general, the rules for committing transactions using the COMMIT statement also apply to END TRANSACTION. See [COMMIT](#).

## Rules for END TRANSACTION Using Embedded SQL

The following rules apply to the use of END TRANSACTION in embedded SQL:

- END TRANSACTION is valid only when you specify the TRANSACT(BTET) or -tr(BTET) options to the preprocessor. Otherwise, an error is returned and the precompilation fails.
- END TRANSACTION cannot be performed as a dynamic SQL statement.
- When END TRANSACTION is processed, the transaction, if not already aborted, is committed.

# EXECUTE (Macro Form)

## Purpose

Performs a macro.

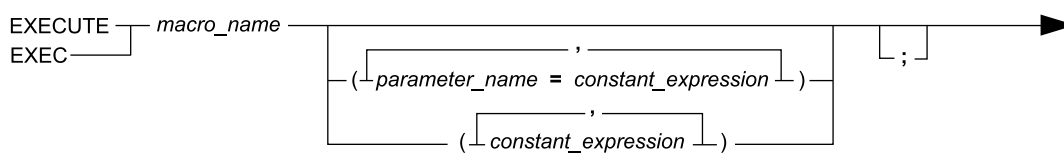
For information about the embedded SQL EXEC statement, which is used to run macros from embedded SQL applications, see *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148.

## Required Privileges

You must have EXECUTE privilege on the macro. The creator or any owner of the macro can grant the EXECUTE privilege to another. In addition, the immediate owner of the macro (the database in which the macro resides) must have the necessary privileges on objects named in the request set that are contained in the macro.

For more information, see *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184 and *Teradata Vantage™ - Database Administration*, B035-1093.

## Syntax



## Syntax Elements

### *macro\_name*

Name of the macro that is to be executed.

### *constant\_expression*

Constant or an expression involving constants that specifies a parameter value.

Values are listed in left-to-right order according to the order in which parameter names were specified in the CREATE MACRO request definition. You can specify nulls by typing successive COMMA characters, for example, (a, , b).

This is referred to as a positional parameter list.

If a value is not supplied for a parameter, a comma must be entered to account for the missing parameter.

Any default value defined for that parameter is inserted automatically.

### *parameter\_name = constant\_expression*

Parameter name as defined in the CREATE MACRO request, and supplies the value for that parameter.

This is referred to as a named parameter list.

The value can be a constant, or an expression involving constants.

If a parameter is omitted, any default value defined for that parameter is inserted automatically.

## ANSI Compliance

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

## Recommendation

A data definition request in a macro is not fully resolved until the macro is submitted for execution. Unqualified references to database objects are resolved at that time using the default database of the executing user.

Because of this, object references in data definition statements should always be fully qualified (as databasename.tablename) in the body of the macro.

## Rules for Performing Macros

The following rules apply to performing macros.

- The number of commas must match the macro definition if you do not use the *parameter\_name* syntax.
- Any value in the EXECUTE constant expression list form without a specified parameter name can be a constant or an expression involving constants. In this context, DATE, TIME, and USER are considered constants.
- If an error message is returned when a macro is executed, it can be caused by an SQL request in the macro.
- The number of parameters used in the calling sequence must be equal to the number of parameters defined.
- When, for example, two parameters are defined and used, if they are both null, the following requests are all valid unless defaults are specified in the macro definition:

```
EXECUTE macro_1 '(, 1);
EXECUTE macro_1 (,);
EXECUTE macro_1 (NULL, NULL);
```

## Access Logging and Errors

Any syntactic or semantic error identified and reported by the SQL parser results in an error being returned to you without logging the request.

## Example: Named Parameter List

This request uses a named parameter list to execute macro *new\_emp1*. See “CREATE MACRO” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144. Named parameters can be listed in any order.

```
EXECUTE new_emp1(number=10015, dept=500, name='Omura H', sex='M',
                position='Programmer');
```

The row for new employee Omura is inserted in the *employee* table.

## Example: Positional Parameter List

This example uses a positional parameter list to execute macro *new\_emp2*. See “CREATE MACRO” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144. Note that a value is not specified for the *dept* parameter, which has a macro-defined default value. A comma is entered in the *dept* position to maintain the integrity of the positional sequence.

```
EXECUTE new_emp2 (10021, 'Smith T', , 'Manager', 'F',
                'May 8, 1959', 16);
```

## Example: Automatically Inserted Value

When the following request is processed, the default value for the dept parameter (900) is inserted automatically. The row for new employee Smith is added to the *employee* table, and then the *department* table is updated by incrementing the value for Department 900 in the *emp\_count* column. The request uses



a named parameter list to execute a macro named *new\_hire*. Note that the value of the *DOH* (Date of Hire) column is an expression involving the DATE constant.

```
EXECUTE new_hire (fl_name='Toby Smith', title='Programmer',
                doh=DATE -1);
```

### Example: Invoking an SQL UDF as an Argument to Macro Execution

The following example invokes the SQL UDF *value\_expression* as an argument to the macro *m1*.

```
EXECUTE m1 (test.value_expression(1,2), 2, 3);
```

## INSERT/INSERT ... SELECT

### Purpose

Adds new rows to a table by directly specifying the row data to be inserted (valued form) or by retrieving the new row data from another table (selected, or INSERT ... SELECT form).

The syntax below describes the valued and selected forms of the statement, where *subquery* represents the SELECT portion of the selected form.

For information about syntax that is compatible with temporal tables, see *Teradata Vantage™ ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ Temporal Table Support*, B035-1182.

For more information, see:

- [Scalar Subqueries](#)
- [MERGE](#)
- *Teradata Vantage™ SQL Date and Time Functions and Expressions*, B035-1211
- “CREATE ERROR TABLE” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ ANSI Temporal Table Support*, B035-1186
- *Teradata Vantage™ Temporal Table Support*, B035-1182
- *Teradata Vantage™ JSON Data Type*, B035-1150
- *Teradata Vantage™ Geospatial Data Types*, B035-1181
- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Database Utilities*, B035-1102
- *Teradata® FastLoad Reference*, B035-2411
- *Teradata® MultiLoad Reference*, B035-2409
- *Teradata® Parallel Data Pump Reference*, B035-3021

### Required Privileges

The following privilege rules apply to the INSERT statement:

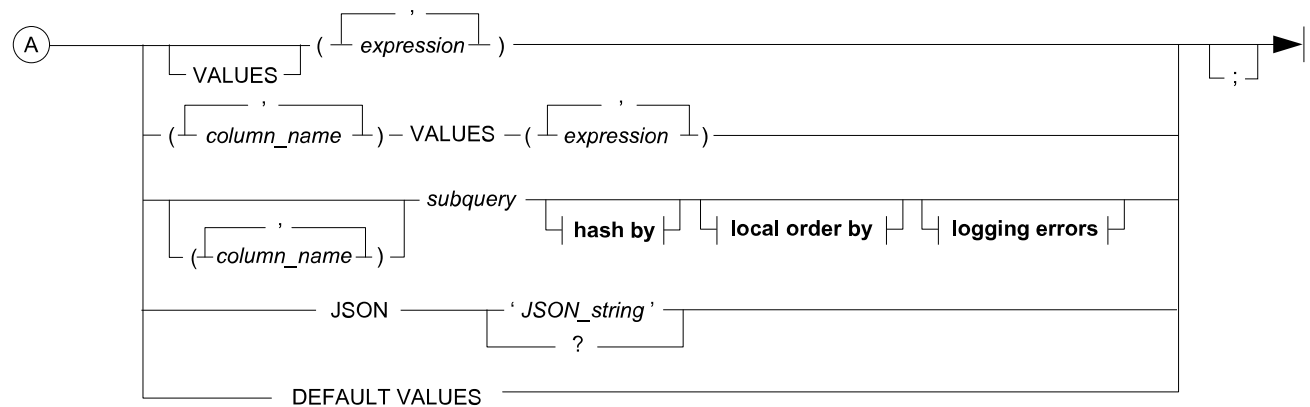
- You must have the INSERT privilege on the referenced table or column set.

- To insert rows into a table through a view, you must have the INSERT privilege on the view. Also, the immediate owner of the view must have the INSERT privilege on the underlying view, base table, or column set. The immediate owner of the view is the database in which the view resides.
- To insert rows into a table using a query specification, you must have the SELECT privilege for the referenced tables or views.
- When you perform an INSERT with a WHERE clause predicate that requires READ access to an object, you must have the SELECT privilege on the accessed object.

The privileges required for an INSERT ... SELECT ... LOGGING ERRORS operation are the same as those for INSERT ... SELECT operations without a LOGGING ERRORS option except that you must also have the INSERT privilege on the error table associated with the target data table for the INSERT ... SELECT operation.

## Syntax

INSERT  
INS with isolated loading INTO *table\_name* (A)

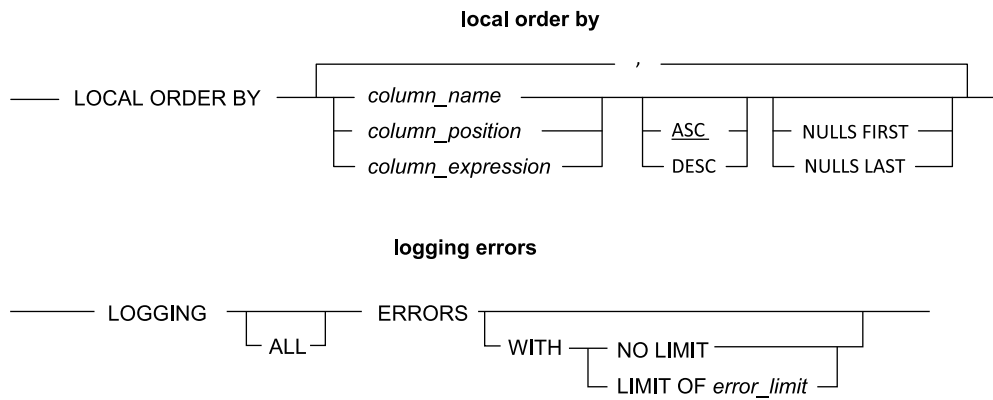


### with isolated loading

WITH NO CONCURRENT ISOLATED LOADING

### hash by

HASH BY column\_name column\_position column\_expression RANDOM



## Syntax Elements

### Isolated Loading Options

#### WITH ISOLATED LOADING

The INSERT can be performed as a concurrent load isolated operation on a table that is defined with load isolation.

#### NO

The INSERT is not performed as a concurrent load isolated operation.

#### CONCURRENT

Optional keyword that you can include for readability.

### Table Name

#### INTO

Keyword that is required for ANSI compliance. If you do not specify INTO, then the request is non-ANSI standard.

#### *table\_name*

Name of the table directly or by means of a view.

### Values Options

#### VALUES

Keyword required for ANSI compliance.

The colon is optional with host variables used in this clause.

<i>column_name</i>	VALUES Keyword
Specified.	Required.
Not specified.	Optional.

### *expression*

A constant or constant expression to be inserted into the new row for the columns specified in *column\_name*. A constant expression is an expression containing only constants (for example, 3+5, or 19000/12).

If you are inserting into a UDT column, *expression* includes the appropriate NEW expression with the necessary constructor and mutator methods specified using dot notation. For more information, see *Teradata Vantage™ SQL Operators and User-Defined Functions*, B035-1210.

The system values CURRENT\_DATE, DATE, CURRENT\_TIME, TIME, and USER can be included in a constant expression. You can also use the DEFAULT function to insert the default value for a column.

You can insert a NULL by specifying the reserved word NULL as an *expression* value.

Values obtained from imported data, with a USING modifier, or as macro parameters, are accepted as constants.

When a column name list is not specified, then values are inserted in column definition order, from left to right. Use commas to indicate skipped columns. See [Example: Insert Ordered Entries](#).

The derived period column position is skipped when mapping the values to the columns.

When a column name list and an expression list are used, values are assigned by matching *column\_name* position with *expression* position. The two lists must have the same number of items.

When an element of the expression list is omitted, it is treated as a NULL. This is not valid in ANSI SQL.

You cannot specify a value for a derived period column.

### *column\_name*

Name of a column for which the insert operation supplies a new row value.

Column names can be specified in any order.

If a *column\_name* is omitted, any default value defined in the CREATE TABLE or CREATE VIEW request is used.

You cannot specify a derived period column name.

The begin and end columns of a derived period column can be independently updated. The end column value must be greater than the begin column value of the derived period column.

## JSON

Specifies JSON format input. See [Rules for Using the JSON Option](#).

For information about formatting JSON data, see *Teradata Vantage™ JSON Data Type*, B035-1150.

### 'JSON\_string'

Specifies a literal string of data in JSON format.

### ?

Specifies parameterized SQL.

For parameterized SQL, INSERT...JSON supports VARCHAR, CLOB, and external JSON data types. However, the operation is a two-AMP process. For optimal performance, specify a JSON literal.

## Examples: Inserting Data in JSON Format

Below is the table definition for the following example.

```
CREATE TABLE MyTable (
  pkey INTEGER,
  val INTEGER,
  j JSON AUTO COLUMN);
```

This statement inserts the value 10 into the column pkey and the value 1234 into the column val. Because the table does not have a column named extra, the value 1234 is inserted into the auto column j.

```
INSERT INTO MyTable JSON '{"pkey":10,"val":1234,"extra":"1234"}';
```

This statement inserts parameterized SQL, represented by the question mark (?), into the table MyTable for a client that supports parameterized SQL.

```
INSERT INTO MyTable JSON ?;
```

Below is the table definition for the following example.

```
CREATE TABLE LDITable, WITH ISOLATED LOADING (
  pkey INTEGER,
  val INTEGER,
  j JSON AUTO COLUMN);
```

This statement inserts into the load isolated table LDITable.

```
INSERT WITH ISOLATED LOADING INTO LDITable
 '{"pkey":10,"val":1234,"extra":"1234"}';
```

Below is the table definition for the example that follows. The table jsonTable includes the auto column j which must contain a value.

```
CREATE TABLE jsonTable (
  a INTEGER,
  b INTEGER,
  j JSON AUTO COLUMN NOT NULL);
```

These statements insert three rows of data in the table JsonTable.

```
INSERT jsonTable JSON '{"a":1,"b":1,"extra":1}';
```

```
INSERT jsonTable JSON '{"a":2,"b":2,"extra1":2,"extra2":222}';
```

```
INSERT jsonTable JSON '{"a":3,"b":3}';
```

Because the table does not have a column named extra, extra1, or extra 2, the values specified for extra, extra1, and extra2 are inserted into the auto column j. The third insert statement only specifies values for columns a and b. Because column j is defined as NOT NULL, an empty set of brackets ({} ) is inserted in the third row for column j.

```
SELECT * FROM jsonTable ORDER BY 1;
      a      b j
-----
      1      1 {"extra":"1"}
      2      2 {"extra1":"2","extra2":"222"}
      3      3 {}
```

## DEFAULT VALUES

A row consisting of default values is to be added to *table\_name*.

If a DEFAULT phrase is not defined for the column, the system assumes a default of null for it unless it is defined as NOT NULL, in which case an error is returned and the insertion fails.

Embedded SQL does not support this option for INSERT.

## Subquery Options

### *column\_name*

Name of a column for which the insert operation supplies a new row value.

*column\_name* can be specified in any order.

If a *column\_name* is omitted, then any default value defined in the CREATE TABLE or CREATE VIEW request is used.

You cannot specify a derived period column name.

The begin and the end columns of a derived period column can be independently updated. The end column value must be greater than the begin column value of the derived period column.

### **subquery**

The row or rows to be inserted consist of column values accessed by a query specification.

If a column name list is not specified, then the SELECT request must query the same number of columns as there are in the table that is receiving new rows.

You can specify nulls in the select expression list for columns that are not to be assigned values.

You cannot specify an ORDER BY clause as part of a subquery in an INSERT ... SELECT request.

If the INSERT operation includes a column name list, values are assigned by matching *column\_name* position with position of items in the select *expression* list.

### **HASH BY *hash\_list***

Hash rows being inserted into a NoPI table across the AMPs based on the *hash\_list*.

### **HASH BY RANDOM**

Hash rows being inserted into a NoPI table across the AMPs randomly, a block at a time.

### **LOCAL ORDER BY *order\_list***

Order rows being inserted into a table locally on the AMPs.

## **Error Logging Options**

Following are the error logging options you can specify for tables that have an associated error table. See “CREATE ERROR TABLE” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

### **LOGGING ERRORS**

#### **LOGGING ALL ERRORS**

Log all data errors, reference index errors, and USI errors.

If you do not specify the LOGGING ERRORS option, the system does not provide error handling.

If an error occurs, the following session-mode behavior occurs:

- If the current session mode is ANSI, the erring request aborts and rolls back.
- If the current session mode is Teradata, the erring transaction aborts and rolls back.

In certain cases, such as a deadlock, the erring transaction roles back.

The optional keyword ALL is the default. If you log errors, then you must log *all* errors.

#### **WITH NO LIMIT**

There is no limit to the number of errors that can be logged in the error table associated with the target data table for this INSERT ... SELECT load operation.

Errors are logged up to the system-determined limit of 16,000,000 errors.

**WITH LIMIT OF *error\_limit***

The limit on the number of errors that can be logged in the error table associated with the target data table for this INSERT ... SELECT load operation is *error\_limit*,

The value you specify for *error\_limit* can be from 1 through 16,000,000. If you do not specify a value for the LIMIT option, the system defaults to a 10 error limit.

**ANSI Compliance**

INSERT is ANSI SQL:2011-compliant with extensions.

The INS abbreviation is a Teradata extension to the ANSI SQL:2011 standard.

In the ANSI SQL:2011 definition, this statement is named INSERT INTO, while in the Teradata definition, INTO is an optional keyword.

**Usage Notes****Locks and Concurrency**

An INSERT operation sets a WRITE lock for the table being updated at the appropriate level:

- Table-level lock for all rows in the table on all AMPs
- Partition lock for all rows in the selected partitions on all AMPs
- Rowhash for the rows with the same hash value on one AMP or rowkey for the rows with the same partition and hash values on one AMP

If the target table is load isolated and the insert is not an isolated load operation, then an EXCLUSIVE lock is applied at that level, instead of a WRITE lock. If the WITH NO ISOLATED LOADING option is specified or isolated load operations are disabled on the table or in the session, then the insert is not an isolated load operation. For information on the CREATE TABLE WITH ISOLATED LOADING option and the SET SESSION FOR ISOLATED LOADING statement, see *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

The lock set for SELECT subquery operations depends on the isolation level for the session, the setting of the AccessLockForUncomRead DBS Control field, and whether the subquery is embedded within a SELECT operation or within an INSERT request.

Transaction Isolation Level	DBS Control AccessLockForUncomRead Field Setting	Default Locking Severity for Outer SELECT and Ordinary SELECT Subquery Operations	Default Locking Severity for SELECT Operations Embedded Within an INSERT Request
SERIALIZABLE	FALSE	READ	READ
	TRUE		READ



Transaction Isolation Level	DBS Control AccessLockForUncomRead Field Setting	Default Locking Severity for Outer SELECT and Ordinary SELECT Subquery Operations	Default Locking Severity for SELECT Operations Embedded Within an INSERT Request
READ UNCOMMITTED	FALSE		READ
	TRUE		ACCESS

See also:

- “SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ - Database Utilities*, B035-1102

## Inserting using the DEFAULT Function Option, the DEFAULT VALUES Option, or Without Specifying a Value

### DEFAULT Function Option

You can use the DEFAULT function to return, and then insert, the default value for a column based on either its position within the VALUES expression list or, if an explicit column name is specified along with the DEFAULT function, its name.

### DEFAULT VALUES Option

For a table with all columns defined as in the following table, an INSERT defined with the DEFAULT VALUES keywords adds a row consisting of defined default values (where indicated), identity values for identity columns, and nulls for other columns, where defined:

Nullable?	Default defined?
Yes	No
Yes	Yes
No	Yes

This INSERT occurs when the row satisfies the conditions imposed by the table definition. Otherwise an error is returned.

Embedded SQL also supports the DEFAULT VALUES option for INSERT.

When inserting into a table with an identity column, the system always generates a number for the identity column, whether it is defined as GENERATED ALWAYS or as GENERATED BY DEFAULT.

In the following example, if *table\_1* has an identity column, the system always generates a value for it:

```
INSERT table_1 DEFAULT VALUES;
```

When any non-identity column in the table is defined as NOT NULL and does not have defined DEFAULT, an error is returned (ANSI session mode) or the transaction aborts (Teradata session mode).

When a row is otherwise valid but duplicates an existing row and the table has been defined not to accept duplicate rows, an error is returned (ANSI session mode) or the transaction aborts (Teradata session mode).

### Inserting Into Columns Without Specifying a Value

The following rules apply to an INSERT operation that does not assign values for every column in a new row:

- If the column is not declared as NOT NULL and no default value is declared, nulls are inserted.
- If the column is declared as NOT NULL and no default value is declared, an error is returned (ANSI session mode) or the transaction aborts (Teradata session mode).

## Inserting into UDT Columns

### Inserting into Distinct UDT Columns

To insert into a distinct column, either of the following must be true:

- The inserted value must be of the same distinct type as the column.
- There must exist a cast, either system-generated or user-defined, that converts the type of the inserted value to the distinct type of the column, and that cast was created with the AS ASSIGNMENT option. See "CREATE CAST" in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

A distinct value can be constructed from any of the following:

- A cast expression where the target is the same distinct type.
- An instance method invocation that returns the same distinct type.
- A UDF invocation that returns the same distinct type.
- A column reference.
- A null.

For example, suppose you have the following two table definitions:

```
CREATE TABLE table_1 (
  column1 euro,
  column2 INTEGER)
UNIQUE PRIMARY INDEX(column2);
```

```
CREATE TABLE table_2 (
  column1 euro,
  column2 INTEGER)
UNIQUE PRIMARY INDEX(column2);
```

For this case, the information in the following table is true:

Example	Comment
INSERT INTO table_1 VALUES (5.64, 1);	Valid if there is a cast with AS ASSIGNMENT, whether system-generated or user-defined, to convert a DECIMAL type to the distinct type euro.
INSERT INTO table_1 VALUES (CAST(5.64 AS euro), 1);	Valid because an explicit cast is specified for conversion from the source type euro.
INSERT INTO table_1 SELECT CAST(us_dollar_col AS euro), c2 FROM us_sales;	Valid if the cast from us_dollar to euro is valid.
INSERT INTO table_1 SELECT column1.roundup(0), column2 FROM TAB2;	Valid if the method roundup() returns the euro type.
INSERT INTO table_1 SELECT column1, column2 from table_2;	Valid column reference.
INSERT INTO table_1 VALUES (NULL, 8);	Valid because the request insert a null into a distinct type column.

## Inserting into Structured UDT Columns

To insert into a structured type column, either of the following must be true:

- The inserted value must be of the same structured type.
- There must exist a cast that converts the type of the inserted value to the structured type of the column, and the cast was created with the AS ASSIGNMENT option. See the CREATE CAST statement in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

A structured type can be constructed from any of the following:

- A NEW specification (constructor method invocation).
- A constructor function invocation.
- A UDF invocation that returns the same structured type.
- Mutator method invocations.
- An instance method invocation that returns the same structured type.
- A cast expression where the target data type is the same structured type.
- A column reference.
- A null.

For example, suppose you have the following two table definitions:

```
CREATE TABLE table_1 (
  column1 address,
  column2 INTEGER)
UNIQUE PRIMARY INDEX(column2);
CREATE TABLE table_2 (
  column1 address,
  column2 INTEGER)
UNIQUE PRIMARY INDEX(column2);
```

For this case, the information in the following table is true:

Example	Comment
INSERT INTO table_1 VALUES ('17095 Via Del Campo;92127', 1 );	Valid if there is a cast with AS ASSIGNMENT that converts a string to the structured type <i>address</i> .
USING (addr VARCHAR(30), c2 INTEGER) INSERT INTO table_1 (:addr, :c2 );	Valid if there is a cast with AS ASSIGNMENT that converts a string to the structured type <i>address</i> .
INSERT INTO table_1 VALUES (NEW address('17095 Via Del Campo', '92127'), 1 );	Valid invocation of a constructor method.
USING (street varchar(20), zip char(5)) INSERT INTO TAB1 VALUES (NEW address(:street, :zip), 2 );	Valid invocation of a constructor method with host variables.
INSERT INTO TAB1 VALUES ( NEW address(), 3 );	Valid invocation of a constructor function.
INSERT INTO TAB1 VALUES ( NEW address().street('17095 Via Del Campo').zip('92127'), 4 );	Valid invocations of mutator methods: <ol style="list-style-type: none"> <li>1. The constructor function is invoked. The result is an address value whose attribute values are set to their defaults.</li> <li>2. The mutator for the <i>street</i> attribute is invoked. The result is an updated address value with its street attribute modified.</li> <li>3. The mutator for the <i>zip</i> attribute is invoked. The result is another updated address value with its zip attribute modified. This result also contains the previous change to the street attribute.</li> <li>4. The result address value is inserted into column1 of <i>table_1</i>.</li> </ol>

Example	Comment
<pre>INSERT INTO table_1   SELECT empl.residence(), table_2.column2   FROM table_2, table_3;</pre>	Valid if method <i>empl.residence()</i> returns the address data type.
<pre>INSERT INTO table_1   VALUES (SELECT CAST(intl_addr_col AS     address), c2)   FROM table_x;</pre>	Valid if the cast from <i>intl_addr</i> to the structured type address is valid.
<pre>INSERT INTO table_1   SELECT column1, column2   FROM table_2;</pre>	Valid column reference.
<pre>INSERT INTO table_1   VALUES (NULL, 8);</pre>	Valid insertion of a null into a structured type column.

## Inserting into Row-Partitioned Tables, Global Temporary Tables, and NoPI Tables

### Inserting into Row-Partitioned Tables

The section lists the rules for inserting rows into row-partitioned tables.

- Inserting rows into empty row partitions is optimized to avoid transient journaling of each row inserted into those partitions. This optimization can only be applied to tables that are not defined with referential integrity relationships.
- The outcome of partitioning expression evaluation errors, such as divide-by-zero errors, depends on the session mode.

Session Modework unit	Work Unit that Expression Evaluation Errors Roll Back
ANSI	Request that contains the aborted request.
Teradata	Transaction that contains the aborted request.

- When inserting rows into a single-level row-partitioned table for which the partitioning expression is not a *RANGE\_N* or *CASE\_N* function, the partition expression must result in a nonnull value between 1 and 65,535 after casting to *INTEGER*, if the data type is not *INTEGER*.
- When inserting rows into a row-partitioned table, any of the partitioning expressions (which must be *RANGE\_N* or *CASE\_N* functions) for that row must result in a nonnull value.
- When inserting rows into a base table that cause an insert into a row-partitioned join index, any of the partitioning expressions for that join index must result in a nonnull value.
- When inserting rows into a base table that cause an insert into a single-level row-partitioned join index, for a partitioning expression that is not *RANGE\_N* or *CASE\_N*, the result of the partitioning

expression must be a nonnull value between 1 and 65,535 after casting to INTEGER, if the data type is not INTEGER.

- When inserting rows into a base table that cause an update of a join index row in a row-partitioned join index, any of the partitioning expressions for that updated index row must result in a nonnull value.
- When inserting rows into a base table that cause an update of a join index row in a row-partitioned join index with single-level row partitioning where the partitioning expression is not RANGE\_N or CASE\_N, the result of the partitioning expression must be a nonnull value between 1 and 65,535 after casting to INTEGER, if the data type is not INTEGER.
- You cannot assign either a value or a null to the system-derived columns PARTITION or PARTITION#L1 through PARTITION#L62 in an insert operation.
- The session mode and session collation at the time the row-partitioned table was created does not have to match the current session mode and collation for the insert operation to succeed. This is because the partition a row is to be inserted into is determined by evaluating the partitioning expression on partitioning column values using the table's session mode and collation.
- Collation has the following implications for inserting rows into tables defined with a character partitioning:
  - If the collation for a row-partitioned table is either MULTINATIONAL or CHARSET\_COLL, and the definition for the collation has changed since the table was created, Teradata Database aborts any request that attempts to insert a row into the table and returns an error to the requestor.
  - If a noncompressed join index with character partitioning under either an MULTINATIONAL or CHARSET\_COLL collation sequence is defined on a table, and the definition for the collation has changed since the join index was created, the system aborts any request that attempts to insert a row into the table and returns an error to the requestor whether the insert would have resulted in rows being modified in the join index or not.
- If a partitioning expression for a table or noncompressed join index involves Unicode character expressions or literals, and the system has been backed down to a release that has Unicode code points that do not match the code points that were in effect when the table or join index was defined, Teradata Database aborts any attempts to insert rows into the table and returns an error to the requestor.

## Inserting into Global Temporary Tables

Inserting into a global temporary table creates an instance of that table in the current session.

## Inserting into NoPI Tables

You can use INSERT to insert rows into NoPI tables. Teradata Parallel Data Pump operations use Array INSERT on NoPI tables to load data. You can also perform batch inserts of data into NoPI tables using INSERT ... SELECT operations. Inserts are transient journaled in the same way they are for primary indexed tables. For more information about NoPI tables, see *Teradata Vantage™ - Database Design*, B035-1094.

The following INSERT examples are based on these table definitions:

```
CREATE TABLE sales,
FALLBACK (
  ItemNbr  INTEGER NOT NULL,
  SaleDate DATE FORMAT 'MM/DD/YYYY' NOT NULL,
  ItemCount INTEGER)
PRIMARY INDEX (ItemNbr);
CREATE TABLE newsales,
FALLBACK (
  ItemNbr  INTEGER NOT NULL,
  SaleDate DATE FORMAT 'MM/DD/YYYY' NOT NULL,
  ItemCount INTEGER)
NO PRIMARY INDEX;
```

For INSERT requests, the system randomly selects an AMP to send the row or rows. This is true for simple requests like the one below, and more complex requests that involve the insertion of multiple arrays of rows.

```
INSERT INTO newsales (100, '11/01/2007', 10);
```

The AMP then converts the row or array of rows into the proper internal format and appends them to the end of the target table (newsales in this example).

When inserting data from a source table into a NoPI target table using an INSERT ... SELECT request like the one below, data from the source table is not redistributed. The data is locally appended to the target table.

If the SELECT is constrained from a source table, and the rows returned from the AMPs are skewed, the NoPI or NoPI column-partitioned table can become skewed, because the set of rows is locally inserted into the table without redistribution.

```
INSERT INTO newsales
SELECT *
FROM sales;
```

## Rules for Using HASH BY or LOCAL ORDER BY to Insert Rows

The following rules apply:

- You can only specify a HASH BY clause if the target table or the underlying target view table is a NoPI table. If the table has a primary index or Primary AMP index, the system returns an error or failure message, depending on the session mode.
- If you do not specify a HASH BY clause, the system does not redistribute the source rows before inserting into the target NoPI table.

If a NoPI target table has row partitioning that is different than the source table, the system sorts the source rows locally on the internal partition number, then copies the rows locally into the target table. The internal partition number is computed based on the row-partitioning of the target table.

- If you specify a HASH BY clause with a hash list, the system first redistributes the selected rows by a hash value based on the hash list.

If you also specify a LOCAL ORDER BY clause or if the target table is row-partitioned, the system orders the rows locally and inserts them locally into the target table or underlying target view table.

This is useful if the result of the subquery does not provide an even distribution of the rows. If the target table or underlying target view table is also column-partitioned, the locally ordered hash redistribution might also be useful to distribute equal values of a column to the same AMP, which might then enable effective autocompression of the column partitions with the columns on which the hash value is calculated.

Because the object you specify for a HASH BY clause is an expression, the expression can be a function call such as `RANDOM(n, m)`.

`HASH BY RANDOM(1, 2000000000)` is useful to redistribute each individual selected row when there is no particular column set on which to hash distribute the rows, and when a more even distribution is needed than the HASH BY RANDOM clause provides.

A poor choice of a hash expression can lead to a very uneven distribution, similar to when a poor choice is made for primary index columns.

- If you specify a HASH BY RANDOM clause, the system first redistributes data blocks of selected rows randomly.

If you also specify a LOCAL ORDER BY clause, the system orders the rows locally and inserts them locally into the target table or underlying target view table.

This is useful if the result of the subquery does not provide an even distribution of rows. Distributing data blocks is more efficient than distributing individual rows and usually provides a nearly even distribution. However, distributing individual rows using an expression like `HASH BY RANDOM(1, 2000000000)` can provide a more even distribution of rows, which might be necessary in some cases.

- You can specify a LOCAL ORDER BY clause if the target table or the underlying target view table is a PI, PA, or NoPI table. If you specify a LOCAL ORDER BY clause and the target table is row partitioned, the system orders the selected rows locally according to their internal partition numbers computed based on the row-partitioning of the target table, with the column partition number (if any), as 1, (if the target table or underlying target view table is column partitioned) using the partitioning of the target table or underlying target view table if it has row partitioning, and then the ordering expressions after they have been redistributed if you have also specified a HASH BY clause. The final step is to insert the rows locally into the target table or underlying target view table.

If the target table or underlying target view table is also column-partitioned, this might allow for more effective autocompression of the column partitions with the columns on which the ordering is done.



- Teradata Database first resolves column references specified in a HASH BY hash list to the select expression in the subquery expression list corresponding to a matching column name of the target table or view and, if it does not find a matching column, resolved to a result or underlying column in the subquery per the existing rules of resolving column references in an ORDER BY clause for a SELECT request.

## Multistatement and Iterated INSERT Requests

Teradata Database provides statement independence for multistatement INSERT requests and iterated INSERT requests. Statement independence enables multistatement or iterated INSERT requests to roll back only the statements that fail within a transaction or multistatement request and not all of the individual statements within the transaction.

These forms of multistatement requests support statement independence:

- INSERT; INSERT; INSERT;
- BT; INSERT; INSERT; ET;
- BT; INSERT; INSERT;
- INSERT; INSERT; ET;
- INSERT; COMMIT;
- INSERT; INSERT; COMMIT;

---

### Note:

The client software you are using must support statement independence to prevent all multistatement requests and iterated INSERT requests in a transaction or multistatement request from being rolled back. Refer to the documentation for your client software for information on support for statement independence.

---

Most statement data errors resulting from multistatement INSERT requests only roll back failed requests within the transaction or multistatement request, but do not abort all of the statements within the transaction or request unless every INSERT request within the multistatement request aborts because of data-related errors. Teradata Database does not roll back the transaction itself if this happens, only the INSERT requests it contains.

Statement independence supports the following multistatement INSERT data error types:

- Column-level CHECK constraint violations
- Data translation errors
- Duplicate row errors for SET tables
- Primary index uniqueness violations
- Referential integrity violations
- Secondary index uniqueness violations

Statement independence is not enabled for multistatement INSERT requests into tables defined with the following options:

- Triggers
- Hash indexes
- Join indexes

Statement independence is also not supported for the following SQL features:

- INSERT ... SELECT requests
- SQL stored procedures

## Inserting When Using a DEFAULT Function

The following rules apply when using a DEFAULT function to insert rows into a table:

- The DEFAULT function takes a single argument that identifies a relation column by name. The function evaluates to a value equal to the current default value for the column. For cases where the default value of the column is specified as a current built-in system function, the DEFAULT function evaluates to the current value of system variables at the time the request is executed.

The resulting data type of the DEFAULT function is the data type of the constant or built-in function specified as the default unless the default is NULL. If the default is NULL, the resulting data type of the DEFAULT function is the same as the data type of the column or expression for which the default is being requested.

- The DEFAULT function has two forms. It can be specified as DEFAULT or DEFAULT (*column\_name*). When no column name is specified, the system derives the column based on context. If the column context cannot be derived, the request aborts and an error is returned to the requestor.
- The DEFAULT function without a column name can be specified in the expression list. If the INSERT request has a column list specified, the column name for the DEFAULT function is the column in the corresponding position of the column list. If the request does not have a column name list, the column name is derived from the position in the VALUES list.
- The DEFAULT function without a column name in the INSERT request cannot be a part of an expression; it must be specified as a standalone element. This form of usage is ANSI compliant.
- The DEFAULT function with a column name can be specified in an expression list. This is a Teradata extension to ANSI.
- The DEFAULT function with a column name can be specified anywhere in the expression. This is a Teradata extension.
- When there is no explicit default value associated with the column, the DEFAULT function evaluates to null.

For more information about the DEFAULT function, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

## Using INSERT ... SELECT With Tables That Have Row-Level Security

You can use INSERT ... SELECT requests with tables that have row-level security if all the tables have exactly the same row-level security constraints.

Teradata Database does not execute the security policy UDF for row-level security constraints on the target table.

- If the session executing the request does not have the appropriate OVERRIDE privilege to use the DML statement on the target table, Teradata Database takes the values for all row-level security constraint columns from the source table.
- If the session has the appropriate OVERRIDE privilege, Teradata Database takes the constraint values from the source table unless they are provided as part of the INSERT ... SELECT request.

### Example: Application of Row-Level Security INSERT Constraint Functions for Single Statement INSERT Requests

This example shows how the INSERT constraint functions are applied for a single-statement INSERT request on a table that has the row-level security INSERT constraint.

An EXPLAIN statement is used to show the steps involved in the execution of the request and the outcome of the application of the constraint functions.

The statement used to create the table in this example is:

```
CREATE TABLE rls_tbl(
  col1 INT,
  col2 INT,
  classification_levels CONSTRAINT,
  classification_categories CONSTRAINT);
```

The user's sessions constraint values are:

```
Constraint1Name LEVELS
Constraint1Value 2
Constraint3Name CATEGORIES
Constraint3Value '90000000'xb
```

This EXPLAIN statement is used to show the steps involved in the execution of the INSERT request and the outcome of the application of the INSERT constraint functions.

```
EXPLAIN INSERT rls_tbl(1,1,,);
```

The system returns this EXPLAIN text.

```
*** Help information returned. 6 rows.
*** Total elapsed time was 1 second.
Explanation
```

```
----- 1)
First, we do an INSERT into RS.rls_tbl constrained by (
  RS.rls_tbl.levels = SYSLIB.INSERTLEVEL (2)), (
  RS.rls_tbl.categories = SYSLIB.INSERTCATEGORIES ('90000000'XB)).
The estimated time for this step is 0.07 seconds.
-> No rows are returned to the user as the result of statement 1.
The total estimated time is 0.07 seconds.
```

### Example: Row-Level Security INSERT and SELECT Constraints When User Lacks Required Privileges (INSERT...SELECT Request)

This example shows how the INSERT and SELECT constraints are applied when a user that does not have the required OVERRIDE privileges attempts to execute an INSERT...SELECT request on a table that has the row-level security INSERT and SELECT constraints.

The statements used to create the tables in this example are:

```
CREATE TABLE rls_src_tbl(
  col1 INT,
  col2 INT,
  classification_levels CONSTRAINT,
  classification_categories CONSTRAINT);

CREATE TABLE rls_tgt_tbl(
  col1 INT,
  col2 INT,
  classification_levels CONSTRAINT,
  classification_categories CONSTRAINT);
```

The user's sessions constraint values are:

```
Constraint1Name LEVELS
Constraint1Value 2
Constraint3Name CATEGORIES
Constraint3Value '90000000'xb
```

Following is the INSERT statement:

```
INSERT rls_tgt_tbl SELECT * FROM rls_src_tbl;
```

An EXPLAIN shows the INSERT and SELECT constraints applied during RETRIEVE step from rls\_src\_tbl with a condition of ("((SYSLIB.SELECTCATEGORIES ( '90000000'XB, RS.rls\_src\_tbl.categories ))= 'T') AND ((SYSLIB.SELECTLEVEL (2, rls\_src\_tbl.levels ))= 'T')").

## INSERT, DEFAULT Function, PERIOD Value Constructor, Scalar UDFs, UDTs, and Stored Procedures

### Using INSERT ... SELECT With a DEFAULT Function

The following rules apply when using a DEFAULT function to load a table using an INSERT ... SELECT operation:

- All of the rules listed for [DEFAULT Function in SELECT Statements](#) also apply the SELECT subquery in an INSERT ... SELECT request.
- The DEFAULT function cannot be specified without a column name as its argument within the SELECT subquery of an INSERT ... SELECT request.

For more information about the DEFAULT function, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

### Using a PERIOD Value Constructor With INSERT

For the rules on using PERIOD value constructors, see *Teradata Vantage™ SQL Date and Time Functions and Expressions*, B035-1211. For examples of how to use PERIOD value constructors in INSERT requests, see [Example: INSERT Using a PERIOD Value Constructor](#).

### INSERT and Scalar UDFs

You can specify a scalar UDF as a column value in the VALUES clause of an INSERT request. The rules for the invocation of a scalar UDF in a VALUES clause are as follows.

- A scalar UDF that passes a value in the VALUES list of an INSERT request must return a value expression.
- The arguments passed to a scalar UDF must be constants, USING values, or parameters that resolve to a constant.

### INSERT, UDTs, and Stored Procedures

The following rules apply to insert operations, UDTs, and stored procedures:

- You can insert into a UDT local variable of a stored procedure.
- You can insert a UDT local variable into a table.

### Inserting Rows into Queue Tables

The first column of a queue table is defined as a Queue Insertion TimeStamp (QITS) column. The values in the column determine the order of the rows in the queue, resulting in approximate first-in-first-out (FIFO) ordering.

If you want the QITS value of a row to indicate the time that the row was inserted into the queue table, then you can use the default value, the result of CURRENT\_TIMESTAMP, instead of supplying a value.

If you want to control the placement of a row in the FIFO order, you can supply a `TIMESTAMP` value for the `QITS` column.

For a multistatement request containing multiple `INSERT` requests that do not supply values for the `QITS` column, the `QITS` values are the same for every row inserted.

If you want unique `QITS` values for every row in a queue table, you can do any of the following things:

- Supply a `TIMESTAMP` value for the `QITS` column in every `INSERT` request.
- Avoid multistatement requests containing multiple `INSERT` statements that do not supply values for the `QITS` column.
- Add incremental offsets to the current timestamp for the `QITS` column value in each `INSERT` request.

For example:

```
INSERT shopping_cart(CURRENT_TIMESTAMP + INTERVAL '0.001', 100)
;INSERT shopping_cart(CURRENT_TIMESTAMP + INTERVAL '0.002', 200)
;INSERT shopping_cart(CURRENT_TIMESTAMP + INTERVAL '0.003', 300);
```

Regarding performance, an `INSERT` operation into a queue table has the following effects:

- Does not affect response time when the system is not CPU-bound.
- Is more expensive than an `INSERT` into a base table because it requires the update of an internal in-memory queue.

For details on queue tables and the queue table cache, see “`CREATE TABLE`” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## Inserting Into Queue Tables Using Iterated Requests

If you use an `INSERT` request in an iterated request to insert rows into a queue table, you might have to limit the number of data records with each request to minimize the number of rowhash-level `WRITE` locks placed on the table and reduce the likelihood of deadlocks occurring because of resource conflicts between the locks and the all-AMPs table-level `READ` lock exerted by the internal row collection processing used by queue tables to update the internal queue table cache.

IF you use an <code>INSERT</code> request in an iterated request to insert rows into a queue table and ...	THEN ...
all of the following conditions are true: <ul style="list-style-type: none"> <li>• the queue table is not empty</li> <li>• either an <code>INSERT</code> request or a <code>SELECT AND CONSUME</code> request has already been performed from the queue table since the last system reset</li> <li>• rows are not updated or deleted from the queue table during the insert operation</li> </ul>	the number of data records that you pack with each request is not an issue.
any of the following conditions are true: <ul style="list-style-type: none"> <li>• the queue table is empty</li> </ul>	pack a maximum of four data records with each request. For example, if you use <code>BTEQ</code> to import rows of data into a queue table, use a

IF you use an INSERT request in an iterated request to insert rows into a queue table and ...	THEN ...
<ul style="list-style-type: none"> <li>neither an INSERT request nor a SELECT AND CONSUME request has already been performed from the queue table since the last system reset</li> <li>rows are updated or deleted from the queue table during the insert operation</li> </ul>	<p>maximum value of 4 with the BTEQ .SET PACK command.</p> <p>These conditions trigger the internal row collection processing used by queue tables to update the internal queue table cache.</p>

For details on queue tables and the queue table cache, see “CREATE TABLE” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## Valid and Invalid INSERT Operations

### Valid INSERT Operations

An INSERT operation does not return an error message if either of the following occurs:

- In Teradata session mode, the operation attempts to insert a character string that is longer or shorter than that declared for the column. The string is automatically adjusted and inserted. This could result in improper strings for the Kanji1 character data type.
- The operation uses a query and no rows are returned.

### INSERT Operations That Are Not Valid

An INSERT operation causes an error or failure message to be returned if any of the following are true.

- The operation attempts to assign a value that will result in a violation of a unique index specification.
- The operation attempts to insert a row with no value for a column that has no default and is defined as NOT NULL.
- The operation attempts to assign a nonnull value that violates a CHECK constraint declared for a column.
- The operation attempts to assign a value that is of a different numeric type than that declared for the column and the assigned value cannot be converted correctly.
- The operation attempts to assign a character value that is not in the repertoire of the destination character data type.
- The operation attempts to insert a character string trailing pad characters into a VARCHAR field, and that operation causes the row to become identical to another row (except for the number of trailing pad characters).
- In ANSI session mode, inserting character data, if in order to comply with maximum length of the target column, non-pad characters are truncated from the source data.

**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.

**Large Objects and INSERT**

The behavior of truncated LOB inserts differs in ANSI and Teradata session modes. The following table explains the differences in truncation behavior.

Session Mode	Result When Non-Pad Bytes are Truncated on Insertion
ANSI	An exception condition is raised. The INSERT fails.
Teradata	Exception condition is not raised. The INSERT succeeds: the truncated LOB is stored.

**Duplicate Rows and INSERT**

When an insert operation would create a duplicate row, the outcome of the operation depends on how the table is defined. The system ignores trailing pad characters in character strings when comparing values for field or row duplication.

Table Definition	Duplicate Row Insert Action
MULTISET with no UNIQUE constraints	Inserted.
<ul style="list-style-type: none"> <li>SET</li> <li>MULTISET with UNIQUE constraints</li> </ul>	<p>Not inserted. An error is returned to the requestor.</p> <p>If inserting into an identity column table, this is true only when the column is defined WITH NO CYCLE. An error can occur in this case if an attempt is made to cycle. The system can also return errors for other reasons, like uniqueness violations, before a duplicate row violation can be reported.</p>

**Duplicate Rows and INSERT ... SELECT**

If an INSERT using a SELECT subquery will create duplicate rows, the result depends on the table definition:

- Duplicate rows are permitted in a MULTISET set only if no UNIQUE constraints or UNIQUE indexes are defined on the table.
- Duplicate rows are not permitted in a SET table.



The following table summarizes the restrictions on duplicate rows and INSERT ... SELECT:

Type of Table	Duplicate Rows
MULTISET with no unique constraints	Permitted. Inserted duplicate rows are stored in the table.
MULTISET with unique constraints	<p>not permitted. An error message is returned to the requestor. The following constraints are considered to be unique in Teradata Database:</p> <ul style="list-style-type: none"> <li>• Unique primary index</li> <li>• Unique secondary index</li> <li>• Primary key</li> <li>• UNIQUE column constraint</li> <li>• GENERATED ALWAYS identity column constraint.</li> </ul> <p>For nontemporal tables, unique secondary indexes, primary keys, and UNIQUE constraints are all implemented internally as unique secondary indexes. For information about how USIs, PKs, and UNIQUE constraints are implemented for temporal tables, see <i>Teradata Vantage™ ANSI Temporal Table Support</i>, B035-1186 and <i>Teradata Vantage™ Temporal Table Support</i>, B035-1182.</p>
SET	<p>Not permitted.</p> <ul style="list-style-type: none"> <li>• In ANSI session mode, the system rejects the request and returns an error message to the requestor. If some other error, such as violating a uniqueness constraint, occurs first, then the system returns that error to the requestor rather than a duplicate row message</li> <li>• In Teradata session mode, Teradata Database does the following. Rejects the duplicate rows in the transaction. Inserts the non-duplicate rows into the table. Does not return an error message to the requestor.</li> </ul>

In Teradata session mode, if an INSERT ... SELECT request specifies a SAMPLE clause that selects a set of rows from a source MULTISET table, and then inserts them into a target SET table, and the sampled row set contains duplicates, the number of rows inserted into the target SET table might be fewer than the number requested in the SAMPLE clause.

In Teradata session mode, the condition occurs because SET tables reject attempts to insert duplicate rows into them. The result is that the INSERT portion of the INSERT ... SELECT operation inserts only distinct rows into SET target tables. As a result, the number of rows inserted into the target table can be fewer than the number specified in the SAMPLE clause.

For example, if an INSERT ... SELECT request SAMPLE clause requests a sample size of 10 rows, and there are duplicate rows from the MULTISET source table in the collected sample, the system rejects the duplicate instances when it attempts to insert the sampled rows into the SET table and inserts only the distinct rows from the sample set. That is, you could request a sample of 10 rows, but the actual number of rows inserted into the target table could be fewer than 10 if there are duplicate rows in the sampled row set. Teradata Database does not return any warning or information message when this condition occurs.

## INSERT Process

An INSERT process performs the following actions:

1. Sets a WRITE lock on the rowkey, partition, or table, as appropriate.
2. Performs the entire INSERT operation as an all-or-nothing operation in which every row is inserted successfully or no rows are inserted.

This is to prevent a partial insert from occurring.

Session Mode	Unsuccessful INSERT Result
ANSI	Rolls back the erring request only.
Teradata	Rolls back the entire containing transaction

The rules for rolling back multistatement INSERT requests for statement independence frequently enable a more relaxed handling of INSERT errors within a transaction or multistatement request. For information about failed INSERT operations in situations that involve statement independence, see [Multistatement and Iterated INSERT Requests](#).

The INSERT operation takes more processing time on a table defined with FALLBACK or a secondary, join, or hash index, because the FALLBACK copy of the table or index also must be changed.

## Inserting Rows through Views

Use caution when granting the privilege to insert data through a view because data in fields not visible to the user might be inserted when a row is inserted through a view.

The following rules apply to inserting rows through a view:

- Both you and the immediate owner of the view must have the appropriate privileges.
- The view must reference columns in only one table or view.
- None of the columns in the view can be derived by using an expression to change values in the underlying table.
- Each column in the view must correspond one to one with a column in the underlying table or view.
- The view must include any column in the underlying table or view that is declared as NOT NULL.
- No two view columns can reference the same column in the underlying table.
- If the request used to define a view contains a WHERE clause, and WITH CHECK OPTION, all values inserted through that view must satisfy constraints specified in the WHERE clause.

If a view includes a WHERE clause but does not include the WITH CHECK OPTION, then data can be inserted that is not visible through that view.

## Subqueries in INSERT Requests

An INSERT operation that uses a subquery, referred to as an INSERT ... SELECT request, differs from a simple INSERT in that many rows can be inserted in a single operation, and the row values can come from more than one table.

The query specification must always include a FROM *table\_name* clause.

Also, an INSERT request that includes a subquery must define a column name list when the following conditions exist:

- The number of columns listed in the query specification differs from the number of columns in the table receiving new rows.
- The order of columns listed in the query specification differs from the order the columns were defined in the table receiving new rows.

### Using INSERT ... SELECT With Subqueries

You cannot specify an ORDER BY clause in the SELECT component of an INSERT ... SELECT request when the SELECT is a subquery.

### Using Scalar Subqueries in INSERT Requests

The following rules and restrictions apply to specifying scalar subqueries in INSERT and INSERT ... SELECT requests:

- You can specify a scalar subquery as a parameterized value in the value list of a simple INSERT request, but Teradata Database always interprets it as a noncorrelated scalar subquery. See [Example: INSERT Using a Scalar Subquery](#).
- You cannot specify a noncorrelated scalar subquery as a value in a value list that is assigned to an identity column in a simple INSERT request.
- You can specify an INSERT request with scalar subqueries in the body of a trigger.

However, you cannot specify a simple INSERT request with a scalar subquery in its value list in the body of a row trigger.

- You can specify a scalar subquery in the SELECT component of an INSERT ... SELECT request.
- Teradata Database processes any noncorrelated scalar subquery specified in the SELECT component of an INSERT ... SELECT in a row trigger as a single-column single-row spool instead of as a parameterized value.

## SELECT AND CONSUME Subqueries in INSERT Requests

An INSERT operation can use a SELECT AND CONSUME subquery to insert data from the row with the lowest value in the QITS column in the specified queue table and delete the row from the queue table.

The target table for the INSERT operation can be a base table or a queue table. If the target is a queue table, it can be the same queue table in the SELECT AND CONSUME subquery.

The SELECT AND CONSUME part of the request behaves like a regular SELECT AND CONSUME request. See [SELECT AND CONSUME](#).

When the target table is a queue table, the INSERT part of the request behaves like an INSERT into a queue table. See [Inserting Rows into Queue Tables](#).

Certain restrictions apply to triggers. An INSERT operation that uses a SELECT AND CONSUME subquery cannot be used as either of the following:

- Triggering event statement that fires a trigger.
- Triggered action statement fired by a triggering event.

## Data Takes the Attributes of the New Table

If the column attributes defined for a new table differ from those of the columns whose data is being inserted using INSERT, then the data takes on the attributes of the new table.

The source column names can differ from the destination column names in the new table, but the data is inserted correctly as long as the SELECT request lists the source column names in the same order as the corresponding destination columns in the CREATE TABLE request. This is true even if the new table has a column that is to contain data derived (either arithmetically or by aggregate operation) from the column data in the existing table.

## Logging Errors for INSERT ... SELECT Requests

Normally, an INSERT ... SELECT request with error logging completes without any USI or RI errors. Exceptions to normal completion of a INSERT ... SELECT request are processed as follows:

- Not all types of errors are logged when you specify the LOGGING ERRORS option for an INSERT ... SELECT request.
  - All local, or data errors, are logged.
  - Errors that occur before the row merge step, such as data conversion errors detected in the RET AMP step before the MRG or MRM AMP steps, are not logged. See *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

These are errors that occur during row merge step processing, such as CHECK constraint, duplicate row, and UPI violation errors.

- When Teradata Database encounters USI or RI errors (or both) in the INSERT ... SELECT operation, the following events occur in sequence:
  1. The transaction or request runs to completion.
  2. The system writes all error-causing rows into the error table.
  3. The system aborts the transaction or request.

4. The system rolls back the transaction or request.

Note that the system does not invalidate indexes, nor does it roll error table rows back, enabling you to determine which rows in the INSERT set are problematic and to determine how to correct them.

If the number of errors in the request is large, running it to completion plus rolling back all the INSERT operations can exert an impact on performance. To minimize the potential significance of this problem, you should always consider either using the default limit or specifying a `WITH LIMIT OF error_limit` clause with a relatively small value for *error\_limit*. In other words, unless you have a good reason for doing so, you should avoid specifying `WITH NO LIMIT`.

- When an INSERT ... SELECT operation encounters data errors only, their occurrence does not abort the transaction or request, the non-erring INSERT operations complete successfully, and the erring rows are logged in the error table so you can correct them.

The following rules and guidelines apply to logging errors in an error table for INSERT ... SELECT loads:

- Before you can log errors for INSERT ... SELECT loads, you must first create an error table for the base data table into which you intend to do an INSERT ... SELECT load. See “CREATE ERROR TABLE” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.
- If error logging is not enabled and you submit an INSERT ... SELECT bulk loading operation with the LOGGING ERRORS option specified, the system aborts the request and returns an error message to the requestor.
- The LOGGING ERRORS option is not valid in a multistatement request.
- Two basic types of errors can be logged when you specify the LOGGING ERRORS option:

- Local errors

Local errors are defined as errors that occur on the same AMP that inserts the data row. The following types of errors are classified as local errors:

- Duplicate row errors, which occur only in ANSI session mode.

The system silently ignores duplicate row errors that occur from a INSERT ... SELECT into a SET table in Teradata session mode.

Duplicate rows can also arise from these INSERT ... SELECT insert situations:

- The source table has duplicate rows.
- An insert is made on a different AMP than the failed update.
- Duplicate primary key errors
- CHECK constraint violations

- Nonlocal errors

Nonlocal errors are defined as errors that occur on an AMP that does not own the data row. The following types of errors are classified as nonlocal errors:

- Referential integrity violations
- USI violations

An exception to this is the case where a USI violation is local because the USI is on the same set of columns as the primary index of a row-partitioned table. The system treats such an error as a nonlocal error, even though it is local in the strict definition of a local error.

The system records the error in the error table, rejects the error-causing rows from the target table, and performs the actions described in the table below, depending on the type of error.

Type of Error	Response
Local	Completes the request or transaction successfully.
Nonlocal	Allows the request or transaction to complete in order to record all of the error causing rows in the INSERT ... SELECT load, then aborts the request or transaction and rolls back its inserts and updates.
Local and nonlocal	

- The system does not handle batch referential integrity violations for INSERT ... SELECT error logging. Because batch referential integrity checks are all-or-nothing operations, a batch referential integrity violation causes the system to respond in the following session mode-specific ways:

Session Mode	Result
ANSI	Request aborts and rolls back.
Teradata	Transaction aborts and rolls back.

- The system does not handle error conditions that do not allow useful recovery information to be logged in the error table. Such errors typically occur during intermediate processing of input data before it are built into a row format that corresponds to the target table.

The system detects this type of error before the start of data row inserts and updates. The following are examples of these types of error:

- UDT, UDF, and table function errors
- Version change errors
- Nonexistent table errors
- Down AMP request against nonfallback table errors
- Data conversion errors

Conversion errors that occur during row inserts are treated as local data errors.

The way the system handles these errors depends on the current session mode:

Session Mode	Result
ANSI	Request aborts and rolls back.
Teradata	Transaction aborts and rolls back.

The system preserves error table rows logged by the aborted request or transaction and does not roll them back.

The system inserts a marker row into the error table at the end of a successfully completed INSERT ... SELECT request with logged errors.

Marker rows have a value of 0 in the ETC\_ErrorCode column of the error table, and their ETC\_ErrSeq column stores the total number of errors logged. All other columns in a marker row except for ETC\_DBQL\_QID and ETC\_TimeStamp are set to null.

If no marker row is recorded, the request or transaction was aborted and rolled back because of one or more of the following reasons:

- The specified error limit was reached.
- The system detected an error that it cannot handle.
- The system detected a nonlocal (RI or USI) violation.

Teradata Database preserves the error rows that belong to the aborted request or transaction.

- In addition to the previously listed errors, the system does not handle the following types of errors, though it preserves logged error table rows if any one of the listed errors is detected:
  - Out of permanent space or out of spool space errors
  - Duplicate row errors in Teradata session mode (because the system ignores such errors in Teradata session mode)
  - Trigger errors
  - Join index maintenance errors
  - Identity column errors

- The LOGGING ERRORS option is applicable to INSERT ... SELECT load requests whose target tables are permanent data tables only.

Other kinds of target tables, such as volatile and global temporary tables, are not supported.

Teradata Database returns a warning message to the requestor if it logs an error.

- You can either specify logging of all errors or logging of no errors. This means that you cannot specify the types of errors to log. The WITH LIMIT OF *error\_limit* option, of course, enables you to terminate error logging when the number of errors logged matches the number you specify in the optional WITH LIMIT OF *error\_limit* clause.

If you do not specify a LOGGING ERRORS option, and an error table is defined for the target data table of the INSERT ... SELECT request, the system does no error handling for INSERT ... SELECT operations against that data table.

In this case, the request or transaction containing the erring INSERT ... SELECT request behaves as follows when an error occurs:

Session Mode	Result
ANSI	Request aborts and rolls back.

Session Mode	Result
Teradata	Transaction aborts and rolls back.

- If you specify neither the WITH NO LIMIT option, nor the WITH LIMIT OF *error\_limit* option, the system defaults to an error limit of 10.

Teradata Database logs errors up to the limit of 10, and then the request of transaction containing the INSERT ... SELECT request behaves as follows when the tenth error occurs:

Session Mode	Result
ANSI	Request aborts and rolls back.
Teradata	Transaction aborts and rolls back.

Teradata Database preserves error table rows logged by the aborted request or transaction and does not roll them back.

- WITH NO LIMIT

Teradata Database places no limit on the number of error rows that can accumulate in the error table associated with the target data table for the INSERT ... SELECT operation.

- WITH LIMIT OF *error\_limit*

Teradata Database logs errors up to the limit of *error\_limit*, and then the request or transaction containing the INSERT ... SELECT request behaves as follows when the *error\_limit* is reached:

Session Mode	Result
ANSI	Request aborts and rolls back.
Teradata	Transaction aborts and rolls back.

Teradata Database preserves error table rows logged by the aborted request or transaction and does not roll them back.

- The activity count returned for an INSERT ... SELECT ... LOGGING ERRORS request is the same as that returned for an INSERT ... SELECT operation without a LOGGING ERRORS option: a count of the total number of rows inserted into the target data table.
- LOGGING ERRORS does not support LOB data. LOBs in the source table are not copied to the error table. They are represented in the error table by nulls.
- An index violation error does not cause the associated index to be invalidated.
- For referential integrity validation errors, you can use the IndexId value with the RI\_Child\_TablesVX view to identify the violated Reference index. For information about Reference indexes, see *Database Design*.

You can distinguish whether an index error is a USI or referential integrity error by the code stored in the ETC\_IdxErType error table column.



ETC_IdxErrType Value	Error
R	Foreign key insert violation.
r	Parent key delete violation.
U	USI validation error.

## Fast Path INSERT ... SELECT Requests

Multistatement INSERT ... SELECTs are optimized so that each request except for the last in the series returns a 'zero row inserted' message as a response. The retrieved rows for each SELECT are sent to an empty target table. The last INSERT ... SELECT request returns the total number of rows inserted for the entire request and sorts and merges the spool table into the target table.

Columns defined with the COMPRESS option do not participate in fast path optimizations, so if you perform an INSERT ... SELECT operation on compressed columns, fast path optimization is not specified by the Optimizer when it creates an access plan.

### INSERT ... SELECT Performance and Target Table Identity Column Primary Indexes

For those cases where INSERT ... SELECT is optimized so that a direct merge from source to target table is possible, like when source and target tables have the identical structure, there can be as much as a threefold degradation of performance caused by the need for an extra step to spool and redistribute rows when the target table has an identity column primary index.

### Rules for Fast Path INSERT ... SELECT Requests

Observe the following restrictions for the high performance fast path optimization in a multistatement request:

- The target table must be empty.
- All INSERT statements in the multistatement request must have the same target table.
- Only INSERT statements can be included in the request.

If you insert other statement types into the multistatement request, the fast path optimization does not occur (only the first INSERT ... SELECT in the series is optimized) and performance degrades accordingly.

## INSERT in Embedded SQL and Stored Procedures

### General Rules for INSERT in Embedded SQL and Stored Procedures

The following rules apply to both forms of INSERT in embedded SQL and stored procedures.

The row to be inserted is constructed by building a candidate row as follows:

- Define each column value of the row to be NULL.
- Assign each inserted value to the corresponding column.

Result Row Value for Any NOT NULL Column	Insertion Result
Non-null	Succeeds.
Null	Fails. SQLCODE is set to -1002.

Insert values are set in the corresponding row column values according to the rules for defining host variables.

If the table identified by *table\_name* is a view that was defined WITH CHECK OPTION, then the row to be inserted must be in the set of rows selected by the view.

### Valued INSERT in Embedded SQL

When using the valued form of INSERT with embedded SQL, the colon is optional with host variables in the VALUES clause.

### Rules for INSERT ... SELECT in Embedded SQL and Stored Procedures

The following rules apply to using the selected form of INSERT with embedded SQL and stored procedures:

- The number of rows in the temporary table returned by *subquery* determines the number of rows to be inserted.
- If an error occurs after one or more of the selected rows has been inserted, then the value -1002 is returned to SQLCODE and the current transaction is terminated with rollback.
- If *subquery* selects no rows, then the value +100 is returned to SQLCODE and no rows are inserted.

### Inserting into Load Isolated Tables

The RowLoadID value in each row of a load isolated table records the LoadID value. The LoadID used in a newly inserted row is derived from the last committed LoadID value for the table as recorded in DBC.TVM.CurrentLoadId. For more information, see *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

### TD\_ROWLOADID Expression

The TD\_ROWLOADID expression reads the RowLoadID. The data type of the expression is BIGINT with a default format of '-(19)9'.

The TD\_RowLoadID expression is not permitted in a *JI* definition, a partitioning expression, or a CHECK Constraint.

You specify the TD\_RowLoadID expression as follows:

```
database_name.table_name.TD_ROWLOADID
```

**database\_name**

Name of the containing database, if other than the default database.

**table\_name**

Name of a load isolated table.

## Rules for Using the JSON Option

- You can only shred JSON data in text format. You cannot use the INSERT statement to shred JSON data that is in one of the binary formats such as BSON or UBJSON.
- The shredded data is in VARCHAR format and implicit casting is used to convert the VARCHAR data to the target table column format. If the VARCHAR data cannot be CAST to the target column format, the insertion fails. For example, if the target column is VARBYTE, casting values other than NULL to VARBYTE returns an error because JSON does not have a matching textual value for binary data.
- The INSERT statement processes a single row of JSON data with JSON OBJECT at the root. That is, the JSON data starts with '{' as the first non-white space character. You cannot insert multiple rows of data using the INSERT statement.
- The INSERT statement supports Load Isolation options if the target table is an LDI table.
- The target table must be a table and not a view.
- The column name matching is not case sensitive.
- If the same column is matched multiple times, the data stored is the last match.
- If any of the target table columns is NOT NULL, and the JSON input data does not contain any data for the column, the following rules apply:
  - If the target column does not have a DEFAULT value, an error is returned.
  - If the target column has a DEFAULT value and the JSON data is a string literal, the DEFAULT value is inserted into the target table.
  - If the target column has a DEFAULT value, and INSERT...JSON uses parameterized SQL, the DEFAULT value is ignored and an error is returned. In this case, specifying a JSON literal is preferable to using parameterized SQL

## Examples

### Example: Inserting a Row

You can use the following request to insert a row for the new employee named *Smith* in the *employee* table.

```
INSERT INTO employee (name, empno, deptno, dob, sex, edlev)
VALUES ('SMITH T', 10021, 700, 460729, 'F', 16);
```

## Example: Insert Using a SELECT Subquery

This example uses a SELECT subquery to insert a row for every employee who has more than ten years of experience into a new table, *promotion*, which contains three columns defined as *name*, *deptno*, and *yrsexp*.

```
INSERT INTO promotion (deptnum, empname, yearsexp)
SELECT deptno, name, yrsexp
FROM employee
WHERE yrsexp > 10 ;
```

## Example: Insert Using a SELECT Subquery Without Target Column List

The INSERT operation performed in the previous example can also be accomplished by the following statement. Note that a column name list is not given for the *promotion* table; therefore, the *employee* columns referenced in the SELECT subquery must match, in both quantity and sequence, the columns contained by *promotion*.

```
INSERT INTO promotion
SELECT name, deptno, yrsexp
FROM employee
WHERE yrsexp > 10 ;
```

## Example: Insert With Named Columns

To add a row to the *employee* table for new employee *Orebo*, you could submit the following statement:

```
INSERT INTO employee (name, empno, deptno, dob, sex, edlev)
VALUES ('Orebo B', 10005, 300, 'Nov 17 1957', 'M', 18) ;
```

In this example, you list the columns that are to receive the data, followed by the data to be inserted, in the same order as the columns are listed.

If you do not specify a value for a named column, the system inserts a null.

## Example: Insert Ordered Entries

You could achieve the same result with the following statement:

```
INSERT INTO employee (10005, 'Orebo B', 300, , , 'Nov 17 1957',
'M', , , 18, );
```

Using this form, you do *not* specify column names for fields because you enter the field data in the same order as the columns are defined in the *employee* table. In all cases, a comma is required as a place marker for any field for which data is not specified.

For ANSI compliance, you should use the keyword NULL for nulls, for example, (10005, 'Ore60B', 300, NULL, NULL, NULL, 'Nov 17 1957', 'M', NULL, NULL, 18).

## Example: Bulk Insert

This INSERT operation performs a bulk insert. With partition level locking, the parser generates a static partition elimination list based on condition "PC = 4" (that is, positional matching of slppit1.pc = srct1.b = 4) and uses this list to place a PartitionRange lock.

The table definitions for this example are as follows:

```
CREATE TABLE HLSDS.SLPPIT1 (PI INT, PC INT, X INT) PRIMARY INDEX (PI)
  PARTITION BY (RANGE_N(PC BETWEEN 1 AND 100 EACH 10));
CREATE TABLE HLSDS.SRCT1 (A INT, B INT, C INT) PRIMARY INDEX (A);
```

An EXPLAIN of the INSERT operation shows the condition PC=4 as WHERE B=4:

```
Explain INSERT INTO HLSDS.SLPPIT1 SELECT A, B, C FROM HLSDS.SRCT1
WHERE B = 4;
1) First, we lock HLSDS.srct1 for read on a reserved RowHash
   to prevent global deadlock.
2) Next, we lock HLSDS.slppit1 for write on a reserved RowHash
   in a single partition to prevent global deadlock.
3) We lock HLSDS.srct1 for read, and we lock HLSDS.slppit1
   for write on a single partition.
4) We do an all-AMPS RETRIEVE step from HLSDS.srct1 by way of
   an all-rows scan with a condition of ("HLSDS.srct1.b = 4") into
   Spool 1(all_amps), which is built locally on the AMPS.
   Then we do a SORT to partition Spool 1 by rowkey.
   The size of Spool 1 is estimated
   with no confidence to be 1 row (25 bytes).
   The estimated time for this step is 0.03 seconds.
5) We do an all-AMPS MERGE into HLSDS.slppit1 from Spool 1
   (Last Use). The size is estimated with no confidence
   to be 1 row. The estimated time for this step is 0.55 seconds.
6) Finally, we send out an END TRANSACTION step to all AMPS
   involved in processing the request.
```

## Example: INSERT Operation Using Single-Writer Lock

This INSERT operation uses a single-writer lock. The single-writer lock is added for bulk DML operations that require index maintenance on a partition-locked table. The lock prevents concurrent write operations on the table, while allowing concurrent read operations on the partitions that are not locked.

The table definitions for this example are as follows:

```
CREATE TABLE HLSDS.SLPPIT3 (PI INT, PC INT, USI INT, X INT, Y INT)
  PRIMARY INDEX (PI)
```

```

PARTITION BY
    (RANGE_N(PC BETWEEN 1 AND 1000 EACH 1, NO RANGE, UNKNOWN))
UNIQUE INDEX (USI);
CREATE TABLE HLSDS.SRCT3 (A INT, B INT, C INT, D INT, E INT);

```

An EXPLAIN of the INSERT operation shows the condition PC=100 as WHERE B=100:

```

Explain INSERT HLSDS.SLPPIT3 SELECT * FROM HLSDS.SRCT3 WHERE B = 100;
1) First, we lock HLSDS.srct3 for read on a reserved RowHash
   to prevent global deadlock.
2) Next, we lock HLSDS.slppit3 for single writer on a
   reserved RowHash in all partitions to serialize concurrent
   updates on the partition-locked table to
   prevent global deadlock.
3) We lock HLSDS.slppit3 for write on a reserved RowHash
   in a single partition to prevent global deadlock.
4) We lock HLSDS.srct3 for read, we lock HLSDS.slppit3 for single
   writer to serialize concurrent updates on the partition-locked
   table, and we lock HLSDS.slppit3 for write on a single partition.
5) We do an all-AMPs RETRIEVE step from HLSDS.srct3 by way of an
   all-rows scan with a condition of ("HLSDS.srct3.b = 100") into
   Spool 1 (all_amps), which is built locally on the AMPs.
   Then we do a SORT to partition Spool 1 by rowkey.
   The size of Spool 1 is estimated
   with no confidence to be 1 row (33 bytes).
   The estimated time for this step is 0.03 seconds.
6) We do an all-AMPs MERGE into HLSDS.slppit3 from Spool 1
   (Last Use). The size is estimated with
   no confidence to be 1 row.
   The estimated time for this step is 1.73 seconds.
7) Finally, we send out an END TRANSACTION step to all AMPs
   involved in processing the request.

```

## Example: INSERT and GENERATED ALWAYS Identity Columns

Assume that *column\_1* of newly created *table\_1* is an identity column defined as GENERATED ALWAYS. The following INSERT statements automatically generate numbers for *column\_1* of the inserted rows, even if they specify a value for that column.

```

INSERT INTO table_1 (column_1, column_2, column_3)
VALUES (111,111,111);
INSERT INTO table_1 (column_1, column_2, column_2)
VALUES (,222,222);

```

Check the inserts by selecting all the rows from *table\_1*.

```
SELECT *
FROM table_1;
*** Query completed. 2 rows found. 3 columns returned.
*** Total elapsed time was 1 second.
  column_1  column_2  column_3
  -----  -
           2         111         111
           1         222         222
```

Note the following things about this result:

- Even though the value 111 was specified for *column\_1* in the first insert, the value is rejected and replaced by the generated value 2 because *column\_1* is defined as GENERATED ALWAYS.
- A warning is returned to the requestor when a user-specified value is overridden by a system-generated number.
- The first number in the identity column sequence is not necessarily allocated to the first row inserted because of parallelism.

## Example: INSERT and GENERATED BY DEFAULT Identity Columns

The result of inserting values into the same table defined in [Example: INSERT and GENERATED ALWAYS Identity Columns](#) shows the difference in the behavior of the two identity column types. Because GENERATED BY DEFAULT only generates values when an INSERT request does not supply them, the same insert operations produce different results.

Assume that *column\_1* of newly created *table\_1* is an identity column defined as GENERATED BY DEFAULT. The following INSERT requests generate numbers for *column\_1* of the inserted rows only if they do not specify a value for that column.

```
INSERT INTO table_1 (column_1, column_2, column_3)
VALUES (111,111,111);

INSERT INTO table_1 (column_1, column_2, column_2)
VALUES (,222,222);
```

Check the result of the insert operations by selecting all the rows from *table\_1*.

```
SELECT *
FROM table_1;
*** Query completed. 2 rows found. 3 columns returned.
*** Total elapsed time was 1 second.
  column_1  column_2  column_3
  -----  -
           2         111         111
           1         222         222
```

111	111	111
1	222	222

## Example: Identity Columns and INSERT ... SELECT

Assume that *column\_1* of *table\_1* is a GENERATED BY DEFAULT identity column.

```
INSERT INTO table_2 (column_1, column_2, column_3)
VALUES (111,111,111);
INSERT INTO table_2 (column_1, column_2, column_3)
VALUES (222,222,222);
INSERT INTO table_2 (column_1, column_2, column_3)
VALUES (1,333,333);

INSERT INTO table_1 (column_1, column_2, column_3)
SELECT * FROM table_2;
```

Check the result of the insert operations by selecting all the rows from *table\_1*.

```
SELECT *
FROM table_1;
*** Query completed. 3rows found. 3 columns returned.
*** Total elapsed time was 1 second.
column_1  column_2  column_3
-----
      111      111      111
      222      222      222
       1      333      333
```

## Example: INSERT and Queue Tables

The following request populates a queue table with values from a base table. The value for the QITS column of the queue table is the default timestamp.

```
INSERT INTO orders_qt (productid, quantity, unitprice)
SELECT productid, quantity, unitprice
FROM backorders
WHERE ordernumber = 1002003;
```

The following request performs a FIFO pop on the *orders\_qt* queue table and stores it into the *backorders* base table:



```
INSERT INTO backorders (productid, quantity, unitprice)
SELECT AND CONSUME TOP 1 productid, quantity, unitprice
FROM orders_qt;
```

### Example: Non-Valid Use of PARTITION In VALUES Clause of INSERT Request

This example is not valid because PARTITION cannot be referenced in the VALUES clause of an INSERT request.

```
INSERT INTO Orders VALUES (PARTITION, 10, 'A', 599.99,
    DATE '2001-02-07', 'HIGH', 'Jack', 3, 'Invalid insert');
```

### Example: Simple INSERT Requests Using a DEFAULT Function

The following example set uses this table definition:

```
CREATE TABLE table7 (
    col1 INTEGER,
    col2 INTEGER DEFAULT 10,
    col3 INTEGER DEFAULT 20,
    col4 CHARACTER(60) );
```

The following INSERT request is valid:

```
INSERT INTO table7
VALUES (1, 2, DEFAULT, 'aaa');
```

The DEFAULT function evaluates to the default value of *col3*, the third column position in the insert list. The example INSERT request inserts the value 20 into *table7* for *col3*.

The resulting row is as follows:

col1	col2	col3	col4
1	2	20	aaa

The following INSERT request is valid:

```
INSERT INTO table7 (col1, col3, col4)
VALUES (1, DEFAULT, 'bbb');
```

The DEFAULT function evaluates to the default value of *col3* because DEFAULT is specified second in the expression list, and the second column in the column list is *col3*.

The resulting row is as follows:

	col1	col2	col3	col4
	-----	-----	-----	-----
1	10	20	bbb	

You can specify a DEFAULT function with a column name in an expression list. This is a Teradata extension to the ANSI SQL:2011 standard.

The following INSERT request is valid:

```
INSERT INTO table7
VALUES (1, 2, DEFAULT(col2), 'aaa');
```

The DEFAULT function evaluates to the default value of *col2* because *col2* is passed as an input argument to the DEFAULT function. The INSERT results in the value 10 for *col3* in the row because *col3* is the third column position in the insert list.

The resulting row is as follows:

	col1	col2	col3	col4
	-----	-----	-----	-----
	1	2	10	aaa

The following INSERT request is valid:

```
INSERT INTO table7 (col1, col3, col4)
VALUES (1, DEFAULT(col2), 'bbb');
```

Because *col2* is passed as an input argument to the DEFAULT function, the function evaluates to the default value of *col2*. Because the second column in the column list is *col3*, the system assigns it the value 10, which is the default value of *col2*.

The resulting row is as follows:

	col1	col2	col3	col4
	-----	-----	-----	-----
	1	10	10	bbb

You can specify the DEFAULT function with a column name anywhere in the expression. This is a Teradata extension to the ANSI SQL:2011 standard.

The following INSERT request is valid:

```
INSERT INTO table7
VALUES (1, 2, DEFAULT (col2)+5, 'aaa');
```

The DEFAULT function evaluates to the default value of *col2* plus 5 (or 10+5). Because *col3* is in the third column position in the insert list, the resulting row is as follows:

col1	col2	col3	col4
1	2	15	aaa

The following INSERT request is valid:

```
INSERT INTO table7 (col1, col3, col4)
VALUES (1, DEFAULT(col2)+5, 'bbb');
```

The DEFAULT function evaluates to the default value of *col2* plus 5 (or 10+5). Because the second column in the column list is *col3*, the system assigns the value 15 to it. The resulting row is as follows:

col1	col2	col3	col4
1	10	15	bbb

When there is no explicit default value associated with a column, the DEFAULT function evaluates to null.

Assume the following table definition for this example:

```
CREATE TABLE table9 (
  col1 INTEGER,
  col2 INTEGER NOT NULL,
  col3 INTEGER NOT NULL DEFAULT NULL
  col4 INTEGER CHECK (Col4>100) DEFAULT 99 );
```

The following INSERT request is valid:

```
INSERT INTO table9
VALUES (DEFAULT, 99, 101, 101);
```

In this example, *col1* is nullable and does not have an explicit default value associated with it. Therefore, the DEFAULT function evaluates to null.

The resulting row is as follows:

col1	col2	col3	col4
?	99	101	101

Assume the following table definition for the examples below:

```
CREATE TABLE table10
  col1 INTEGER ,
  col2 INTEGER DEFAULT 55,
  col3 INTEGER NOT NULL DEFAULT 99 );
```

The following examples are correct and use Teradata extensions to the DEFAULT function:

```

INSERT INTO table10
VALUES (1, DEFAULT(col2), DEFAULT(col3));

INSERT INTO table10 (col1, col2)
VALUES (1, DEFAULT(col2));

INSERT INTO table10 (col1, col2, col3)
VALUES (1, DEFAULT(col2), DEFAULT(col3));

```

The following examples are correct and use ANSI SQL:2011-compliant syntax for the DEFAULT Function:

```

INSERT INTO table10
VALUES (1, DEFAULT, DEFAULT);

INSERT INTO table10 (col1, col2)
VALUES (1, DEFAULT);

INSERT INTO table10 (col1, col2, col3)
VALUES (1, DEFAULT, DEFAULT);

```

The resulting row for all the above insert operations is the following:

col1	col2	col3
1	55	99

The following INSERT requests are all equivalent to one another. The first example uses an ANSI-SQL:2011-compliant syntax:

```

INSERT INTO table10
VALUES (5, DEFAULT, 99);

```

The following example uses Teradata extensions to the ANSI-SQL:2011 syntax:

```

INSERT INTO table10
VALUES (5, DEFAULT(col2), 99);

```

The resulting row for both of these insert operations is as follows:

col1	col2	col3
5	55	99

## Example: Using the DEFAULT Function With INSERT ... SELECT

Assume the following table definitions for this example:

```
CREATE TABLE tbl_source (
  col1 INTEGER,
  col2 INTEGER DEFAULT 10,
  col3 INTEGER DEFAULT 20,
  col4 CHARACTER(60));

CREATE TABLE tbl_destination (
  col1 INTEGER,
  col2 INTEGER DEFAULT 10,
  col3 INTEGER DEFAULT 20,
  col4 CHARACTER(60));
```

The following example shows correct use of the DEFAULT function within an INSERT ... SELECT request:

```
INSERT INTO tbl_destination (col1)
SELECT COALESCE(col3, DEFAULT(col3)) END
FROM tbl_source
WHERE col3 <> DEFAULT;
```

In this example, the DEFAULT function evaluates to a constant value, which is the default value of *col3* of *tbl\_source*.

## Example: Logging Errors With INSERT ... SELECT

The following examples show various types of error logging with INSERT ... SELECT requests:

The following request logs all data, referential integrity, and USI errors to the default limit of 10 errors.

```
INSERT INTO t
SELECT *
FROM s
LOGGING ERRORS;
```

The following request logs all data, referential integrity, and USI errors to the default limit of 10 errors.

```
INSERT INTO t
SELECT *
FROM s
LOGGING ALL ERRORS;
```

The following request logs data, referential integrity, and USI errors with no error limit. This does *not* mean that there is no limit on the number of errors the system can log; instead, it means that errors will continue to be logged until the system-determined limit of 16,000,000 have been logged. See “CREATE ERROR TABLE” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

```
INSERT INTO t
SELECT *
FROM s
LOGGING ERRORS WITH NO LIMIT;
```

The following request logs data row, referential integrity, and USI errors to a limit of 100 errors.

```
INSERT INTO t
SELECT *
FROM s
LOGGING ERRORS WITH LIMIT OF 100;
```

## Example: Using the ST\_Geometry Data Type to Represent Other Geospatial Data Types for INSERTs

You can use the ST\_Geometry data type to represent implicitly all of the geospatial data types defined by the ANSI SQL:2011 standard, as indicated by the following set of examples. See *Teradata Vantage™ Geospatial Data Types*, B035-1181. Note that in every case, the geospatial values are actually stored using the ST\_Geometry data type, not the type specified in the INSERT request.

The POINT type has a 0-dimensional geometry and represents a single location in two-dimensional coordinate space.

The following example inserts such a point into table *tab1*:

```
INSERT INTO tab1 VALUES (0, 'POINT(10 20)');
```

The LINESTRING type has a 1-dimensional geometry and is usually stored as a sequence of points with a linear interpolation between the individual points.

The following example inserts such a point sequence into table *tab1*:

```
INSERT INTO tab1 VALUES (0, 'LINESTRING(1 1, 2 2, 3 3, 4 4)');
```

The POLYGON type has a 2-dimensional geometry consisting of one exterior boundary and 0 or more interior boundaries, where each interior boundary defines a hole.

The following example inserts such a polygon into table *tab1*:

```
INSERT INTO tab1 VALUES (0, 'POLYGON((0 0, 0 20, 20 20, 20 0, 0 0),
                                     (5 5, 5 10, 10 10, 10 5, 5 5))
                           ');
```

The GEOMCOLLECTION type is a collection of 0 or more ST\_Geometry values.

The following example inserts such a geometric collection into table *tab1*:

```
INSERT INTO tab1 VALUES (0, 'GEOMETRYCOLLECTION(
    POINT(10 10),
    POINT(30 30),
    LINESTRING(15 15, 20 20 ) )');
```

The MULTIPOINT type is a 0-dimensional geometry collection whose elements are restricted to POINT values.

The following example inserts such a geometric collection into table *tab1*:

```
INSERT INTO tab1 VALUES (0, 'MULTIPOINT(1 1, 1 3, 6 3, 10 5, 20 1)');
```

The MULTILINESTRING type is a 1-dimensional geometry collection whose elements are restricted to LINESTRING values.

The following example inserts such a geometric collection into table *tab1*:

```
INSERT INTO tab1 VALUES (0, 'MULTILINESTRING((1 1, 1 3, 6 3),
    (10 5, 20 1))');
```

The MULTIPOLYGON type is a 2-dimensional geometry collection whose elements are restricted to POLYGON values.

The following example inserts such a geometric collection into table *tab1*:

```
INSERT INTO tab1 VALUES (0, 'MULTIPOLYGON(
    ((1 1, 1 3, 6 3, 6 0, 1 1)),
    ((10 5, 10 10, 20 10, 20 5, 10 5)))');
```

## Example: Using the ST\_Geometry Data Type When Inserting Geospatial Data Into Tables

This is a more detailed example of using the ST\_Geometry data type to represent other geospatial data types when inserting geospatial data into tables. Create two tables in the *test* database: *cities*, which represents a list of cities and *streets*, which represents a list of streets. The cities are Polygons and the streets are LineStrings, and both are implemented using the ST\_Geometry type as you can see from the table definitions for *cities* and *streets*. Insert some cities and streets into these tables using the well-known text format for these geospatial types and then submit a query to see if any of the streets are within any of the cities.

Here are the table definitions:

```
CREATE TABLE test.cities (
    pkey      INTEGER,
    CityName  VARCHAR(40),
    CityShape ST_Geometry);
```

```
CREATE TABLE test.streets (
  pkey          INTEGER,
  StreetName    VARCHAR(40),
  StreetShape  ST_Geometry);
```

First insert three rows into the *cities* table:

```
INSERT INTO test.cities VALUES(0, 'San Diego',
                                'POLYGON((1 1, 1 3, 6 3, 6 0, 1 1))'
                                );

INSERT INTO test.cities VALUES(1, 'Los Angeles',
                                'POLYGON((10 10, 10 20, 20 20,
                                           20 15, 10 10))'
                                );

INSERT INTO test.cities VALUES(2, 'Chicago',
                                'POLYGON((100 100, 100 200,
                                           300 200, 400 0, 100 100))'
                                );
```

Then insert three rows into the *streets* table:

```
INSERT INTO test.streets VALUES(1, 'Lake Shore Drive',
                                 'LINESTRING(110 180, 300 150)'
                                 );

INSERT INTO test.streets VALUES(1, 'Via Del Campo',
                                 'LINESTRING(2 2, 3 2, 4 1)'
                                 );

INSERT INTO test.streets VALUES(1, 'Sepulveda Blvd',
                                 'LINESTRING(12 12, 18 17)'
                                 );
```

Now join *cities* and *streets* on `StreetShape.ST_Within(CityShape)=1` and select the *StreetName* and *CityName* column values from them:

```
SELECT StreetName, CityName
FROM test.cities, test.streets
WHERE StreetShape.ST_Within(CityShape) = 1
ORDER BY CityName;
```

If you use BTEQ, the result looks like this:



StreetName	CityName
-----	-----
Lake Shore Drive	Chicago
Sepulveda Blvd	Los Angeles
Via Del Campo	San Diego

### Example: INSERT Using a Scalar Subquery

A scalar subquery can be specified as a parameterized value in the value list of a simple INSERT request, but Teradata Database always interprets it as a noncorrelated scalar subquery.

For example, Teradata Database processes the scalar subquery `SELECT d2 FROM t2 WHERE a2=t1.a1` in the select list of the following INSERT request as if it were `SELECT d2 FROM t2, t1 WHERE a2=t1.a1`.

In other words, Teradata Database interprets *t1* in the scalar subquery as a distinct instance of *t1* rather than as the target table *t1* of the insert operation.

```
INSERT INTO t1 VALUES (1,2,3 (SELECT d2
                               FROM t2
                               WHERE a2=t1.a1));
```

### Example: INSERT Using a PERIOD Value Constructor

The following examples use tables *t1* and *t2*, which are defined as follows:

```
CREATE TABLE t1 (
  c1 INTEGER
  c2 PERIOD(DATE))
UNIQUE PRIMARY INDEX (c1);
CREATE TABLE t2 (
  a INTEGER
  b DATE
  c DATE)
UNIQUE PRIMARY INDEX (a);
```

The following two INSERT requests both use a PERIOD value constructor:

```
INSERT INTO t1
VALUES (1, PERIOD(DATE '2005-02-03', DATE '2006-02-04'));
INSERT INTO t1
  SELECT a, PERIOD(b, c)
  FROM t2;
```

## Example: Passing an SQL UDF to a Single-Row Request

This example passes an SQL UDF into a single-row INSERT request. Note that the arguments must be passed as constants.

```
INSERT INTO t1
VALUES (1, test.value_expression(2, 3), 4);
```

## Example: Using the HASH BY Option With a NoPI Table

Assume that you have created the following NoPI tables for this example:

```
CREATE TABLE orders (
  o_orderkey    INTEGER NOT NULL,
  o_custkey     INTEGER,
  o_orderstatus CHAR(1) CASESPECIFIC,
  o_totalprice  DECIMAL(13,2) NOT NULL,
  o_ordertsz    TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_comment     VARCHAR(79))
UNIQUE INDEX (o_orderkey),
PARTITION BY COLUMN;
CREATE TABLE orders_staging AS orders
WITH NO DATA
NO PRIMARY INDEX;
```

The following INSERT ... SELECT request redistributes rows by the hash value of *o\_orderkey* to provide even distribution of the data selected from *orders\_staging* and ordered locally to obtain better run length compression on *o\_ordertsz* before locally copying into *orders*.

```
INSERT INTO orders
SELECT *
FROM orders_staging
HASH BY o_orderkey
LOCAL ORDER BY o_ordertsz;
```

For the HASH BY clause, *o\_orderkey* resolves to *o\_orderkey* in the *orders* table. Because this corresponds to the first expression in the select expression list of the SELECT request, which is *o\_orderkey* from *orders\_staging*, Teradata Database distributes the spool generated for the SELECT request on *o\_orderkey*. Similarly, Teradata Database uses *orders.staging.o\_ordertsz* to order the spool before locally inserting into *orders*.

This example uses the same tables.

```
INSERT INTO orders
  SELECT o_orderkey, o_custkey, o_orderstatus, o_totalprice + 10,
         o_orderts, o_comment
  FROM orders_staging
  HASH BY o_totalprice;
```

For the HASH BY clause, *o\_totalprice* resolves to *o\_totalprice* in the *orders* table. Because this corresponds to the fourth expression in the select expression list of the SELECT request, which is *o\_totalprice + 10*, Teradata Database distributes the spool generated for the SELECT request on the value of this expression, not on the value of *orders\_staging.o\_totalprice*.

To distribute on the values of *orders\_staging.o\_totalprice*, qualify the reference in the HASH BY clause as the following revision of the previous INSERT ... SELECT request does.

```
INSERT INTO orders
  SELECT o_orderkey, o_custkey, o_orderstatus, o_totalprice + 10,
         o_orderts, o_comment
  FROM orders_staging
  HASH BY orders_staging.o_totalprice;
```

## Example: Using the LOCAL ORDER BY Option

This example uses the same tables as the previous example except that it uses a new version of *orders\_staging* that has an integer column named *o\_sequence*, which it uses to order the rows it selects locally.

```
ALTER TABLE orders_staging
  ADD o_sequence INTEGER;
```

The following INSERT ... SELECT request locally orders the rows of the columns it selects from *orders\_staging* by *o\_sequence* before inserting them into *orders*.

```
INSERT INTO orders
  SELECT o_orderkey, o_custkey, o_orderstatus, o_totalprice+10,
         o_orderts, o_comment
  FROM orders_staging
  LOCAL ORDER BY o_sequence;
```

For the LOCAL ORDER BY clause, *o\_sequence* does not resolve to a column in the target table *orders*, so Teradata Database resolves using the standard rules for resolution to *o\_sequence* in the underlying table of the SELECT request, *orders\_staging*. Note that *orders\_staging.o\_sequence* is not included in the select expression list and the spool is generated locally and sorted on the value of *orders\_staging.o\_sequence* before being locally inserting into *orders*.

## Example: Using the RANDOM Option to Randomly Redistribute Data Blocks of Rows and Individual Rows Before Copying Them Locally

Assume that you created the following NoPI tables for this example.

```
CREATE TABLE tnopi1 (
  a INT,
  b INT,
  c INT)
NO PRIMARY INDEX;

CREATE TABLE tnopi2 (
  a INT,
  b INT,
  c INT)
NO PRIMARY INDEX;
```

The following INSERT ... SELECT request uses the RANDOM option to redistribute data blocks of rows randomly from *tnopi1* before locally copying them into *tnopi2*.

```
INSERT INTO tnopi2
  SELECT *
  FROM tnopi1
  HASH BY RANDOM;
```

The following INSERT ... SELECT request uses the RANDOM function to redistribute individual rows randomly from *tnopi1* before locally copying them into *tnopi2*:

```
INSERT INTO tnopi2
  SELECT *
  FROM tnopi1
  HASH BY RANDOM(1, 2000000000);
```

## Example: Inserting into a Table with an Implicit Isolated Load Operation

For information on defining a load isolated table, see the WITH ISOLATED LOADING option for CREATE TABLE and ALTER TABLE in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

Following are the table definitions for the example.

```
CREATE TABLE ldi_table1,
  WITH CONCURRENT ISOLATED LOADING FOR ALL
```

```

(a INTEGER,
 b INTEGER,
 c INTEGER)
PRIMARY INDEX ( a );

```

```

CREATE TABLE t1
(c1 INTEGER,
 c2 INTEGER,
 c3 INTEGER)
PRIMARY INDEX ( c1 );

```

This statement performs an insert into the load isolated table ldi\_table1 as an implicit concurrent load isolated operation:

```

INSERT WITH ISOLATED LOADING INTO ldi_table1
SELECT * FROM t1 WHERE c1 > 10;

```

## Example: Inserting into a Table with an Explicit Isolated Load Operation

For information on defining a load isolated table and performing an explicit isolated load operation, see the WITH ISOLATED LOADING option for CREATE TABLE and ALTER TABLE, in addition to Load Isolation Statements in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

Following are the table definitions for the example.

```

CREATE TABLE ldi_table1,
  WITH CONCURRENT ISOLATED LOADING FOR ALL
(a INTEGER,
 b INTEGER,
 c INTEGER)
PRIMARY INDEX ( a );

```

```

CREATE TABLE t1
(c1 INTEGER,
 c2 INTEGER,
 c3 INTEGER)
PRIMARY INDEX ( c1 );

```

This statement starts an explicit concurrent load isolated operation on table ldi\_table1:

```

BEGIN ISOLATED LOADING ON ldi_table1
  USING QUERY_BAND 'LDILoadGroup=Load1;';

```

This statement sets the session as an isolated load session:

```
SET QUERY_BAND='LDILoadGroup=Load1;' FOR SESSION;
```

This statement performs an explicit concurrent load isolated insert into table ldi\_table1:

```
INSERT INTO ldi_table1 SELECT * FROM t1 WHERE c1 > 10;
```

This statement ends the explicit concurrent load isolated operation:

```
END ISOLATED LOADING FOR QUERY_BAND 'LDILoadGroup=Load1;';
```

You can use this statement to clear the query band for the next load operation in the same session:

```
SET QUERY_BAND = 'LDILoadGroup=NONE;' FOR SESSION;
```

## LOCKING Request Modifier

### Purpose

Locks a database, table, view, or row at the specified severity, overriding the default locking severity that the system places for a request. With the appropriate privileges, you can place this lock to upgrade or downgrade the default lock.

In addition to DML statements, you can also specify the LOCKING modifier with DDL statements including CREATE VIEW, REPLACE VIEW, CREATE RECURSIVE VIEW, REPLACE RECURSIVE VIEW, CREATE MACRO, and REPLACE MACRO. You can use the LOCKING modifier to override the default locking severities and prevent deadlocks.

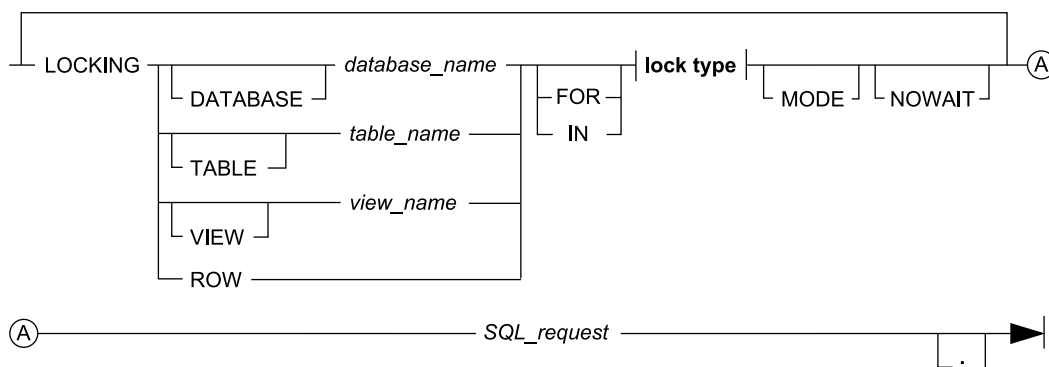
For more information, see:

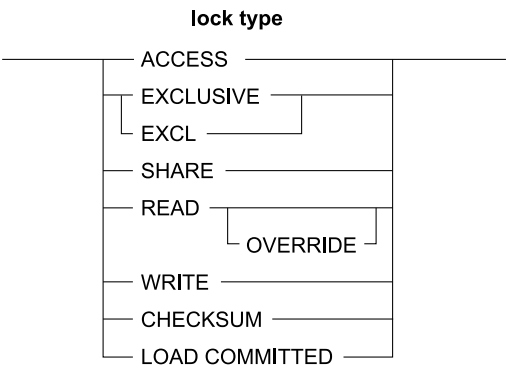
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142
- *Teradata® Archive/Recovery Utility Reference*, B035-2412
- *Teradata Vantage™ - Database Utilities*, B035-1102

### Required Privileges

None.

### Syntax





## Syntax Elements

### Locking Object

#### DATABASE

Optional keyword to indicate the type of object to lock is a database.

#### *database\_name*

Name of the database or user to be locked.

#### TABLE

Optional keyword to indicate the type of object to lock is a table.

#### *table\_name*

Name of the base table to be locked.

#### VIEW

Optional keyword to indicate the type of object to lock is a view.

#### *view\_name*

Name of the view to be locked.

#### ROW

Optional keyword specifying rowhash or rowkey level for locking in accordance with the defining statement. See [Using LOCKING ROW](#).

### Lock Type

You can specify the following lock severities. For additional information about locking, see “Locking and Transaction Processing” in *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

#### FOR

#### IN

Introduction to the type of lock to be placed.

#### ACCESS

Permits selection of data from a base table that can be locked for write access by other users.

The data selected using an ACCESS lock can be inconsistent because the data may be modified concurrently with the request. Therefore, you should only use this lock for casual inspection of data.

Placing an ACCESS lock requires the SELECT privilege on the specified object.

## **EXCLUSIVE**

Excludes all other users.

This is the most restrictive lock.

EXCLUSIVE locks are rarely used except to make structural changes to a database.

Placing an EXCLUSIVE lock on a database object requires the DROP privilege on that object.

## **READ SHARE**

Ensures data consistency during a read operation such as a SELECT request.

Multiple users can concurrently hold a READ lock on a base table. As long as a READ lock is in place, no modification of the object is allowed.

Placing a READ lock requires the SELECT privilege on the specified object.

SHARE is a deprecated synonym for READ.

## **WRITE**

Enables a single user to modify data.

As long as the WRITE lock is in place, all other users are excluded from viewing or modifying the object except readers who are viewing data using an ACCESS lock.

Until a WRITE lock is released, no new READ locks are permitted on the locked object.

Placing a WRITE lock requires an UPDATE, INSERT, or DELETE privilege on the specified object.

## **CHECKSUM**

Used only for updatable cursor queries submitted outside of stored procedures.

## **OVERRIDE**

Permit a READ lock for a single-table SELECT operation when a rollback on an underlying base table in the query was canceled using the Recovery Manager utility CANCEL ROLLBACK ON TABLE command. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

You can only specify the OVERRIDE option with a READ lock request.

## **LOAD COMMITTED**

Load-committed read mode. This option is independent of the session transaction isolation level.

Permits selection of committed data from a base load isolated table that can be locked for write access by other transactions.

Placing an LOAD COMMITTED lock requires the SELECT privilege on the specified object.

Internally, an ACCESS lock is applied when load-committed read mode is specified.



**MODE**

Optional keyword following the lock type.

**Do Not Wait Option****NOWAIT**

If the indicated lock cannot be obtained, the request should be aborted.

Specify this option for situations in which it is not desirable to have a request wait for resources, and possibly tie up resources another request could use, while waiting.

**SQL Request*****SQL\_request***

A valid SQL request.

You cannot specify LOCKING with the CALL statement.

If *SQL\_request* is null, then the only effect of LOCK is to lock the specified object.

This specification is mandatory for row-level locking (rowhash or rowkey), but optional for database-level locking, base table-level locking, and view-level locking.

**ANSI Compliance**

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

Other SQL dialects support similar non-ANSI standard statements with names such as LOCK TABLE.

**Usage Notes****LOCKING Request Modifier and CALL Requests**

You cannot specify a LOCKING request modifier with a CALL request.

**LOCKING Request Modifier Use With DML Statements**

The following table lists the associations between individual SQL DML statements and lock upgrades and downgrades at the row (rowkey or rowhash), view, base table, and database object levels:

LOCKING Modifier Severity	SQL DML Statements
EXCLUSIVE or WRITE	DELETE INSERT MERGE SELECT SELECT AND CONSUME

LOCKING Modifier Severity	SQL DML Statements
	UPDATE
READ or SHARE	SELECT
ACCESS	SELECT
ACCESS for LOAD COMMITTED	SELECT

EXCLUSIVE and WRITE are the only lock severities for the DELETE, INSERT, MERGE, UPDATE, and SELECT AND CONSUME statements because the default lock severity for these statements is WRITE. You cannot downgrade a WRITE lock for these statements because doing so would compromise the integrity of the database. Because the SELECT statement does not change data, and therefore cannot compromise database integrity, you are permitted to change its default locking severity to any other severity.

You can specify lower locking severity levels than are listed in this table, but if you do, the system ignores them and uses WRITE.

For details about locking levels, locking severities, and the relationship between them, see *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

## Canceling a Lock

An operation can wait for a requested lock indefinitely unless you specify the NOWAIT option. If you are working interactively and do not want to wait for a lock, you can issue the BTEQ .ABORT command to cancel the transaction.

## Positioning Explicit Lock Requests

When you need to place an explicit lock, the LOCKING modifier must precede the SQL request that is to be affected by the requested lock.

The system places the requested lock on the object referenced by the LOCKING modifier for the duration of the transaction containing the modified SQL request. If the transaction is a single-statement request, then the specified lock is only in effect for the duration of the request.

## Using Locks with NULL SQL Requests

If the LOCKING modifier is followed by a null SQL request, the only effect of the modifier is to lock the specified object. See [Null](#). While the LOCKING modifier can be used with a null request, it is best to use LOCKING with a user-generated transaction or as part of a multistatement request.

## Using LOCKING ROW

LOCKING ROW does not generally lock a single row, but rather all rows that hash to a specific value. It is a rowhash lock or rowkey lock, not a row lock.

The LOCKING ROW modifier cannot be used to lock multiple rowhashes. If you specify LOCKING ROW FOR ACCESS with multiple rowhashes, the lock is converted to LOCKING TABLE FOR ACCESS.

The use of LOCKING ROW on a row-partitioned table may place a rowhash or rowkey lock. The rowhash lock places a lock on a single rowhash in all partitions. The rowkey lock places a lock on a single rowhash in a single partition.

The use of LOCKING ROW prevents possible deadlocks occurring when two simultaneous primary index SELECT statements are followed by an UPDATE, DELETE, or INSERT on the same row.

For example:

User A:

```
BEGIN TRANSACTION;

SELECT y
FROM t
WHERE x=1;

UPDATE t
SET y=0
WHERE x=1;

END TRANSACTION;
```

User B:

```
BEGIN TRANSACTION;

SELECT z
FROM t
WHERE x=1;

UPDATE t
SET z=0
WHERE x=1;

END TRANSACTION;
```

The Lock Manager assigns a rowhash-level READ lock when a simple SELECT request is made using a UPI, NUPI, or USI.

The User A UPDATE request ROW FOR WRITE lock request waits for the User B ROW OR READ lock to be released. The ROW FOR READ lock is not released because the User B UPDATE request ROW FOR WRITE lock request is also waiting for the User A ROW FOR READ lock to be released. Previously, this deadlock was avoided by using the LOCKING TABLE modifier:

```
BEGIN TRANSACTION;

LOCKING TABLE t FOR WRITE
SELECT z
FROM t
WHERE x = 1;

UPDATE ...

END TRANSACTION;
```

Locking an entire base table across all AMPS is undesirable, and the use of LOCKING ROW here prevents the need to lock an entire base table across all AMPs.

The following example illustrates the use of LOCKING ROW:

User A:

```
BEGIN TRANSACTION;

LOCKING ROW FOR WRITE
SELECT y
FROM t
WHERE x=1;

UPDATE t
SET y=0
WHERE x=1
END TRANSACTION;
```

User B:

```
BEGIN TRANSACTION;

LOCKING ROW FOR WRITE
SELECT z
FROM t
WHERE x=1;

UPDATE t
```

```
SET z=0
WHERE x=1;
```

A deadlock does not occur because the User B LOCKING ROW FOR WRITE request is blocked by the User A LOCKING ROW FOR WRITE. The User B LOCKING ROW FOR WRITE request completes when the User A END TRANSACTION statement is complete.

The system honors a LOCKING ROW modifier request only in the following situations:

- For single-table retrievals, using primary or unique secondary index searches.
- When specified with a SELECT request, because the LOCKING ROW modifier picks up the rowhash to be locked from the SELECT request.
- To upgrade any of the following locks:

Lock to Upgrade	Upgraded Lock
<ul style="list-style-type: none"> <li>• READ</li> <li>• SHARE</li> </ul>	WRITE
<ul style="list-style-type: none"> <li>• READ</li> <li>• SHARE</li> </ul>	EXCLUSIVE
WRITE	EXCLUSIVE

The system does *not* honor a LOCKING ROW modifier request in the following situations:

- Where a lock on the target base table is already in place.
- Where an aggregate/DISTINCT/GROUP BY operation is part of the SELECT request.
- To downgrade a lock from READ to ACCESS, once that lock is in place.

If no rowhash lock is in place already, an ACCESS lock can be set instead of a default READ rowhash lock.

Deadlock occurs when two primary index SELECTs are followed by primary index UPDATES and the SELECTs are on different rowhashes. This is because a primary index update requires a table-level WRITE lock on the target base table.

For example:

User A:

```
BEGIN TRANSACTION;

LOCKING ROW WRITE
SELECT x
FROM t
WHERE x = 1; (x is UPI, NUPI, or USI)
UPDATE t
```

```
SET x=2
WHERE x=1;
```

User B:

```
BEGIN TRANSACTION;

LOCKING ROW WRITE
SELECT x
FROM t
WHERE x=2;

UPDATE t
SET x=1
WHERE x=2;
```

The User B SELECT ROW WRITE lock is not queued behind the User A transaction because it has a different rowhash access. Deadlock occurs because the User A table-level lock request waits on the User B rowhash lock, while the User B table-level lock request waits on the User A rowhash lock.

## Multiple Locks

Multiple LOCKING modifiers can precede a request if it is necessary to lock more than one object at the same time, for example:

```
LOCKING TABLE employee FOR ACCESS
LOCKING TABLE department FOR ACCESS
SELECT name, loc
FROM employee, department
WHERE (empno=mgrno);
```

## Referencing a Locked Object

The object that is locked by the LOCKING modifier does not have to be referenced in a subsequent SQL request.

A lock can be executed separately from the SQL request that it precedes. Therefore, a LOCKING modifier must reference the object on which a lock is being placed. The objects referenced in the SQL request have no effect on the execution of a LOCKING modifier.

When a LOCKING modifier references a view, the specified lock is applied to all underlying base tables. For example, if a view refers to tables t1 and t2, then a lock on that view would apply to both tables.

LOCKING DATABASE is the only way to lock a database or user.

## Specify the Keyword For the Object To Be Locked

Be sure to specify the keyword for the object (database, table, view, or rowhash) that is to be locked.

For example, if a database and table are both named *acctnrec*, then you could specify the following form:

```
LOCKING TABLE acctnrec FOR ACCESS
SELECT *
FROM acctnrec;
```

If the TABLE keyword had not been included, the lock would have been placed on the *acctnrec* database.

## Locks and Views

A LOCKING modifier can be specified in a view definition. When a view is defined with a LOCKING modifier, the specified lock is placed on the underlying base table set each time the view is referenced in an SQL request.

For more information on defining views, see CREATE VIEW and CREATE RECURSIVE VIEW in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## When the Request and View Referenced Include LOCKING Request Modifiers

Although views are often created to enforce a LOCKING FOR ACCESS rule, any user can override the LOCKING FOR ACCESS by specifying a LOCKING FOR READ request modifier on the view. For example:

```
REPLACE VIEW vprod.ad_me_inf AS
LOCKING TABLE prod.ad_me_inf FOR ACCESS
SELECT ad_me_id, ad_me_dsc_tx
FROM prod.ad_me_inf;
```

If you do an EXPLAIN on the following query, the ACCESS lock can be seen in statement 1.

```
SELECT COUNT(*)
FROM vprod.ad_me_inf;
```

If you do an EXPLAIN on the following query, you can see a READ lock in statement 1 of the report.

```
LOCKING TABLE vprod.ad_me_inf FOR READ
SELECT COUNT (*)
FROM vprod.ad_me_inf;
```

This behavior could be considered undesirable because the LOCKING FOR ACCESS request modifier can be overridden by anyone at any time. However, some users find this to be *useful* and depend on being able to override lock clauses in views by placing a lock in the request.

## READ Locks and Canceled Rollback Operations

When you use the RcvManager utility to cancel a transaction rollback, the system marks the base table on which the rollback was canceled as nonvalid. As a result, the table cannot be updated. With some restrictions, you can inspect the rows of the nonvalid base table if you specify a LOCKING FOR READ OVERRIDE modifier.

The following rules document the restrictions on the use of a LOCKING FOR READ OVERRIDE modifier:

- You can only read from a single base table using this modifier. Attempts to perform multitable operations return an error.
- You can specify LOCKING FOR READ OVERRIDE for any single base table, whether a rollback has been canceled on that table or not.
- The Optimizer only uses indexes to read base valid base tables. If you use LOCKING FOR READ OVERRIDE to access a nonvalid table, then the system always uses a full-table scan.

## Determining Which Locks Are Set

Use the EXPLAIN request modifier with an SQL request to determine what locks are set when the request is executed.

## Examples

### Example: LOCKING Request Modifier

The following LOCKING clause can be used to select data from the *employee* table while it is being modified:

```
LOCKING TABLE personnel.employee FOR ACCESS
SELECT name, salary
FROM employee
WHERE salary < 25000 ;
```

The query results may:

- Return rows whose data can be updated or deleted an instant later by a concurrent operation initiated by another user who has obtained a WRITE lock.
- Omit rows that are undergoing a concurrent insert operation.



- Include rows that were not permanently inserted in the base table, because a transaction inserting the new rows was aborted and the new rows were backed out.

## Example: LOCKING Request Modifier and Secondary Indexes

The system synchronizes base data rows and index subtable rows. However, an ACCESS lock can allow inconsistent results even when secondary indexes are used in conditional expressions because index constraints are not always rechecked against the data row.

For example, a column named *qualify\_accnt* is defined as a secondary index for a base table named *acct\_rec*, as in the following request:

```
LOCKING TABLE acct_rec FOR ACCESS
SELECT acct_no, qualify_accnt
FROM acct_rec
WHERE qualify_accnt = 1587;
```

The request could return:

acct_no	qualify_accnt
-----	-----
1761	4214

In this case, the value 1587 was found in the secondary index subtable and the corresponding data row was selected and returned. However, the data for account 1761 had been changed by another user while the retrieval was in process. This is referred to as a *dirty read*. See *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

Anomalous results like these are possible even if the data is changed only momentarily by a transaction that is ultimately aborted. The ACCESS lock is most useful to those who simply want an overview of data and are not concerned with consistency.

## Example: LOCKING ROW

This example shows the proper use of a rowhash lock.

```
CREATE TABLE customer (
  cust_id INTEGER,
  phone   INTEGER,
  fax     INTEGER,
  telex   INTEGER)
PRIMARY INDEX (cust_id),
UNIQUE INDEX(fax),
INDEX(telex);
```

```
CREATE TABLE sales (
  custcode INTEGER,
  zip      INTEGER,
  salesvol INTEGER);
```

User A:

```
BEGIN TRANSACTION;

LOCKING ROW EXCLUSIVE
SELECT phone
FROM customer
WHERE cust_id=12345;

UPDATE customer
SET phone=3108292488
WHERE cust_id=12345;
```

The User A EXCLUSIVE rowhash lock prevents another user from accessing the same row.

In the following, the user A rowhash WRITE lock, in conjunction with LOCKING TABLE, prevents user B from accessing the same row:

User A:

```
BEGIN TRANSACTION;

LOCKING TABLE sales FOR READ,
LOCKING ROW FOR WRITE
SELECT telex
FROM customer
WHERE fax=0;

UPDATE customer
SET telex=0
WHERE fax=0;

SELECT zip
FROM sales
WHERE custcode=111;

SELECT salesvol
FROM sales
WHERE custcode=222;

...
```

```
END TRANSACTION;
```

User B:

```
BEGIN TRANSACTION;

LOCKING ROW FOR WRITE
SELECT *
FROM customer
WHERE cust_id=12345
INSERT INTO customer (12345, 3108284422, 3108684231, 5555);

END TRANSACTION;
```

The User B LOCKING ROW FOR WRITE modifier waits until the User A transaction ends before it can be completed.

## Example: NOWAIT Option

You have a request that you do not want to be placed in the lock queue if it cannot be serviced immediately. In this case, use the NOWAIT option. For example:

```
LOCKING employee FOR READ NOWAIT
SELECT employee_number, last_name
FROM employee
WHERE department_number=401;

*** Failure 7423 Object already locked and NOWAIT.
Transaction Aborted. Statement# 1, Info =0
```

Another request had *employee* locked, so you must resubmit the request.

```
LOCKING employee FOR READ NOWAIT
SELECT employee_number, last_name
FROM employee
WHERE department_number=401;

*** Query completed. 7 rows found. 2 columns returned.
*** Total elapsed time was 1 second.
employee_number  last_name
-----
1003  Trader
1004  Johnson
1013  Phillips
```

1002	Brown
1010	Rogers
1022	Machado
1001	Hoover

This time, no locks were being held on *employee*, so the request completed successfully.

## MERGE

### Purpose

Merges a source row set into a primary-indexed target table based on whether any target rows satisfy a specified matching condition with the source row. The target table cannot be column partitioned.

IF the source and target rows ...	THEN the merge operation ...
satisfy the matching condition	updates based on the WHEN MATCHED THEN UPDATE clause.
	deletes based on the WHEN MATCHED THEN DELETE clause.
do not satisfy the matching condition	inserts based on the WHEN NOT MATCHED THEN INSERT clause.

For details on the temporal form of MERGE, see *Teradata Vantage™ Temporal Table Support*, B035-1182

For more information, see:

- [INSERT/INSERT ... SELECT](#)
- [UPDATE](#)
- [UPDATE \(Upsert Form\)](#)
- “CREATE ERROR TABLE” and “HELP ERROR TABLE” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ Temporal Table Support*, B035-1182
- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Database Utilities*, B035-1102
- *Teradata® FastLoad Reference*, B035-2411
- *Teradata® MultiLoad Reference*, B035-2409
- *Teradata® Parallel Data Pump Reference*, B035-3021

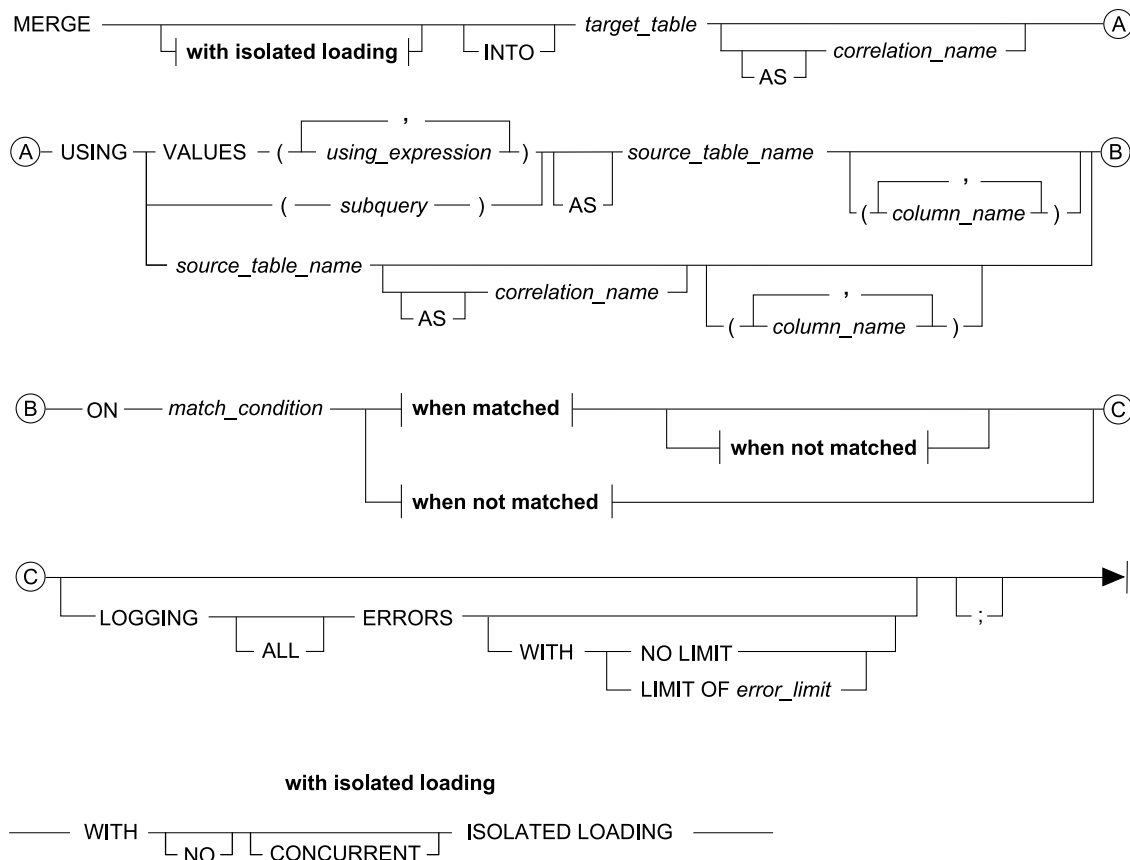
### Required Privileges

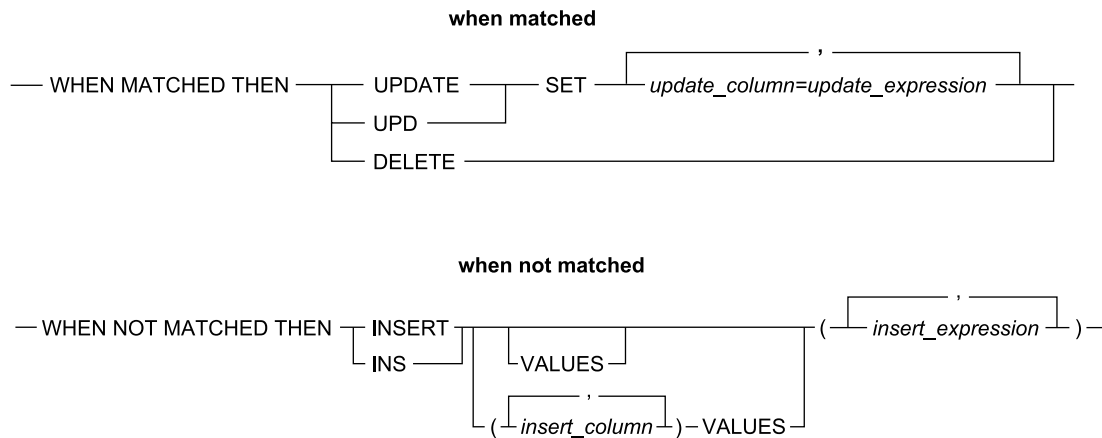
The privileges required to perform MERGE depend on the merge matching clause of the request you submit.

Merge Matching Clause	Privilege Required
WHEN MATCHED	UPDATE on every column of <i>target_table</i> that is specified in the UPDATE SET set clause list. DELETE on every column of <i>target_table</i> that is specified in the DELETE SET set clause list.
WHEN NOT MATCHED	INSERT on <i>target_table</i> . The INSERT privilege is also required on all of the columns specified in the INSERT column list.
both	<ul style="list-style-type: none"> <li>SELECT on any source table specified in a USING subquery.</li> <li>all the update and insert privileges required for the WHEN MATCHED and WHEN NOT MATCHED clauses.</li> </ul> Note: DELETE cannot be combined with INSERT or UPDATE in a MERGE statement.

The privileges required for a MERGE ... LOGGING ERRORS operation are the same as those for MERGE operations without a LOGGING ERRORS option with the exception that you must also have the INSERT privilege on the error table associated with the target data table for the MERGE operation.

## Syntax





## Syntax Elements

### Isolated Loading Options

#### WITH ISOLATED LOADING

The MERGE can be performed as a concurrent load isolated operation.

#### NO

The MERGE is not performed as a concurrent load isolated operation.

#### CONCURRENT

Optional keyword that you can include for readability.

### Target Table Options

#### *target\_table*

The base data table, global temporary table, volatile table, or queue table to:

- update rows in or delete rows from.
- insert rows into.

The target table must have a primary index and can be row partitioned but cannot be column partitioned.

### USING Clause

#### USING

An introduction to the table expression that defines the source table rows for the merge operation.

#### *using\_expression*

An expression defining the columns of the table expression that specifies the source table rows for the merge as a list of *using\_expression* values.

The table expression representing the source table rows is composed of the comma-separated list of *using\_expression* values. It is not a single value expression.

*using\_expression* can reference any of the following:

- Constants
- Built-in functions. For details, see “Built-In Functions” in *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.
- USING request modifier variables
- Macro parameters
- Stored procedure parameters
- Host variables
- String literals
- The target table for the MERGE operation.

For example, you can use the target table as the source table, but in that case, the target table becomes another instance table that contains the row data as it was prior to the update to the target table by the MERGE operation (see [Example: Using the Target Table as the Source Table](#)).

*using\_expression* cannot reference columns in any other base table.

### **subquery**

A subquery table expression that specifies the source table for the merge.

You can specify host variables in the WHERE clause, but *not* in the select list, of *subquery*.

All host variables must be preceded by a COLON character.

If *subquery* returns no rows, the source table is empty, no triggers are fired (see [MERGE as a Triggering Action](#)), and the merge operation performs neither update nor insert operations on *target\_table*.

*subquery* can reference any of the following:

- Rows in the queried table
- Constants
- USING request modifier variables

*subquery* cannot specify an ORDER BY clause.

The WHERE clause for *subquery* must include an equality condition on the UPI or USI of the queried table to ensure a singleton select.

### **source\_table\_name**

Name of the source table to be merged into *target\_table*.

### **column\_name**

Optional names to be used for the source row columns defined by the corresponding *using\_expression* or *subquery* select list expression or by *source\_table\_name*.

If you do not specify any column names, MERGE uses the column names specified in the *using\_expression* or *subquery* of the USING clause.

Source row columns, qualified by *source\_table\_name*, can be referenced in *match\_condition* or in an *update\_expression* or *insert\_expression*.

### **AS correlation\_name**

Optional correlation name for the source table specified by *using\_expression*, *subquery*, or *source\_table\_name*.

## **ON Clause**

### **ON match\_condition**

Conditional expression that determines whether the source row matches a given row in the target table. If the condition is met for any target rows and a WHEN MATCHED clause is specified, then the matching target rows are updated or deleted.

Match\_condition must specify an equality constraint on the primary index of *target\_table* to ensure that the candidate target row set can be hash-accessed on a single AMP. The specified primary index value must match the primary index value implied by the column values specified in the WHEN NOT MATCHED clause.

If the primary index value is the result of a *using\_expression*, the expression cannot reference any column in *target\_table*. Additionally, *match\_condition* cannot specify subqueries or references to columns that do not belong to either the source table or to *target\_table*.

If *target\_table* is a row-partitioned table, the values of the partitioning columns must also be specified in *match\_condition*, and the WHEN NOT MATCHED clause must specify the same partitioning column values as *match\_condition*.

Host variables are permitted in *match\_condition*.

All host variables must be preceded by a COLON character.

*match\_condition* cannot reference a table that is neither the target table nor the source table.

### **WHEN MATCHED THEN**

Introduction to the operation to be performed on matching rows.

You can specify WHEN MATCHED THEN and WHEN NOT MATCHED THEN clauses in any order.

### **UPDATE SET**

#### **UPD SET**

an update set clause operation to be performed for matching rows.

### **update\_column = update\_expression**

An equality condition on a target table column (specified by *update\_column*) that defines how the specified field is to be updated.

*update\_expression* produces a new value to be placed into the field to be updated.

*update\_expression* can include source table column references, target table column references, a constant, a null expressed by the reserved word NULL, a DEFAULT function, host variables, or an arithmetic expression for calculating the new value.



All host variables must be preceded by a COLON character.

You cannot specify a derived period column name.

## DELETE

Matching rows are deleted.

Note: DELETE cannot be combined with INSERT or UPDATE in a MERGE statement.

## WHEN NOT MATCHED THEN

An introduction to the operation to be performed on nonmatching rows.

You can specify WHEN MATCHED and WHEN NOT MATCHED clauses in any order. You can also specify a WHEN NOT MATCHED clause without also specifying a WHEN MATCHED clause.

## INSERT

### INS

An introduction to a value list to be inserted for nonmatching rows.

Because an inserted row might be a duplicate of an existing row in *target\_table*, its acceptance depends on whether *target\_table* is defined to be SET or MULTiset.

## VALUES

Optional keyword for the value list.

### *insert\_column*

Column name into which a corresponding value in the value list is to be inserted for nonmatching rows.

### *insert\_expression*

Value to be inserted into a corresponding *insert\_column* for nonmatching rows.

Host variables are permitted in *insert\_expression*.

All host variables must be preceded by a COLON character.

## Error Logging Options

If you do not specify a value for the LIMIT option, the system defaults to a 10 error limit.

## LOGGING ERRORS

### LOGGING ALL ERRORS

Log all data errors, reference index errors, and USI errors.

If you do not specify the LOGGING ERRORS option, the system does no error handling. If an error occurs, the following session-mode behavior occurs:

- If the current session mode is ANSI, then the erring request rolls back.
- If the current session mode is Teradata, then the erring transaction rolls back.

The optional keyword ALL is the default.

### WITH NO LIMIT

that there is no limit to the number of errors that can be logged in the error table associated with the target data table for this MERGE operation.

Note that this does *not* mean that there is no limit on the number of errors the system can log for a MERGE request; instead, it means that errors will continue to be logged until the system-determined limit of 16,000,000 errors have been logged. See “CREATE ERROR TABLE” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

#### **WITH LIMIT OF *error\_limit***

that the limit on the number of errors that can be logged in the error table associated with the target data table for this MERGE operation is *error\_limit*.

If this limit is exceeded, the system aborts the request in Teradata session mode or the transaction in ANSI session mode, and rolls back all changes made to the target table, but does not roll back the logged error table rows.

The value you specify for *error\_limit* can be anything in the range from 1 through 16,000,000, inclusive. The default value for *error\_limit* is 10.

### **ANSI Compliance**

MERGE is ANSI SQL:2011-compliant.

Note that in the ANSI definition, this statement is named MERGE INTO, while in the Teradata definition, INTO is an optional keyword.

## **Usage Notes**

### **Exceptions to Full ANSI Compliance**

The five exceptions to full ANSI compliance for the Teradata implementation of the MERGE statement are as follows:

- The ANSI definition for this statement is MERGE INTO, while the Teradata definition is MERGE, with INTO being an optional keyword.
- The Teradata implementation of MERGE does not support the ANSI OVERRIDE clause in the INSERT specification.

In the ANSI definition of the MERGE statement, this clause applies to identity columns only and allows the overriding of either user-specified or system-generated identity values. Teradata does not support this operation in its regular non-error logging MERGE statements, either.

- You cannot update or delete primary index column values using MERGE.
- The *match\_condition* you specify with the ON keyword must specify an equality constraint on the primary index of the target table. The target table cannot be a NoPI table or column-partitioned table.
  - Inequality conditions are not valid, nor are conditions specified on a column set other than the primary index column set for the target table.
  - The specified primary index value must match the primary index value implied by the column values specified in the INSERT clause.

- *match\_condition* cannot contain subqueries or references to columns that do not belong to either the source or target tables.
- *match\_condition* cannot equate explicitly with NULL.
- If the primary index value is the result of an expression, then the expression cannot reference any column in the target table.
- If the target table is a row-partitioned table, you must also specify the values of the partitioning columns in *match\_condition*, and the INSERT clause must specify the same partitioning column values as *match\_condition*.
- For multiply-sourced rows in a MERGE operation, the firing sequence of triggers defined on the target table depends on the order of the UPDATE and INSERT components of the MERGE request. This can impact the results of the MERGE operation.

If you specify the UPDATE component before the INSERT component, the order of processing is as follows:

1. BEFORE UPDATE triggers
2. BEFORE INSERT triggers
3. MERGE UPDATE and MERGE INSERT operations
4. AFTER UPDATE triggers
5. AFTER INSERT triggers

If you place the INSERT specification before the UPDATE specification, the order of processing is as follows:

1. BEFORE INSERT triggers
2. BEFORE UPDATE triggers
3. MERGE INSERT and MERGE UPDATE operations
4. AFTER INSERT triggers
5. AFTER UPDATE triggers

## Locks and Concurrency

The lock set for SELECT subquery operations depends on the isolation level for the session, the setting of the AccessLockForUncomRead DBS Control field, and whether the subquery is embedded within a SELECT operation or within a MERGE request.

Transaction Isolation Level	DBS Control AccessLockForUncomRead Field Setting	Default Locking Severity for Outer SELECT and Ordinary SELECT Subquery Operations	Default Locking Severity for SELECT Operations Embedded Within a MERGE Request
SERIALIZABLE	FALSE	READ	READ

Transaction Isolation Level	DBS Control AccessLockForUncomRead Field Setting	Default Locking Severity for Outer SELECT and Ordinary SELECT Subquery Operations	Default Locking Severity for SELECT Operations Embedded Within a MERGE Request
	TRUE		READ
READ UNCOMMITTED	FALSE		READ
	TRUE		ACCESS

MERGE requests are also affected by the locking levels set by you or the system. The default locking for MERGE requests is as follows.

- Table-level WRITE locks on the target table. For a nonconcurrent isolated merge on a load isolated table, the merge operation sets an EXCLUSIVE lock on the target table.
- READ or ACCESS locks on the source table depending on the situation and whether you specify a LOCKING request modifier.

The following cases illustrate the effect of these locking levels.

### Case 1

The query plan includes a table-level WRITE lock on target table *t1* in steps 1 and 2.

```

MERGE INTO t1
  USING (SELECT a2, b2, c2
        FROM t2
        WHERE a2 = 1) AS source (a2, b2, c2)
  ON a1 = a2
  WHEN MATCHED THEN
    UPDATE SET b1 = b2
  WHEN NOT MATCHED THEN
    INSERT (a2, b2, c2);

```

An EXPLAIN shows a write lock on *t1*.

### Case 2

Locking considerations become critical when you submit MERGE requests in an array-processing environment because those requests can cause transaction deadlocks if they are not coded correctly. For details, see *Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems*, B035-2417 or *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418. See also *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

This case is an example of how improper coding can cause deadlocks. In this case, the same request is repeated once each time through two different sessions as specified by the BTEQ command `.REPEAT 2` `PACK 100`. Because the sessions are running under ANSI transaction semantics, the lock on *t1* cannot be released until the COMMIT request is processed successfully.

Suppose there are two sessions, numbers 1025 and 1026. Assume further that session 1026 executes first. Session 1026 places a table-level WRITE lock on target table *t1* and completes the execution of the MERGE request. However, session 1026 cannot release the lock on target table *t1* until session 1025 completes because its transaction is not committed until both sessions have completed.

Session 1025 cannot complete its transaction because session 1026 has a table-level WRITE lock on *t1*. This is a classic case of deadlock where both sessions are waiting on one other for the lock to be released, causing both requests to hang.

```
.SET SESSION TRANSACTION ANSI
.SET SESSIONS 2
.LOGON nbmps05/ob,ob
.IMPORT INDICDATA file = ./recind.data
.REPEAT 2
USING (c3 INTEGER)
MERGE INTO t1
USING (SELECT a2, b2, c2
        FROM t2
        WHERE a2 = 1) AS source (a2, b2, c2)
ON a1 = a2 AND c1 =:c3
WHEN MATCHED THEN
    UPDATE SET b1 = b2
WHEN NOT MATCHED THEN
    INSERT (a2, b2, c2);
.REPEAT 20
COMMIT;
```

There are two ways to avoid the deadlock that results from this case:

- Redesign the application.
- Run the existing application in Teradata session mode to avoid the problem with the transaction-terminating COMMIT request.

## Related Topics

- “SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ - Database Utilities*, B035-1102

## About the MERGE Statement

The MERGE statement combines the UPDATE and INSERT statements into a single statement with two conditional test clauses:

- WHEN MATCHED, UPDATE.
- WHEN NOT MATCHED, INSERT.

You can also use the MERGE statement to delete rows by specifying: WHEN MATCHED, DELETE.

The following table explains the meaning of these conditional clauses:

Clause Evaluates to True	MERGE Description
WHEN MATCHED	Updates a matching target table row with the set of values taken from the current source row. Deletes a matching target table row.
WHEN NOT MATCHED	Inserts the current source row into the target table.

A MERGE statement can specify one WHEN MATCHED clause and one WHEN NOT MATCHED clause, in either order. You need not specify both. However, you must specify at least one of these clauses.

The Merge with Matched Updates and Unmatched Inserts AMP step performs inserts and updates in a single step. A merge with matched updates and unmatched inserts step can perform any of the following operations:

- INSERT only
- UPDATE only
- INSERT and UPDATE

A merge with matched deletes performs a DELETE only.

The merge with matched updates and unmatched inserts step assumes that the source table is always distributed on the join column of the source table, which is specified in the ON clause as an equality constraint with the primary index of the target table and sorted on the RowKey.

The step does a RowKey-based Merge Join internally, identifying source rows that qualify for updating target rows and source rows that qualify for inserts, after which it performs those updates and inserts.

This step is very similar to the APPLY phase of MultiLoad because it guarantees that the target table data block is read and written only once during the MERGE operation.

The order of evaluating whether a source row should be inserted into the target table or whether a matching target table row should be updated with the value set taken from the source varies accordingly. Note that the result depends on the order in which you specify the two clauses.

## MERGE Statement Processing

This section describes the general actions performed by a MERGE statement:

The Venn diagram below divides the source table rows into two disjunct sets: set *A* and set *B*. Set *A* is the set of matching rows and set *B* is the set of nonmatching rows.



This description uses the following terms:

- The *target\_table* variable specifies the target table being modified.
- The *correlation\_name* variable is an optional alias for the target table.
- The *source\_table\_name* variable following the USING keyword specifies the source table whose rows act as the source for update or delete and insert operations.
- The *match\_condition* variable following the ON keyword divides the source table into a set of rows that match rows in the target table and a set of rows that do not match rows in the target table.

The set of matching rows, *A*, defines the *update* or *delete* source, or staging, table.

The set of nonmatching rows, *B*, defines the *insert* source, or staging, table.

Either set can be empty, but both *cannot* be empty for a MERGE operation.

The system uses the set of matching rows as the update or delete source table for the update operation, as specified by the *update\_column=update\_expression* variable.

The system uses the set of nonmatching rows as the insert source table for the insert operation, as specified by the *insert\_expression* variable.

## MERGE Update and Insert Order of Application

The order of application of MERGE update and insert operations depends on the order in which you specify their respective WHEN MATCHED and WHEN NOT MATCHED clauses when you code your MERGE request. For details, see [Exceptions to Full ANSI Compliance](#).

## MERGE With Triggers

When a MERGE statement with a MERGE WHEN MATCHED clause executes, the system activates triggers defined on UPDATE or DELETE operations.

When a MERGE statement with a MERGE WHEN NOT MATCHED clause executes, the system activates triggers defined on INSERT operations.

The order of activation of UPDATE and INSERT triggers is the same as the order of the MERGE WHEN MATCHED and MERGE WHEN NOT MATCHED clauses in the MERGE request.

The following orders of operations apply to MERGE operations on source tables that have multiple rows. Assume for all cases that both updates to existing rows in the target table and inserts of new rows into the target table occur without incident.

## MERGE Order of Operations

The order of operations depends on whether you specify the MERGE WHEN MATCHED (update) clause first or the MERGE WHEN NOT MATCHED (insert) clause first as follows.

### MERGE WHEN MATCHED (Update) First

The sequence of actions when the MERGE statement executes are as follows:

1. All BEFORE triggers associated with UPDATE actions are applied.
2. All BEFORE triggers associated with INSERT actions are applied.
3. The UPDATE operations specified by the MERGE update specification and the INSERT operations specified by the MERGE insert specification are applied.
4. The system checks any specified constraints, which might result in referential actions being executed.
5. All AFTER triggers associated with UPDATE actions are applied.
6. All AFTER triggers associated with INSERT actions are applied.

### MERGE WHEN NOT MATCHED (Insert) First

The sequence of actions when the MERGE statement executes are as follows.

1. All BEFORE triggers associated with INSERT actions are applied.
2. All BEFORE triggers associated with UPDATE actions are applied.
3. The INSERT operations specified by the MERGE insert specification and the UPDATE operations specified by the MERGE update specification are applied.
4. The system checks any specified constraints, which might result in referential actions being executed.
5. All AFTER triggers associated with INSERT actions are applied.
6. All AFTER triggers associated with UPDATE actions are applied.

## MERGE With Duplicate Rows

MERGE processes duplicate UPDATE rows in the same way that the UPDATE statement does, and it processes duplicate INSERT rows in the same way that an INSERT ... SELECT request does, as indicated by the following bullets.

### MERGE INSERT Duplicate Rows

When MERGE detects duplicate rows during an INSERT operation, then Teradata Database takes different actions on the request depending on several factors.

- If no error logging is specified for the request AND the target table is a SET table AND the session is in ANSI session mode, then the request aborts and rolls back.
- If error logging is specified for the request AND the target table is a SET table AND the session is in ANSI session mode, then the request aborts and rolls back if there are any nonlocal errors, but only after the MERGE request completes or the specified error limit is reached.

Error-causing rows are logged in an error table and are *not* rolled back.



- If the target table is a SET table AND the session is in Teradata session mode, then any duplicate rows are silently ignored.

The INSERT source relation for a MERGE statement can contain duplicate rows.

Like the case for an INSERT ... SELECT statement, MERGE silently ignores duplicate row INSERT attempts into a SET table in Teradata session mode.

When the system inserts rows into the target table, the insertion of duplicate rows is governed by the normal constraint check rules enforced by the session mode types:

IN this session mode ...	The system handles duplicate row insert attempts by ...
ANSI	not inserting them into the target table. It logs them as errors in the appropriate error table. The system inserts <i>no</i> rows into the target table under these circumstances.
Teradata	inserting the first row of the duplicate set into the target table and rejecting all the remaining rows from that set without logging them as errors. The system inserts <i>one</i> row into the target table under these circumstances.

Unlike an INSERT ... SELECT statement, MERGE does *not* silently ignore duplicate row insert attempts into a SET table in Teradata session mode.

## MERGE UPDATE Duplicate Rows

When MERGE detects duplicate rows during an UPDATE operation, the actions Teradata Database performs depends on several factors. Duplicate row processing for UPDATE operations is the same in ANSI or Teradata session mode.

- If no error logging is specified for the request AND the target table is a SET table, then the request aborts and rolls back.
- If error logging is specified for the request AND the target table is a SET table, the request aborts and rolls back if there are any nonlocal errors, but only after the MERGE request completes or the specified error limit is reached.

Error-causing rows are logged in an error table and are *not* rolled back.

The UPDATE source relation for a MERGE statement can contain duplicate rows.

When the system updates rows in the target table, duplicate updates are processed as described in the following table:

IN this session mode ...	The system processes duplicate update attempts by ...
ANSI	taking one of the following actions: <ul style="list-style-type: none"> <li>• If error logging is enabled, Teradata Database logs each duplicate update attempt as an error in the appropriate error table.</li> </ul>

IN this session mode ...	The system processes duplicate update attempts by ...
	<p>This means that the system updates the target table row only once under these circumstances.</p> <ul style="list-style-type: none"> <li>If error logging is not enabled, Teradata Database aborts and rolls back the request.</li> </ul>
Teradata	<p>updating the row from the duplicate set the first time and rejecting all the remaining update attempts from that set.</p> <p>The system updates only <i>one</i> row in the target table under these circumstances.</p> <ul style="list-style-type: none"> <li>If error logging is enabled, Teradata Database logs each duplicate update attempt as an error in the appropriate error table.</li> </ul> <p>This means that the system updates the target table row only once under these circumstances.</p> <ul style="list-style-type: none"> <li>If error logging is not enabled, Teradata Database aborts and rolls back the request.</li> </ul>

## Rules and Limitations for MERGE

The following rule sets apply to different aspects of the MERGE statement.

### Single Row MERGE Rules and Limitations

The following items define the meaning of single row with respect to source table rules and limitations for MERGE requests:

- The specified *match\_condition* must specify a single primary index for the target row.
- If you specify a subquery for *source\_table\_reference*, then that subquery must be a singleton SELECT (that is, it cannot retrieve more than a single row from the referenced table) and it must specify either a UPI or USI to make the retrieval. The selected columns must be referenced explicitly: the SELECT \* syntax is not permitted.
- If you specify a simple list of column values for *source\_table\_reference*, then the retrieval, by definition, retrieves only a single row.

### Rules for MERGE Request ON Clauses

A MERGE request ON clause has:

- One, and only one, primary condition
- Zero, one, or many secondary conditions

A primary condition is the necessary condition for the MERGE request to execute, while secondary conditions are optional and are not required unless the application requires them.

Following are examples of primary and secondary ON clause conditions.

Suppose you create the following two tables and then submit the MERGE request that follows their definition.

```

CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER,
  c1 INTEGER);
CREATE TABLE t2 (
  a2 INTEGER,
  b2 INTEGER,
  c2 INTEGER);

MERGE INTO t1
USING t2
  ON a1=b2
WHEN MATCHED THEN
  UPDATE SET b1=b2;

```

The predicate `ON a1 = b2` is the primary, and only, condition in this MERGE request.

If the `ON` clause is modified to `ON a1=b2 AND c1=c2`, then `c1=c2` is a secondary condition.

### Row-Partitioned Tables Primary Condition Definition

For row-partitioned tables, the partitioning column set must be a component of the primary condition definition. For example, consider the MERGE request for the following target and source table definitions:

```

CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER,
  c1 INTEGER)
PRIMARY INDEX (a1)
PARTITION BY (c1);

CREATE TABLE t2 (
  a2 INTEGER,
  b2 INTEGER,
  c2 INTEGER);

MERGE INTO t1
USING t2
  ON a1=b2 AND c1=10 AND b1<b2
WHEN MATCHED THEN
  UPDATE SET b1=b2;

```

For this MERGE request, the primary condition is `a1=b2 AND c1=10`, and `b1<b2` is a secondary condition.

## Rules for ON Clause Definitions

The following set of rules clarifies these ON clause definitions.

### Rules for Primary Condition in the ON Clause

The primary condition in the ON clause must be an equality constraint. This is the minimum condition required for a valid ON clause predicate. Secondary conditions do not have this restriction.

For example, the following MERGE request is not valid because the primary ON clause condition,  $a1 < a2$ , is not an equality constraint.

```

MERGE INTO t1
  USING t2
    ON  a1<a2
  WHEN MATCHED THEN
    UPDATE SET b1=b2;

```

The following MERGE request is valid because the primary condition in its ON clause,  $a1 = a2$ , is an equality constraint. The secondary condition  $b1 <> b2$  being a nonequality has no bearing on the validity of the request because any  $\Theta$  operator is valid for a secondary condition.

```

      USING t2
MERGE INTO t1
    ON  a1=a2 AND b1<>b2
  WHEN MATCHED THEN
    UPDATE SET b1=b2;

```

### Restrictions for Primary Condition in the ON clause

The primary condition in the ON clause must not specify an expression on any of the following:

- Target table primary index
- Target table partitioning expression
- Both the target table primary index and its partitioning expression

Consider the following target and source table definitions:

```

CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER,
  c1 INTEGER)
PRIMARY INDEX (a1)
PARTITION BY (c1);

CREATE TABLE t2 (
  a2 INTEGER,

```

```
b2 INTEGER,
c2 INTEGER);
```

The following MERGE request is not valid because the primary condition in the ON clause specifies the expressions `a1+10` and `c1*b1` on the primary index `a1`, and the partitioning column `c1`, of target table `t1`, respectively.

```
MERGE INTO t1
USING t2
  ON a1+10=b2 AND c1*b1=10 AND b1<b2
WHEN MATCHED THEN
  UPDATE SET b1=b2;
```

However if the primary index, or the partitioning column set, or both are specified in a secondary condition, this restriction does not apply, as is demonstrated by the following valid example:

```
MERGE INTO t1
USING t2
  ON a1=b2 AND c1=10 AND a1+10=c2 AND c1*b1=10
WHEN MATCHED THEN
  UPDATE SET b1=b2;
```

In this MERGE request, the ON condition expressions `a1+10=c2` and `c1*b1=10` are specified in a secondary condition, so the request is valid.

### Primary Condition in ON Clause Must be Equality Condition

The primary condition in the ON clause must specify an equality condition with the primary index of the target table, and with its partitioning column if it is a row-partitioned table. The expression must also be identical to the expression specified for the primary index and partitioning column in the INSERT specification. The specified primary index value must equal the primary index value (and partitioning column value if the target table has row partitioning) implied by the column values specified in the INSERT specification of the WHEN NOT MATCHED clause.

### Primary Index Value Results from an Expression

If the primary index value results from the evaluation of an expression, then that expression cannot reference any column in the target table.

Consider the following target and source table definitions:

```
CREATE TABLE t1 (
  x1 INTEGER,
  y1 INTEGER,
  z1 INTEGER)
PRIMARY INDEX(x1,y1);
```

```
CREATE TABLE t2 (
  x2 INTEGER,
  y2 INTEGER,
  z2 INTEGER)
PRIMARY INDEX(x2,y2);
```

The following MERGE request is valid because it specifies equality conditions on each of the primary index columns of target table t1, the columns x1 and y1:

```
MERGE INTO t1
USING t2
  ON  x1=z2 AND y1=y2
WHEN MATCHED THEN
  UPDATE SET z1=10
WHEN NOT MATCHED THEN
  INSERT (z2, y2, x2);
```

The following MERGE request is valid because it specifies equality conditions on each of the primary index columns of target table t1 and because the expressions on the RHS of those conditions, z2+10 and y2+20, are also specified for the primary index columns of t1 in the INSERT specification of the request:

```
MERGE INTO t1
USING t2
  ON  x1=z2+10 AND y1=y2+20
WHEN MATCHED THEN
  UPDATE SET z1=10
WHEN NOT MATCHED THEN
  INSERT (x1, y1, z1) VALUES (z2+10, y2+20, x2);
```

### Target Table with Row Partitioning

For a target table with row partitioning, the MERGE request ON clause must specify a condition on the partitioning column and its WHEN NOT MATCHED clause must match that condition.

Consider the following target and source table definitions:

```
CREATE TABLE t1 (
  x1 INTEGER,
  y1 INTEGER,
  z1 INTEGER)
PRIMARY INDEX (x1)
PARTITION BY y1;

CREATE TABLE t2 (
  x2 INTEGER,
```

```

    y2 INTEGER,
    z2 INTEGER)
PRIMARY INDEX (x2);

```

The following MERGE request is valid because it specifies a condition on the partitioning column of t1, y1=y2, and its INSERT specification inserts a value from column t2.y2 into column t1.y1:

```

MERGE INTO t1
USING (SELECT *
      FROM t2) AS s
  ON x1=x2 AND y1=y2
WHEN MATCHED THEN
  UPDATE SET z1=z2
WHEN NOT MATCHED THEN
  INSERT (x2, y2, z2);

```

The following MERGE request is not valid because while its ON clause specifies a valid condition of x1=z2 AND y1=y2, its INSERT specification inserts a value from column t2.y2 into column t1.x1, which does not match the ON clause condition.

```

MERGE INTO t1
USING t2
  ON x1=z2 AND y1=y2
WHEN MATCHED THEN
  UPDATE SET z1=10
WHEN NOT MATCHED THEN
  INSERT (x1, y1, z1) VALUES (y2, z2, x2);

```

The following MERGE request is not valid because while its ON clause specifies a valid condition of x1=z2+10, its INSERT specification inserts the value for the expression t2.y2 + 20 into t1.x1, which does not match the ON clause condition.

```

MERGE INTO t1
USING t2
  ON x1=z2+10 AND y1=y2+20
WHEN MATCHED THEN
  UPDATE SET z1=10
WHEN NOT MATCHED THEN
  INSERT (y2+20, z2+10, x2);

```

### Primary Condition Must Be ANDed with Secondary Conditions

The primary condition must be conjunctive (ANDed) with any secondary conditions specified in the ON clause. The secondary condition terms can be disjunctive (ORed) among themselves, but not with the primary condition.

For example, consider the following target and source table definitions:

```
CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER,
  c1 INTEGER);

CREATE TABLE t2 (
  a2 INTEGER,
  b2 INTEGER,
  c2 INTEGER);
```

The following MERGE request is valid because the target table primary index equality expression  $a1=a2$ , which is the primary condition, is ANDed with the secondary condition  $b1=b2$ .

```
MERGE INTO t1
USING t2
  ON a1=a2 AND b1=b2
WHEN MATCHED THEN
  UPDATE SET c1=c2;
```

The following MERGE request is not valid because the target table primary index equality expression  $a1=a2$  is ORed with the secondary condition  $b1=b2$ .

```
MERGE INTO t1
USING t2
  ON a1=a2 OR b1=b2
WHEN MATCHED THEN
  UPDATE SET c1=c2;
```

However, the ON clause secondary conditions can contain ORed terms.

For example, the following MERGE request is valid because its primary condition  $a1=a2$  is ANDed with the secondary condition  $(b1=b2 \text{ OR } c1=c2)$ , and the disjunction is contained entirely within the secondary condition:

```
MERGE INTO t1
USING t2
  ON a1=a2 AND (b1=b2 OR c1=c2)
WHEN MATCHED THEN
  UPDATE SET c1=c2;
```

## MERGE ON Deterministic and Nondeterministic Functions

The rules for specifying deterministic and nondeterministic functions in a MERGE statement ON clause differ for primary and secondary conditions.



The section describes restrictions for specifying deterministic and nondeterministic functions in the primary condition of an ON clause.

### **MERGE ON Deterministic Functions in the Primary Condition**

There are no restrictions on the specification of deterministic functions in the primary condition of a MERGE statement ON clause.

For example, the following MERGE request is valid:

```
MERGE INTO t1
  USING t2
    ON a1=deterministic_udf(b2) AND b1=b2
  WHEN MATCHED THEN
    UPDATE SET c1=c2
  WHEN NOT MATCHED THEN
    INSERT (deterministic_udf(b2), a2, c2);
```

The deterministic UDF expression `deterministic_udf(b2)` executes only once for each row in the source table while spooling the source table `t2`. It is not executed again while the system processes the INSERT specification, but the value computed in the source spool is used in its place. This is a specific performance optimization and saves significant CPU time if the function is complicated.

### **MERGE ON Nondeterministic Functions in the Primary Condition**

You cannot specify nondeterministic functions in the primary condition of the ON clause because even when the ON clause expression matches the primary index value in the INSERT specification, the evaluation of the UDF might be different when it is executed in the context of the ON clause than when it is executed in the context of the INSERT specification, leading to an unreasonable insert.

In this context, an unreasonable insert is an insert operation that causes a nonlocal AMP insert. In other words, the row might need to be redistributed to a different AMP before it could be inserted into the target table.

For example, the following MERGE request is not valid because it specifies a nondeterministic function as the primary condition in the ON clause, even though that condition matches the INSERT specification:

```
MERGE INTO t1
  USING t2
    ON a1=non_deterministic_udf(b2)
  WHEN MATCHED THEN
    UPDATE SET b1=b2
  WHEN NOT MATCHED THEN
    INSERT (non_deterministic_udf(b2), a2, c2);
```

A parallel example that specifies the `RANDOM` function as the primary condition in the ON clause and matches the INSERT specification is also nondeterministic and, therefore, not valid. For more information

about the RANDOM function, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

```
MERGE INTO t1
USING t2
  ON a1=RANDOM (1,100)
WHEN MATCHED THEN
  UPDATE SET b1=b2
WHEN NOT MATCHED THEN
  INSERT (RANDOM (1,100), b2, c2);
```

To avoid this problem, you should always specify the appropriate DETERMINISTIC or NOT DETERMINISTIC option for the CREATE FUNCTION statement when you create your external UDFs. For details, see “CREATE FUNCTION” and “CREATE FUNCTION (Table Form)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184. This assists the Parser to take the actions necessary to process the MERGE request properly. If the external UDF is specified as DETERMINISTIC even though its behavior is NOT DETERMINISTIC, its execution can cause an internal AMP error during processing of the request, causing it to abort.

### **MERGE ON Deterministic and Nondeterministic Functions in Secondary Conditions**

The restrictions for specifying deterministic and nondeterministic functions in the secondary conditions of an ON clause are as follows:

There are no restrictions regarding either deterministic or nondeterministic functions that are specified in the secondary conditions of a MERGE request ON clause.

For example, the following case is a valid MERGE request because the deterministic\_udf and non\_deterministic\_udf functions are specified in the ON clause as secondary conditions, which is valid.

```
MERGE INTO t1
USING t2
  ON a1=a2 AND b1=deterministic_udf(b2)
      AND c1=non_deterministic_udf(c2)
WHEN MATCHED THEN
  UPDATE SET b1=b2
WHEN NOT MATCHED THEN
  INSERT (a2, deterministic_udf(b2),
        non_deterministic_udf(c2));
```

The RANDOM function is nondeterministic by definition. Therefore, the restrictions that apply to nondeterministic UDFs apply equally to RANDOM.

The following MERGE request is valid because it specifies a RANDOM function in a secondary condition of its ON clause.

```

MERGE INTO t1
USING t2
  ON a1=a2 AND  b1=RANDOM(1,100)
WHEN MATCHED THEN
  UPDATE SET b1=b2
WHEN NOT MATCHED THEN
  INSERT (a2, RANDOM(1,100), c2);

```

### Target Table Primary Index and Partitioning Column

The following rules apply to updating the primary index and partitioning column of a target table:

You cannot update the primary index or partitioning column of the target table.

Consider the following target and source table definitions:

```

CREATE TABLE t1 (
  x1 INTEGER,
  y1 INTEGER,
  z1 INTEGER);

CREATE TABLE t2 (
  x2 INTEGER,
  y2 INTEGER,
  z2 INTEGER)
PRIMARY INDEX(x2)
UNIQUE INDEX (y2);

```

The following MERGE request is valid because the source relation is a single row due to its WHERE clause specifying a constant for the USI column y2:

```

MERGE INTO t1
USING (SELECT x2, y2, z2
      FROM t2
      WHERE y2=10) AS s
  ON  x1=10 AND y1=20
WHEN MATCHED THEN
  UPDATE SET x1=10
WHEN NOT MATCHED THEN
  INSERT (y2, z2, x2);

```

The following MERGE request is not valid because even though the source relation s (derived from t2) is a single row, the ON clause does not specify a constant condition, which violates ON clause rule 2, and the primary index of target table t1, x1, is updated, which violates the nonupdatability rule on primary index and partitioning column expressions.

```

MERGE INTO t1
USING (SELECT x2, y2, z2
      FROM t2
      WHERE y2=10) AS s
ON x1=y2 AND y1=z2
WHEN MATCHED THEN
  UPDATE SET x1=10
WHEN NOT MATCHED THEN
  INSERT (y2, z2, x2);

```

### Primary Index of the Target Table

The primary index of the target table cannot be an identity column if you stipulate an INSERT specification and the ON clause predicate specifies an equality condition with the target table primary index (and with its partitioning column if it has row partitioning), and the expression specifies only source table columns.

Following are exceptions to this rule:

- The source relation is a valid single-row subquery.
- The request does not specify an INSERT specification and the primary condition in the ON clause is an equality constraint.
- You do not specify an INSERT clause if the MERGE request has an equality condition with the primary index of the target table (and partition column set, if the target table has row partitioning).

See [Example: MERGE and Identity Columns](#) for three valid examples.

Consider the following target and source table definitions. Note that target table t1 defines an identity column on its default primary index, which is column x1.

```

CREATE TABLE t1 (
  x1 INTEGER GENERATED BY DEFAULT AS IDENTITY,
  y1 INTEGER,
  z1 INTEGER);

CREATE TABLE t2 (x2 INT,
  y2 INTEGER,
  z2 INTEGER)
PRIMARY INDEX (x2)
UNIQUE INDEX (y2);

```

The following MERGE request is valid because its source relation s, based on a projection of t2, is a valid single-row subquery:

```

MERGE INTO t1
USING (SELECT x2, y2, z2
      FROM t2
      WHERE y2=10) AS s

```

```

    ON x1=10 AND y1=20
  WHEN MATCHED THEN
    UPDATE SET z1=10
  WHEN NOT MATCHED THEN
    INSERT (y2, z2, x2);

```

For the following MERGE request, if a constant is not specified, you would have to follow ON clause rule 1 for the case where a WHEN NOT MATCHED clause is not specified, which would render the request to be nonvalid. However, this example specifies the constant value 10 in the WHEN MATCHED clause, so it is valid.

```

MERGE INTO t1
  USING (SELECT x2, y2, z2
        FROM t2
        WHERE y2=10) AS s
  ON x1=y2 AND y1=z2
  WHEN MATCHED THEN
    UPDATE SET z1=10;

```

### **MERGE Columns Must Reference Source or Target Tables**

You can only specify columns that reference the source or target tables for the MERGE request in the ON, WHEN MATCHED, or WHEN NOT MATCHED clauses.

The following MERGE request is not valid because it references a column in its ON clause, t4.x4, that is neither from the source nor the target table for the request.

```

MERGE INTO t1
  USING (SELECT x2, y2, z3
        FROM t2, t3
        WHERE y2=10) AS s
  ON x1=y2 AND t4.x4=z2
  WHEN MATCHED THEN
    UPDATE SET z1=10
  WHEN NOT MATCHED THEN
    INSERT (y2, z2, x2);

```

The following MERGE request is not valid because it references a table and column in its ON clause, t3.x4, that are neither the source nor the target for the request.

```

MERGE INTO t1
  USING (SELECT x2, y2, z3
        FROM t2, t3
        WHERE y2=10) AS s
  ON x1=y2 AND t3.x4=z2
  WHEN MATCHED THEN

```

```
UPDATE SET z1=10
WHEN NOT MATCHED THEN
  INSERT (y2, z2, x2);
```

### ON Clause Conditions

ON clause conditions have the same restrictions as join conditions with the additional restriction that an ON clause cannot specify a subquery.

The following MERGE request is not valid because it specifies a subquery in its ON clause:

```
MERGE INTO t1
USING t2
  ON a1=a2 AND c1 IN (SELECT b2
                      FROM t2)
WHEN MATCHED THEN
  UPDATE SET c1=c2+2;
```

The following MERGE request is not valid because it specifies an aggregate operator, SUM, in its ON clause:

```
MERGE INTO t1
USING t2
  ON a1=a2 AND SUM(b1)=10
WHEN NOT MATCHED THEN
  INSERT (a2,b2,c2);
```

### Rules for Source Tables in a MERGE Statement

If the source relation is defined by a subquery, the subquery must conform to the restrictions for derived tables. See [Rules and Restrictions for Derived Tables](#). You cannot specifying any of the following SQL elements:

- ORDER BY
- recursion
- WITH
- WITH ... BY

The following MERGE request is not valid because it specifies a WITH ... BY clause in its source relation subquery:

```
MERGE INTO t1
USING (SELECT a2, b2, c2
      FROM t2
      WITH SUM(b2) BY c2) AS source (a2, b2, c2)
ON a1=a2
WHEN MATCHED THEN
  UPDATE SET c1=c2;
```

The following MERGE request is not valid because it specifies an ORDER BY clause in its source relation subquery:

```
MERGE INTO t1
  USING (SELECT a2, b2, c2
        FROM t2
        ORDER BY b2) AS source (a2, b2, c2)
  ON a1=a2
  WHEN NOT MATCHED THEN
    INSERT (a2,b2,c2);
```

If you specify the source table using a subquery, then the select list of that subquery cannot reference columns from the derived table it creates.

If the UPDATE source table contains more than one matching row for any given row in the target table, it is an error.

For example, suppose you have defined the following target and source tables:

```
CREATE TABLE target (
  a INTEGER,
  b INTEGER);

CREATE TABLE source (
  c INTEGER,
  d INTEGER);
```

You then populate the tables using the following INSERT requests:

```
INSERT INTO target VALUES (1,1);

INSERT INTO source VALUES (1,2);

INSERT INTO source VALUES (1,3);
```

Next you perform the following MERGE request:

```
MERGE INTO target AS t
  USING (SELECT c, d
        FROM source) AS s
  ON t.a=s.c
  WHEN MATCHED THEN
    UPDATE SET b=s.d;
```

Assume that the target table looks like this before it is updated. Note that the values for column d are ordered differently for version 1 and version 2 of the source table:

source - version 1		target		
c	d		a	b
1	2		1	1
1	3			

source - version 2	
c	d
1	3
1	2

The outcome of this MERGE update operation depends on the order of the rows in source. For example, compare the result of the update on target using version 1 of source versus the result of the update on target using version 2 of source.

Using version 1:

target	
a	b
1	2

Using version 2:

target	
a	b
1	3

As you can see, depending on the order of the rows in source, the final value for column b in target can be either 2 or 3, proof of nondeterministic behavior.

This is an error condition.

To avoid this error, you must collapse multiple matching rows in the source table into a single row beforehand. For example, you can aggregate multiple rows incrementing a column value in the *source\_table\_name* reference using a GROUP BY clause.

If the source is guaranteed to be a single row (that is, you specify an explicit value list or it is single table subquery with a UPI or USI constraint specified), then the ON clause predicate can have an equality condition between the primary index and a constant value, and the partitioning column set (if the target is a row-partitioned table) and a constant value.



The INSERT specification can either match or not match with the constant specified in the ON clause equality condition.

When the USING subquery guarantees a single-row source relation by specifying an equality condition on a unique index, you do not need to specify the partitioning column set in the subquery. See [Example: With a Guaranteed Single-Row Source Relation, You Do Not Need To Specify the Partitioning Column Set](#).

## Rules for Target Tables in a MERGE Statement

Following are the rules for target tables in a MERGE statement.

Only target table rows that existed before the MERGE statement began its execution are candidates for being updated. Rows inserted into the target table after the MERGE request begins its execution cannot be updated until after that MERGE request stops executing.

The target relation in a MERGE operation can be any of the following types of relation:

- Base data table
- Global temporary table
- Hashed table
- Queue table
- Updatable (single-table) view
- Volatile table

The target relation of a MERGE operation cannot be a joined table, or nonupdatable, view. For example, you cannot specify a view like the following as the target relation in a MERGE operation:

```
CREATE VIEW v (a, b, c) AS
  SELECT a2, b2, c2
  FROM t2 INNER JOIN t1 ON a1=a2;
```

The target relation of a MERGE operation cannot be any of the following types of relation:

- Derived table
- Global temporary trace table
- Hash index
- Joined table (nonupdatable) view
- Join index
- Journal table

When the target table is a row-partitioned table, you cannot substitute the system-derived PARTITION column or any of the system-derived PARTITION#L *n* columns, for the partitioning column set in the ON condition.

You can include the system-derived PARTITION column in the ON clause predicate as a secondary condition.

The target table in a MERGE operation can have an identity column. See “Identity Columns” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184. The system generates numbers for MERGE inserts into the identity column in the same way it performs singleton inserts into tables that do not have an identity column.

However, the identity column cannot be the primary index for the target table under most circumstances. For a list of rules and exceptions that apply to primary index identity columns, see [Primary Index of the Target Table](#).

You cannot specify target table columns in the INSERT specification of a WHEN NOT MATCHED THEN clause.

### Rules for MERGE WHEN MATCHED and WHEN NOT MATCHED

A MERGE request can specify at most one WHEN MATCHED clause and at most one WHEN NOT MATCHED clause.

When a MERGE request specifies a WHEN MATCHED and a WHEN NOT MATCHED clause, then the INSERT and UPDATE specifications of those clauses must apply to the same AMP.

The value specified for the primary index in the INSERT specification must match the primary index of the target table specified in the ON clause.

For a MERGE statement with a WHEN MATCHED clause, you must have the UPDATE privilege on every column that is being updated in the target table or the DELETE privilege on every column that is being deleted from the target table.

You must also have the SELECT privilege on the columns referenced in conditions and right-hand side of assignments for your MERGE request.

MERGE request UPDATE specifications have the same restrictions as an UPDATE request. See [UPDATE](#).

The *match\_condition* of a WHEN MATCHED clause must fully specify the primary index of the target table.

To use the WHEN NOT MATCHED clause, you must have the INSERT privilege on every column of the target table. You must also have the SELECT privilege on the columns referenced in conditions and right-hand side of assignments for your MERGE request.

MERGE request INSERT specifications have the same restrictions as an INSERT request with the exception that you cannot INSERT duplicate rows into a table even if it is defined as MULTiset and the request is made in an ANSI mode session. See [INSERT/INSERT ... SELECT](#)

You cannot specify columns referring to a table that is neither the source table nor the target table in a WHEN MATCHED or WHEN NOT MATCHED clause.

You cannot specify target table columns as values in the INSERT specification for a WHEN NOT MATCHED THEN clause because rows must be inserted from a source table, and an INSERT merged row cannot exist as a hybrid of source and target table rows. The following request is not valid because its INSERT specification includes target table column z1.

```

MERGE INTO t1
USING (SELECT x2, y2, z3
      FROM t2, t3
      WHERE y2=10) AS s
ON x1=y2 AND t4.x4=z2
WHEN MATCHED THEN
  UPDATE SET z1=10
WHEN NOT MATCHED THEN
  INSERT (x1, y1, z1) VALUES (y2, t1.z1, x2);

```

### Rules for MERGE Statements with DELETE

The MERGE-INTO statement with the DELETE clause deletes the target table rows for which the condition evaluates to true. The *source\_table\_name* following the USING keyword specifies the target table for deletes.

WHEN MATCHED THEN can include either UPDATE or DELETE, but not both.

If you specify DELETE, you cannot specify INSERT.

If the DELETE clause is specified and there is a DELETE trigger defined on the target object, the MERGE-INTO statement can act as the triggering event for the DELETE trigger.

### Rules for Using MERGE Requests With Embedded SQL Applications

You can embed MERGE requests in an embedded SQL application program, then submit them interactively or prepare and then execute them dynamically.

You can specify host variables for the WHERE clause of the USING *subquery* clause (but not in its select list), *match\_condition*, *update\_expression*, and *insert\_expression*.

All host variables must be preceded by a COLON character.

### Miscellaneous Rules for MERGE Requests

- You cannot specify INSERT DEFAULT VALUES in a MERGE request.
- MERGE supports UDTs.

See [MERGE Insert Operations](#), [MERGE Update Operations](#), and [UDTs](#).

- MERGE can be specified as a triggering action. See [MERGE as a Triggering Action](#) and “CREATE TRIGGER” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.
- MERGE cannot be specified as a triggered action. See [MERGE as a Triggered Action](#) and “CREATE TRIGGER” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

### Rules for Using the DEFAULT Function With MERGE Requests

The following rules apply when using a DEFAULT function within a MERGE request:

- The DEFAULT function takes a single argument that identifies a relation column by name. The function evaluates to a value equal to the current default value for the column. For cases where the default value of the column is specified as a current built-in system function, the DEFAULT function evaluates to the current value of system variables at the time the request is executed.

The resulting data type of the DEFAULT function is the data type of the constant or built-in function specified as the default unless the default is NULL. If the default is NULL, the resulting data type of the DEFAULT function is the same as the data type of the column or expression for which the default is being requested.

- You can specify the DEFAULT function as DEFAULT or DEFAULT (*column\_name*). When a column name is not specified, the system derives the column based on context. If the column context cannot be derived, the request aborts and an error is returned to the requestor.
- All the rules listed for the UPDATE statement also apply to the UPDATE in a MERGE statement. See [Rules for Using the DEFAULT Function With Update](#).
- All the rules listed for the INSERT statement also apply to the INSERT in a MERGE statement. See [Inserting When Using a DEFAULT Function](#).
- When the SQL Flagger is enabled, a DEFAULT function specified with a column name argument for inserts or updates is flagged as a Teradata extension.

For more information about the DEFAULT function, see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145.

## Rules for Using MERGE With Row-Partitioned Tables

The following rules apply to using the MERGE statement to insert rows into a row-partitioned primary index table or updating the columns of a partitioning expression.

- The target table can be row partitioned. However, the target table cannot be column partitioned.
- For MERGE requests that update the partitioning columns of a table, a partitioning expression must result in a value between 1 and the number of partitions defined for that level.
- For MERGE requests that insert a row into a table, the partitioning expression for that row must result in a value between 1 and the number of partitions defined for that level.
- If you specify a Period column as part of a partitioning expression, then you can only specify equality conditions on that Period column for a MERGE request. A MERGE request that specifies inequality conditions on a Period column included in a partitioning expression for the table returns an error.

You can specify a Period column that is not defined as part of a partitioning expression for both equality and inequality conditions on that column for a MERGE request.

- If you specify a function that references BEGIN or END Period values in a partitioning expression, an equality condition on that function is processed as a partitioning value matching condition.

If you specify a function that references a BEGIN and an END Period in a partitioning expression, the system processes the equality condition on the BEGIN and END as a partitioning matching condition.

Such a request must result in a single partition.

- The system processes the conditions IS UNTIL\_CHANGED and IS UNTIL\_CLOSED as equality conditions for the function that references the END Period only. See *Teradata Vantage™ ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ Temporal Table Support*, B035-1182.
- The INSERT clause must specify the same partitioning column values as the match condition.  
This rule also applies when the matching condition specifies a function that references a Period.
- You cannot update the system-derived columns PARTITION and PARTITION#L1 through PARTITION#L62.
- You cannot insert either a value or a null into any of the system-derived PARTITION columns.
- Errors, such as divide by zero, can occur during the evaluation of a partitioning expression. The system response to such an error varies depending on the session mode in effect at the time the error occurs.

Session Mode	Work Unit Rolled Back in Response to Expression Evaluation Errors
ANSI	Request that contains the error.
Teradata	Transaction that contains the error.

Take care in designing your partitioning expressions to avoid expression errors.

- For the merge operation to succeed, the session mode and collation at the time the table was created do not need to match the current session mode and collation. This is because the row partition in which a row is to be inserted or updated is determined by evaluating the partitioning expression on partitioning column values using the table's session mode and collation.
- Collation has the following implication for merging rows into a table defined with a character partitioning. If the collation for the table is either MULTINATIONAL or CHARSET\_COLL and the definition for the collation has changed since the table was created, the system aborts any request that attempts to merge a row into the table and returns an error to the requestor.
- If the partitioning expression for a table involves Unicode character expressions or literals, and the system has been backed down to a release that has Unicode code points that do not match the code points that were in effect when the table or join index was defined, Teradata Database aborts any attempts to insert rows into the table and returns an error to the requestor.

### Rules for Using MERGE on Tables with Row-Partitioned Join Indexes

Following are the rules for using MERGE on tables with row-partitioned join indexes:

- For MERGE requests that insert or update a row in a base table that causes an insert into a join index with row partitioning, the partitioning expression for that index row must result in a value between 1 and the number of partitions defined for that level.
- For MERGE requests that insert or update a row in a base table that causes an update of an index row in a join index with row partitioning, the partitioning expression for that index row after the update must result in a value between 1 and the number of partitions defined for that level.
- Merging a row into a base table does not always cause inserts or updates to a join index on that base table.

For example, you can specify a WHERE clause in the CREATE JOIN INDEX statement to create a sparse join index for which only those rows that meet the condition of the WHERE clause are inserted into the index, or, for the case of a row in the join index being updated in such a way that it no longer meets the conditions of the WHERE clause after the update, cause that row to be deleted from the index.

The process for this activity is as follows:

1. Teradata Database checks the WHERE clause condition for its truth value after the update to the row.

IF the condition evaluates to ...	THEN the system ...
FALSE	deletes the row from the sparse join index.
TRUE	retains the row in the sparse join index and proceeds to stage b.

2. Teradata Database evaluates the new result of the partitioning expression for the updated row.

Partitioning Expression Evaluation	Result
Null and no appropriate NO RANGE and UNKNOWN option has been specified.	Request returns error and does not update either the base table or the sparse join index.
Null and appropriate NO RANGE and UNKNOWN options have been specified.	Stores the row in either the NO RANGE or UNKNOWN partition for the sparse join index, and continues processing requests.
Value.	Stores the row in the appropriate partition, which might be different from the partition in which it was previously stored, and continues processing requests.

- Collation has the following implications for merging rows into a table defined with a character partitioning:

If a noncompressed join index with a character partitioning under either an MULTINATIONAL or CHARSET\_COLL collation sequence is defined on a table and the definition for the collation has changed since the join index was created, a request that attempts to merge a row into the table returns an error, whether the merge would have resulted in rows being modified in the join index or not.

If the partitioning expression for a noncompressed join index involves Unicode character expressions or literals, and the system has been backed down to a release that has Unicode code points that do not match the code points that were in effect when the table or join index was defined, an attempt to insert rows into the table returns an error.

### Rules for Invoking a Scalar UDF From a MERGE Request

You can invoke a scalar UDF from the following clauses of a MERGE request.

- **SELECT**

You can invoke a scalar UDF from the USING SELECT clause of a MERGE request.

You must use the alias to reference the result of a scalar UDF invocation that has an alias.

- **ON**

You can invoke a scalar UDF from the ON clause of a MERGE request as a secondary condition if it is invoked within an expression within the specified search condition or in the primary condition when it is bound to the primary index.

The other rules that are applicable to the ON clause of a SELECT request are applicable to the ON clause of a MERGE request. See [SELECT](#) and [Join Expressions](#).

- **UPDATE ... SET**

You can invoke a scalar UDF from the right-hand side of the SET clause in an UPDATE request as long as the UDF always return a value expression.

- **INSERT ... VALUES**

You can invoke a scalar UDF from the VALUES clause of the INSERT specification of a MERGE request.

The rules for specifying a scalar UDF in the VALUES clause of an INSERT specification are as follows:

- The arguments passed to a scalar UDF are restricted to the following categories:
  - Constants
  - Parameters that resolve to a constant
  - USING values
- The scalar UDF must always return a value expression.

## **MERGE Insert Operations, MERGE Update Operations, and UDTs**

For the rules about inserting into UDT columns, see [Inserting into UDT Columns](#).

## **Rules for Using Scalar Subqueries With MERGE Requests**

You cannot specify scalar subqueries in MERGE requests.

## **Rule for Using MERGE With Tables Protected by Row-Level Security Constraints**

You can use MERGE requests to:

- Update tables that have row-level security constraints (as long as the tables are defined with the same row-level security constraints).
- Insert rows into target tables.

## Rule for Queue Tables and MERGE

The following restrictions apply to specifying queue tables in a MERGE request:

- You cannot specify a MERGE statement in a multistatement request that contains a SELECT AND CONSUME request for the same queue table.
- You cannot specify a SELECT AND CONSUME request as the subquery in the USING clause of a MERGE request.

## MERGE as a Triggering Action

If *subquery* returns no rows, then no triggers are fired.

Triggers invoke the following behavior when fired as the result of a MERGE request:

Type of Action for which Trigger is Defined	Clause that Fires Trigger	<i>match_condition</i> Result
UPDATE	WHEN MATCHED THEN	Met.
DELETE	WHEN MATCHED THEN	Met.
INSERT	WHEN NOT MATCHED THEN	Not met.

## MERGE as a Triggered Action

MERGE is not supported as a triggered action.

## MERGE Support for Load Isolated Tables

A nonconcurrent load isolated merge delete operation on a load isolated table physically deletes the matched rows. A nonconcurrent isolated merge update operation on a load isolated table updates the matched rows in-place.

A concurrent load isolated merge delete operation on a load isolated table logically deletes the matched rows. A concurrent load isolated merge update operation on a load isolated table logically deletes the matched rows and inserts the rows with the modified values. The rows are marked as deleted and the space is not reclaimed until you issue an ALTER TABLE statement with the RELEASE DELETED ROWS option. See *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## Logging MERGE Errors In an Error Table

The assumption is that in the normal case, a MERGE request with error logging completes without any USI or RI errors. Exceptions to normal completion of a MERGE request are handled as follows:

- Not all types of errors are logged when you specify the LOGGING ERRORS option for a MERGE request.
  - All local, or data errors, are logged.



These are errors that occur during row merge step processing, such as CHECK constraint, duplicate row, and UPI violation errors.

- Errors that occur *before* the row merge step, such as data conversion errors detected in the RET AMP step before the MRG or MRM AMP steps, are not.
- When the system encounters USI or RI errors (or both) in the MERGE operation, the following events occur in sequence:
  1. The transaction or request runs to completion
  2. The system writes all erring rows into the error table
  3. The system aborts the transaction or request
  4. The system rolls back the transaction or request

Note that the system does not invalidate indexes, nor does it roll error table rows back, enabling you to determine which rows in the MERGE set are problematic and to determine how to correct them.

If the number of errors in the request is large, running it to completion plus rolling back all the INSERT and UPDATE operations can exert an impact on performance. To minimize the potential significance of this problem, you should always consider specifying a WITH LIMIT OF *error\_limit* clause.

The following rules and guidelines apply to logging errors in an error table for MERGE loads:

- Before you can log errors for MERGE loads, you must create an error table (see “CREATE ERROR TABLE” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144) for the base data table into which you intend to do a MERGE load.
- If error logging is not enabled and you submit a MERGE load operation with the LOGGING ERRORS option specified, the system aborts the request and returns an error message to the requestor.
- The LOGGING ERRORS option is valid in both ANSI and Teradata session modes.
- The LOGGING ERRORS option is *not* valid in a multistatement request.
- Two general categories of errors can be logged when you specify a LOGGING ERRORS option:
  - Local errors

Local errors are defined as errors that occur on the same AMP that inserts the data row. The following types of errors are classified as local errors:

Duplicate row errors, which occur only in ANSI session mode.

The system silently ignores duplicate row errors that occur from a MERGE into a SET table in Teradata session mode.

Duplicate rows can also arise from the following MERGE insert situations:

- The source table has duplicate rows.
- An insert is not well-behaved, meaning that the insert is made on a different AMP than the failed update.

Duplicate primary key errors

CHECK constraint violations

Attempts to update the same target table row with multiple source table rows

- Nonlocal errors

Nonlocal errors are defined as errors that occur on an AMP that does *not* own the data row. The following types of errors are classified as nonlocal errors:

- Referential integrity violations
- USI violations

An exception to this is the case where a USI violation is local because the USI is on the same set of columns as the primary index. The system treats such an error as a nonlocal error, even though it is local in the strict definition of a local error.

The system response depends on the type of error:

IF this type of error occurs ...	THEN the system records it in the error table, rejects the error-causing rows from the target table, and ...
local	completes the request or transaction successfully.
nonlocal	lets the request or transaction run to completion in order to record all the error causing rows in the MERGE load, then aborts the request or transaction and rolls back its inserts and updates.
both local and nonlocal	

- Teradata Database rejects data rows that cause local errors from the target table. However, the system does *not* reject data rows that cause nonlocal errors from the target table, but instead inserts them into the target table.
- Teradata Database does not handle batch referential integrity violations for MERGE error logging. Because batch referential integrity checks are all-or-nothing operations, a batch referential integrity violation causes the system to respond in the following session mode-specific ways:

IF this session mode is in effect ...	THEN the erring ...
ANSI	request aborts and rolls back.
Teradata	transaction aborts and rolls back.

- Teradata Database does not handle error conditions that do not allow useful recovery information to be logged in the error table. Such errors typically occur during intermediate processing of input data before it are built into a row format that corresponds to the target table.

Teradata Database detects this type of error before the start of data row inserts and updates. The following are examples of these types of error:

- UDT, UDF, and table function errors
- Version change errors
- Nonexistent table errors
- Down AMP request against nonfallback table errors
- Data conversion errors

Note that the system handles conversion errors that occur during data row inserts as local data errors.

The way Teradata Database handles these errors depends on the current session mode, as explained by the following table:

IF this session mode is in effect ...	THEN the erring ...
ANSI	request aborts and rolls back.
Teradata	transaction aborts and rolls back.

Teradata Database preserves error table rows logged by the aborted request or transaction and does *not* roll them back.

Teradata Database inserts a marker row into the error table at the end of a successfully completed MERGE request with logged errors.

Marker rows have a value of 0 in the ETC\_ErrorCode column of the error table, and their ETC\_ErrSeq column stores the total number of errors logged. All other columns in a marker row except for ETC\_DBQL\_QID and ETC\_TimeStamp are set to null.

If no marker row is recorded, the request or transaction was aborted and rolled back because of one or more of the following reasons:

- The specified error limit was reached.
- Teradata Database detected an error that it cannot handle.
- Teradata Database detected a nonlocal (RI or USI) violation.

The system preserves the error rows that belong to the aborted request or transaction.

- In addition to the previously listed errors, Teradata Database does not handle the following types of errors. However, it preserves the logged error rows for any of the errors listed.
  - Out of permanent space or out of spool space errors
  - Duplicate row errors in Teradata session mode, because the system ignores such errors in Teradata session mode
  - Trigger errors
  - Join index maintenance errors
  - Identity column errors
  - Implicit USI violations

When you create a table with a primary key that is not also the primary index, Teradata Database implicitly defines a USI on that primary key.

Teradata Database cannot invalidate a violated implicit USI on a primary key because it does not allow such a USI to be dropped and then recreated later.

Teradata Database handles the errors in the following session mode-specific ways:

IF this session mode is in effect ...	THEN the erring ...
ANSI	request aborts and rolls back.
Teradata	transaction aborts and rolls back.

- The LOGGING ERRORS option is applicable to MERGE load requests whose target tables are permanent data tables *only*.

Other kinds of target tables, such as volatile and global temporary tables, are not supported.

The system returns a warning message to the requestor if it logs an error.

- You cannot log errors for MERGE requests that specify unreasonable update or insert operations. In this case, the request or transaction containing the erring MERGE request behaves as follows when the system detects the unreasonable INSERT specification:

IF this session mode is in effect ...	THEN the erring ...
ANSI	request aborts and rolls back.
Teradata	transaction aborts and rolls back.

- You can log all errors or not log errors. You cannot specify the *types* of errors to log. The WITH LIMIT OF *error\_limit* option enables you to terminate error logging when the number of errors logged matches the number you specify in the optional WITH LIMIT OF *error\_limit* clause.

If you do not specify a LOGGING ERRORS option, and an error table is defined for the target data table of the MERGE request, the system does no error handling for MERGE operations against that data table.

In this case, the request or transaction containing the erring MERGE request behaves as follows when an error occurs:

IF this session mode is in effect ...	THEN the erring ...
ANSI	request aborts and rolls back.
Teradata	transaction aborts and rolls back.

- If you specify neither the WITH NO LIMIT option, nor the WITH LIMIT OF *error\_limit* option, the system defaults to an error limit of 10.

Teradata Database logs errors up to the limit of 10, and then the request of transaction containing the MERGE request behaves as follows when the eleventh error occurs:

IF this session mode is in effect ...	THEN the erring ...
ANSI	request aborts and rolls back.
Teradata	transaction aborts and rolls back.

Teradata Database preserves error table rows logged by the aborted request or transaction and does *not* roll them back.

- **WITH NO LIMIT**

Teradata Database places no limit on the number of error rows that can accumulate in the error table associated with the target data table for the MERGE operation.

- **WITH LIMIT OF *error\_limit***

Teradata Database logs errors up to the limit of *error\_limit*, and then the request or transaction containing the MERGE request behaves as follows when the *error\_limit* + 1 error occurs:

IF this session mode is in effect ...	THEN the erring ...
ANSI	request aborts and rolls back.
Teradata	transaction aborts and rolls back.

Teradata Database preserves error table rows logged by the aborted request or transaction and does *not* roll them back.

- The activity count returned for a MERGE ... LOGGING ERRORS request is the same as that returned for a MERGE operation without a LOGGING ERRORS option, a count of the total number of rows in the target table that were:
  - updated
  - inserted
  - deleted

**Note:**

DELETE cannot be used with INSERT or UPDATE. INSERT and UPDATE can be used individually or together.

The possible activity types returned to the application are listed in the following table:

Activity Type Name	Activity Type Number	Description
PCLMRGMIXEDSTMT	127	A mix of updates and inserts.
PCLMRGUPDSTMT	128	All updates, no inserts or deletes.
PCLMRGINSSTMT	129	All inserts, no updates or deletes.
PCLMRGDELSTMT	206	All deletes, no updates or inserts.

A dynamic MERGE request returns the following activity types:

This activity type ...	Is returned when this clause is specified for the MERGE request ...
PCLMRGMIXEDSTMT	WHEN MATCHED <i>and</i> WHEN NOT MATCHED.

This activity type ...	Is returned when this clause is specified for the MERGE request ...
	<ul style="list-style-type: none"> <li>• If the MERGE request only updates rows of the target table, Teradata Database may internally convert the activity type to PCLMRGUPDSTMT.</li> <li>• If the MERGE request only inserts rows into the target table, Teradata Database may internally convert the activity type to PCLMRGINSSTMT.</li> </ul>
PCLMRGUPDSTMT	WHEN MATCHED THEN UPDATE only.
PCLMRGINSSTMT	WHEN NOT MATCHED THEN INSERT only.
PCLMRGDELSTMT	WHEN MATCHED THEN DELETE only.

Following is an example of a MERGE statement and resulting output:

```

MERGE INTO t1 target
  USING (SEL a, b, c FROM src WHERE a <= 100) AS source (a, b, c)
  ON target.a1 = source.a
    WHEN MATCHED UPDATE SET b1=b1+100
    WHEN NOT MATCHED THEN INSERT (a,b,c);
*** Merge completed. 80 rows affected.
    30 rows inserted, 50 rows updated, no rows deleted.

```

See *Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems*, B035-2417 or *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418 for details about these activity types.

- LOGGING ERRORS does not support LOB data. LOBs in the source table are not copied to the error table. They are represented in the error table by nulls.
- An index violation error does not cause the associated index to be invalidated.
- For referential integrity validation errors, you can use the IndexId value with the RI\_Child\_TablesVX view (for information about the *RI\_Child\_TablesVX* view, see *Teradata Vantage™ Data Dictionary*, B035-1092) to identify the violated Reference index (for information about Reference indexes, see *Teradata Vantage™ - Database Design*, B035-1094). You can determine whether an index error is a USI or referential integrity error by the code stored in the ETC\_IdxErType error column.

IF the value of ETC_IdxErType is ...	THEN the error is a ...
R	foreign key insert violation.
r	parent key delete violation.
U	USI validation error.

## Examples

### Example: Using MERGE to Update and Insert

This example uses dynamically supplied values for an *employee* table row to update the table if the data matches an existing employee or insert the new row into the table if the data does not match an existing employee. Column *empno* is the unique primary index for the *employee* table.

This example can also be coded as the following upsert form of the UPDATE statement. See [UPDATE \(Upsert Form\)](#).

```

USING (empno  INTEGER,
       name   VARCHAR(50),
       salary INTEGER)
UPDATE employee
  SET salary=:salary
  WHERE empno=:empno
  ELSE INSERT INTO employee
    (empno, name, salary) VALUES ( :empno, :name, :salary);

```

### Example: MERGE With Subquery

This example generalizes [Example: Using MERGE to Update and Insert](#) by using a subquery for *source\_table\_reference* rather than an explicit value list.

```

USING (empno  INTEGER,
       salary INTEGER)
MERGE INTO employee AS t
USING (SELECT :empno, :salary, name
       FROM names
       WHERE empno=:empno) AS s(empno, salary, name)
  ON t.empno=s.empno
 WHEN MATCHED THEN UPDATE
   SET salary=s.salary, name = s.name
 WHEN NOT MATCHED THEN INSERT (empno, name, salary)
   VALUES (s.empno, s.name, s.salary);

```

### Example: Using the DEFAULT Function With MERGE

The following examples show the correct use of the DEFAULT function within the MERGE statement.

```

MERGE INTO emp
USING VALUES (100, 'cc', 200, 3333) AS emp1 (empnum, name,
                                             deptno, sal)

  ON emp1.empnum=emp.s_no
WHEN MATCHED THEN
  UPDATE SET sal=DEFAULT
WHEN NOT MATCHED THEN
  INSERT VALUES (emp1.empnum, emp1.name, emp1.deptno, emp1.sal);
MERGE INTO emp
USING VALUES (100, 'cc', 200, 3333) AS emp1 (empnum, name,
                                             deptno, sal)

  ON emp1.empnum=emp.s_no
WHEN MATCHED THEN
  UPDATE SET sal=DEFAULT(emp.sal)
WHEN NOT MATCHED THEN
  INSERT VALUES (emp1.empnum, emp1.name, emp1.deptno, emp1.sal);
  USING (empno INTEGER,
        name  VARCHAR(50),
        salary INTEGER)
MERGE INTO employee AS t
USING VALUES (:empno, :name, :salary) AS s(empno, name, salary)
  ON t.empno=s.empno
WHEN MATCHED THEN UPDATE
  SET salary=s.salary
WHEN NOT MATCHED THEN INSERT (empno, name, salary)
  VALUES (s.empno, s.name, s.salary);

```

## Example: Logging MERGE Errors

The following MERGE request logs all error types, including data errors, referential integrity errors, and USI errors, with an error limit of 100 errors.

```

MERGE INTO tgttbl AS t1
USING (SELECT c1,c2,c3
      FROM srctbl) AS t2
  ON t1.c1=t2.c1
WHEN MATCHED THEN
  UPDATE SET t1.c2=t2.c2 + t1.c2, t1.c3=t2.c3
WHEN NOT MATCHED THEN
  INSERT INTO t1 VALUES (t2.c1, t2.c2, t2.c3)
LOGGING ERRORS WITH LIMIT OF 100;

```



## Example: Using MERGE for Update and Insert Operations Within a Single SQL Request

The MERGE statement provides the ability to perform update and insert operations within a single SQL request. MERGE can also perform index and referential integrity maintenance in a single pass, unlike the case where update and insert operations must be executed separately.

For example, suppose you create the following tables and then use a MERGE request to update and insert rows from source table *t2* into target table *t1*:

```

USING (empno  INTEGER,      CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER,
  c1 INTEGER);

CREATE TABLE t2 (
  a2 INTEGER,
  b2 INTEGER,
  c2 INTEGER);

MERGE INTO t1
  USING t2
  ON a1=a2
  WHEN MATCHED THEN
    UPDATE
    SET b1=b2
  WHEN NOT MATCHED THEN
    INSERT (a2, b2, c2);

```

An EXPLAIN shows a merge with matched updates and unmatched inserts into OB.t1 from OB.t2 with a condition of ("OB.t1.a1 = OB.t2.a2").

This MERGE request can also be coded as the following semantically equivalent multistatement UPDATE INSERT request:

```

UPDATE t1
  FROM t2
  SET b1=b2
  WHERE a1=a2
;INSERT INTO t1
  SELECT a2, b2, c2
  FROM t2, t1
  WHERE NOT (a1=a2);

```

A comparison the two EXPLAIN reports shows that the MERGE request would outperform the semantically equivalent UPDATE and INSERT multistatement request.

In the UPDATE and INSERT multistatement request, the following steps are performed in parallel:

- MERGE Update to OB.t1 from OB.t2
- RETRIEVE step from OB.t2

Next, there is a JOIN step of the results to OB.t1 with a join condition of ("OB.t1.a1 <> a2").

## Example: Target Table Composite Primary Index

The following example shows the necessity of specifying an equality condition on the primary index (and also on the partitioning column set for a row-partitioned table) in the ON clause for a target table with a composite primary index:

Suppose you create the following two tables:

```
CREATE TABLE t1 (
  x1 INTEGER,
  y1 INTEGER,
  z1 INTEGER)
PRIMARY INDEX (x1, y1);

CREATE TABLE t2 (
  x2 INTEGER,
  y2 INTEGER,
  z2 INTEGER)
PRIMARY INDEX(x2, y2);
```

The following two MERGE requests are both valid:

```
MERGE INTO t1
USING t2
  ON x1=z2 AND y1=y2
WHEN MATCHED THEN
  UPDATE SET z1=10
WHEN NOT MATCHED THEN
  INSERT (z2,y2,x2);

MERGE INTO t1
USING t2
  ON x1=z2+10 AND y1=y2+20
WHEN MATCHED THEN
  UPDATE SET z1=10
```

```

WHEN NOT MATCHED THEN
  INSERT INTO (x1,y1,z1) VALUES (z2+10,y2+20,x2);

```

The following MERGE request is not valid because the ON clause specifies an equality condition where  $x1=z2$ , but the INSERT specification updates  $y2$  for  $x1$  rather than duplicating the ON clause specification, so it returns an error to the requestor:

```

MERGE INTO t1
USING t2
  ON x1=z2 AND y1=y2
WHEN MATCHED THEN
  UPDATE SET z1=10
WHEN NOT MATCHED THEN
  INSERT INTO (x1,y1,z1) VALUES (y2,z2,x2);

```

The following MERGE request is not valid because the ON clause specifies an equality condition where  $x1=z2+10$ , but the INSERT specification updates  $y2+20$  for  $x1$  rather than  $z2+10$ :

```

MERGE INTO t1
USING t2
  ON x1=z2+10 AND y1=y2+20
WHEN MATCHED THEN
  UPDATE SET z1=10
WHEN NOT MATCHED THEN
  INSERT (y2+20, z2+10, x2);

```

## Example: ON Clause Conditions Must Be ANDed With The Primary Index and Partitioning Column Equality Constraints

The following examples show the proper and improper specification of ON clause conditions in a MERGE request.

Consider the following table definitions, with  $t1$  being the target relation and  $t2$  being the source relation for all the examples that follow:

```

CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER,
  c1 INTEGER)
PRIMARY INDEX (a1);

CREATE TABLE t2 (
  a2 INTEGER,
  b2 INTEGER,

```

```
c2 INTEGER)
PRIMARY INDEX (a2);
```

The following example is correct because the primary index equality constraint  $a1=a2$  is ANDed with the other specified condition in the request,  $b1=b2$ :

```
MERGE INTO t1
USING t2
  ON a1=a2 AND b1=b2
WHEN MATCHED THEN
  UPDATE SET c1=c2
WHEN NOT MATCHED THEN
  INSERT (a2, b2, c2);
```

The following example is correct because the primary index equality constraint  $a1=a2$  is ANDed with the other specified condition in the request  $c1+c2=1$  OR  $b1+b2=1$ .

Even though the second condition is internally disjunctive, the result it evaluates to is ANDed with the primary index condition.

```
MERGE INTO t1
USING t2
  ON a1=a2 AND (c1+c2=1 OR b1+b2=1)
WHEN MATCHED THEN
  UPDATE SET c1=c2;
```

The following example is not valid. It aborts and returns an error message to the requestor because the primary index equality constraint  $a1=a2$  is ORed with the other specified condition in the request,  $c1=c2$ .

The primary index equality constraint, as well as the partitioning column equality constraint if the target table has a partitioned primary index, must always be ANDed to any other conditions you specify in the ON clause.

```
MERGE INTO t1
USING t2
  ON a1=a2 OR b1=b2
WHEN MATCHED THEN
  UPDATE SET c1=c2;
```

### Example: For a RPPI Table, the ON Clause Must Specify a Condition on the Partitioning Column and the INSERT Specification Must Match

When the target table of a MERGE operation has a row-partitioned primary index, the ON clause of the MERGE request must specify a condition on the partitioning column of the table and the order of the columns in the INSERT clause must be the same as the order you specify in the ON clause.

Consider the following table definitions:

```
CREATE TABLE t1 (
  x1 INTEGER,
  y1 INTEGER,
  z1 INTEGER)
PRIMARY INDEX (x1)
PARTITION BY y1;

CREATE TABLE t2 (
  x2 INTEGER,
  y2 INTEGER,
  z2 INTEGER)
PRIMARY INDEX (x2);
```

You want to use the following MERGE request to insert or update rows in *t1*, which has a primary index defined on *x1* and a partitioning expression defined on *y1*:

```
MERGE INTO t1
USING (SELECT *
      FROM t2) AS s
  ON x1=x2 AND y1=y2
WHEN MATCHED THEN
  UPDATE SET z1=z2
WHEN NOT MATCHED THEN
  INSERT (x2, y2, z2);
```

The request is successful because you have defined a condition on *y1* in the ON clause (*y1=y2*) and the specified order of columns in the INSERT clause matches the ordering specified in the ON clause.

### Example: Incorrect Examples Because of ON Clause Errors or Mismatches Between the ON Clause and the INSERT Specification

The following MERGE request fails because its ON clause specifies a condition of *x1=z2*, but its INSERT clause substitutes *y2* for *x1*:

```
MERGE INTO t1
USING t2
  ON x1=z2 AND y1=y2
WHEN MATCHED THEN
  UPDATE SET z1=10
WHEN NOT MATCHED THEN
  INSERT (x1, y1, z1) VALUES (y2, z2, x2);
```

The following MERGE request fails because its ON clause specifies a condition of  $x1=z2+10$ , but its INSERT clause inserts  $y2+20$  for  $x1$ :

```
MERGE INTO t1
USING t2
  ON x1=z2+10 AND y1=y2+20
WHEN MATCHED THEN
  UPDATE SET z1=10
WHEN NOT MATCHED THEN
  INSERT (y2+20, z2+10, x2);
```

### Example: MERGE With ON Clause and UPI or USI

Consider the following table definitions:

```
CREATE TABLE t1 (
  x1 INTEGER,
  y1 INTEGER,
  z1 INTEGER)
PRIMARY INDEX (x1)
PARTITION BY y1;

CREATE TABLE t2 (
  x2 INTEGER,
  y2 INTEGER,
  z2 INTEGER)
PRIMARY INDEX (x2)
UNIQUE INDEX (y2);
```

When the source relation is guaranteed to be a single row, either because you specify a value list or because it is created from a single table subquery with a UPI or USI constraint, and the ON clause specifies an equality condition on the primary index of the target table as a constant, the INSERT specification might or might not match the constant value specified in the ON clause. Either specification is valid, as the following two requests show.

The following MERGE request is valid even though the primary index value specified in the INSERT specification does not match the ON clause primary index specification:

```
MERGE INTO t1
USING (SELECT *
      FROM t2
      WHERE y2=10) AS s
  ON x1=10
WHEN MATCHED THEN
  UPDATE SET z1=z2
```

```

WHEN NOT MATCHED THEN
  INSERT (x2, y2, z2);

```

The following MERGE request is valid because the primary index value specified in the INSERT specification matches the ON clause primary index specification:

```

MERGE INTO t1
  USING (SELECT x2, y2, z2
        FROM t2
        WHERE y2=10) AS s
  ON x1=10 AND y1=z2
  WHEN MATCHED THEN
    UPDATE SET z1=10
  WHEN NOT MATCHED THEN
    INSERT (10, z2, x2);

```

When the source relation is guaranteed to be a single row, either because you specify a value list or because it is created from a single table subquery with a UPI or USI constraint, and the ON clause has an equality condition on the primary index of the target table that is *not* a constant, the INSERT clause primary index specification must match the primary index value for the target table specified in the ON clause. The UPDATE specification also must not update the primary index of the target table.

The following MERGE request is valid because the primary index value specified in the INSERT specification, *y2*, matches the primary index value specified in the ON clause, and the UPDATE specification does not update the primary index of *t1*:

```

MERGE INTO t1
  USING (SELECT x2, y2, z2
        FROM t2
        WHERE y2=10) AS s
  ON x1=y2 AND y1=z2
  WHEN MATCHED THEN
    UPDATE SET z1=10
  WHEN NOT MATCHED THEN
    INSERT (y2, z2, x2);

```

The following MERGE request fails because the source relation is not guaranteed to be a single row. The failure to guarantee a single row occurs because *z2* is neither a UPI nor a USI, so the INSERT specification must match the ON clause specification, which it does not.

```

MERGE INTO t1
  USING (SELECT x2, y2, z2
        FROM t2
        WHERE z2=10) AS s
  ON x1=10 AND y1=20
  WHEN MATCHED THEN

```

```
UPDATE SET z1=10
WHEN NOT MATCHED THEN
  INSERT (y2, z2, x2);
```

## Example: MERGE Updating a Primary Index

You cannot update primary index or partitioning column unless the source is a valid single-row subquery.

Suppose you have created the following tables:

```
CREATE TABLE t1 (
  x1 INTEGER,
  y1 INTEGER,
  z1 INTEGER)
PRIMARY INDEX(x1);

CREATE TABLE t2 (
  x2 INTEGER,
  y2 INTEGER,
  z2 INTEGER)
PRIMARY INDEX(x2)
UNIQUE INDEX(y2);
```

The following case is valid:

```
MERGE INTO t1
  USING (SELECT x2, y2, z2
        FROM t2
        WHERE y2=10) AS s
  ON x1=10 AND y1=20
WHEN MATCHED THEN
  UPDATE SET x1=10
WHEN NOT MATCHED THEN
  INSERT (y2, z2, x2);
```

The following case is *not* valid because while the subquery guarantees a single row, no constant is specified in the ON clause equality condition, so you cannot update the primary index as this request attempts to do:

```
MERGE INTO t1
  USING (SELECT x2,y2, z2
        FROM t2
        WHERE y2=10) AS s
  ON x1=y2 AND y1=z2
WHEN MATCHED THEN
```



```
UPDATE SET x1=10
WHEN NOT MATCHED THEN
INSERT (y2, z2, x2);
```

## Example: MERGE and Identity Columns

The primary index of the target table cannot be an identity column if you specify an INSERT clause even when the ON and INSERT clauses are valid.

However, if the source relation is created from a single row subquery, or if you do not specify an INSERT clause, then the target table primary index *can* also be an identity column.

Consider the following rules and examples based on these table definitions:

```
CREATE TABLE t1 (
  x1 INTEGER GENERATED ALWAYS AS IDENTITY,
  y1 INTEGER,
  z1 INTEGER)
PRIMARY INDEX(x1);

CREATE TABLE t2 (
  x2 INTEGER,
  y2 INTEGER,
  z2 INTEGER)
PRIMARY INDEX(x2)
UNIQUE INDEX(y2);
```

The rules, explicitly stated, are as follows:

- If you do not specify a WHEN NOT MATCHED THEN INSERT clause, the MERGE request is valid because there is no attempt to insert a value into *x1*, which is both the primary index of *t1* and an identity column.

For example, the following MERGE request is valid because no WHEN NOT MATCHED THEN INSERT clause is specified:

```
MERGE INTO t1
USING (SELECT x2, y2, z2
      FROM t2
      WHERE y2 = 1)
ON x1 = x2
WHEN MATCHED THEN
UPDATE SET y1 = y2;
```

- If you do specify a WHEN NOT MATCHED THEN INSERT clause, the MERGE request fails and returns an error to the requestor because an attempt is made to insert a value into *x1*, which is both the primary index of *t1* and an identity column.

For example, the following MERGE request fails because it specifies a WHEN NOT MATCHED THEN INSERT clause that attempts to insert the value of *x2* into *x1*, which is both the primary index of *t1* and an identity column:

```
MERGE INTO t1
  USING (SELECT x2, y2, z2
        FROM t2
        WHERE y2 = 1)
    ON x1 = x2
  WHEN MATCHED THEN
    UPDATE SET y1 = y2
  WHEN NOT MATCHED THEN
    INSERT (x2, y2, z2);
```

### Example: MERGE and Target Columns

The following example fails because the INSERT specification of the WHEN NOT MATCHED clause specifies a column, *z1*, from the target table *t1*, which is an illegal operation.

```
MERGE INTO t1
  USING (SELECT x2,y2, z3
        FROM t2, t3
        WHERE y2=10) AS s
    ON x1=y2 AND t4.x4=z2
  WHEN MATCHED THEN
    UPDATE SET z1=10
  WHEN NOT MATCHED THEN
    INSERT (x1,y1, z1) VALUES (y2, t1.z1, x2);
```

### Example: Reference the Source or Target in the ON, WHEN MATCHED, or WHEN NOT MATCHED Clauses

The following examples fail because they reference a table other than the source or target table in either their ON, WHEN MATCHED, or WHEN NOT MATCHED clauses.

The following example fails because table *t4* is neither the derived source table *s* nor the target table *t1*.

```
MERGE INTO t1
  USING (SELECT x2,y2, z3
        FROM t2, t3
        WHERE y2=10) AS s
    ON x1=y2 AND t4.x4=z2
  WHEN MATCHED THEN
```

```

UPDATE SET z1=10
WHEN NOT MATCHED THEN
  SELECT (y2, z2, x2);

```

The following example fails because table *t3* is neither the derived source table *s* nor the target table *t1*. Even though *t3* is specified in the USING source table subquery, it violates the restriction that only source and target tables can be referenced in an ON clause.

```

MERGE INTO t1
  USING (SELECT x2,y2, z3
        FROM t2, t3
        WHERE y2=10) AS s
  ON x1=y2 AND t3.x4=z2
WHEN MATCHED THEN
  UPDATE SET z1=10
WHEN NOT MATCHED THEN
  INSERT (y2, z2, x2);

```

### Example: Specifying the Partitioning Column Set in the ON Clause When the Target Relation Has a Row-Partitioned Primary Index

Consider the following table definitions where *t1* has a primary index on *a1* and is partitioned on column *b1*.

```

CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER,
  c1 INTEGER)
PRIMARY INDEX (a1)
PARTITION BY b1;
CREATE TABLE t2 (
  a2 INTEGER,
  b2 INTEGER,
  c2 INTEGER)
PRIMARY INDEX (a2);

```

The following MERGE request is valid because it specifies the partitioning column, *b1*, of the target table, *t1*, in its ON clause:

```

MERGE INTO t1
  USING t2
  ON a1=a2 AND b1=b2
WHEN MATCHED THEN

```

```
UPDATE SET c1=c2
WHEN NOT MATCHED THEN
  INSERT (a2, b2, c2);
```

The following MERGE request aborts and returns an error message to the requestor because it does not specify the partitioning column of the target table, *b1*, in its ON clause:

```
MERGE INTO t1
  USING t2
  ON a1=a2
WHEN MATCHED THEN
  UPDATE SET c1=c2
WHEN NOT MATCHED THEN
  INSERT (a2, b2, c2);
```

The following MERGE request aborts and returns an error message to the requestor because its INSERT specification orders columns *b2* and *c2* in a different sequence than they were specified in its ON clause. The INSERT specification must always match the ON clause constraints on the primary index of the target table, and its partitioning column set if the target table has row-partitioning.

```
MERGE INTO t1
  USING t2
  ON a1=a2 AND b1=b2
WHEN MATCHED THEN
  UPDATE SET c1=c2
WHEN NOT MATCHED THEN
  INSERT (a2, c2, b2);
```

If the target table has row-partitioning, the values of the partitioning columns must also be specified in *search\_condition*, and the INSERT clause must specify the same partitioning column values as *search\_condition*.

### Example: You Cannot Substitute the System-Derived PARTITION Column For the Partitioning Column Set For a MERGE Operation With a RPPI Target Table

Assume you have defined the following source and target relation definitions:

```
CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER,
  c1 INTEGER)
PRIMARY INDEX (a1)
```

```

PARTITION BY b1;

CREATE TABLE t2 (
  a2 INTEGER,
  b2 INTEGER,
  c2 INTEGER)
PRIMARY INDEX (a2),
UNIQUE INDEX(b2);

```

The following example fails because you cannot substitute the system-derived PARTITION column for the partitioning column set of the target relation in the ON clause.

```

MERGE INTO t1
  USING t2
  ON a1=a2 AND t1.PARTITION=10
WHEN MATCHED THEN
  UPDATE SET b1=10
WHEN NOT MATCHED THEN
  INSERT (a2, b2, c2);

```

This MERGE request is valid because it specifies conditions on the primary index of the target table, *a1*, and its partitioning column, *b1*:

```

MERGE INTO t1
  USING t2
  ON a1=a2 AND b1=10
WHEN MATCHED THEN
  UPDATE SET c1=c2;

```

The following MERGE request, again written against the same set of source and target tables, is not valid because it fails to specify the partitioning column for the target table, *b1*, in its ON clause.

```

MERGE INTO t1
  USING t2
  ON a1=a2
WHEN MATCHED THEN
  UPDATE SET c1=c2;

```

You *can* specify a system-derived PARTITION column-based condition in the ON clause, but only as a residual condition. For example, the following example works correctly because the primary index equality condition *a1=a2* and the target table partitioning column condition *b1=10* are both specified. The system treats the additional *t1.PARTITION* condition, *t1.PARTITION=25* as a residual condition only.

## Example: With a Guaranteed Single-Row Source Relation, You Do Not Need To Specify the Partitioning Column Set

Because the following MERGE request guarantees a single-row source relation by specifying an equality condition on the USI of target RPPI table *t1*, it is not necessary to specify the partitioning column set:

```
CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER,
  c1 INTEGER)
PRIMARY INDEX (a1)
PARTITION BY b1;

CREATE TABLE t2 (
  a2 INTEGER,
  b2 INTEGER,
  c2 INTEGER)
PRIMARY INDEX (a2),
UNIQUE INDEX(b2);
MERGE INTO t1
  USING (SELECT *
        FROM t2
        WHERE b2=10) AS s
  ON a1=1
WHEN MATCHED THEN
  UPDATE SET c1=c2
WHEN NOT MATCHED THEN
  INSERT (c2, b2, c2);
```

## Example: Using the Target Table as the Source Table

The following example uses target table *t1* as its source table:

```
MERGE INTO t1
  USING t1 AS s
  ON t1.a1 = s.a1
WHEN MATCHED THEN
  UPDATE SET b1 = 10
WHEN NOT MATCHED THEN
  INSERT VALUES (s.a1, s.b1, s.c1);
```

## Example: Using the BEGIN Period Bound Function as a Condition When Merging Into a RPPI Table

Assume you define the following two tables.

```
CREATE SET TABLE testing.t11 (
  a INTEGER,
  b DATE FORMAT 'YY/MM/DD',
  c DATE FORMAT 'YY/MM/DD')
PRIMARY INDEX (a);

CREATE SET TABLE testing.t12 (
  a INTEGER,
  b PERIOD(DATE),
  c INTEGER)
PRIMARY INDEX (a)
PARTITION BY RANGE_N((BEGIN(b) BETWEEN DATE '2009-01-01'
                      AND DATE '2011-12-31'
                      EACH INTERVAL '1' MONTH);
```

The following MERGE request merges rows from *t11* in to *t12*.

```
MERGE INTO t12
  USING t11
  ON t12.a = t11.a
  AND BEGIN (t12.b) = t11.b
WHEN MATCHED THEN
  UPDATE SET c= 4
WHEN NOT MATCHED THEN
  INSERT VALUES (t11.a, PERIOD(t11.b), 4);
```

## Example: Using the BEGIN and END Period Bound Functions as Conditions When Merging Into a RPPI Table

Assume you define the following two tables:

```
CREATE SET TABLE testing.t21 (
  a INTEGER,
  b DATE FORMAT 'YY/MM/DD',
  c DATE FORMAT 'YY/MM/DD')
PRIMARY INDEX (a);
```

```
CREATE SET TABLE testing.t22 (
  a INTEGER,
  b PERIOD (DATE))
PRIMARY INDEX (a)
PARTITION BY (CASE_N((END(b))<= DATE '2008-03-31',
                     (END(b))<= DATE '2008-06-30',
                     (END(b))<= DATE '2008-09-30',
                     (END(b))<= DATE '2008-12-31'),
              CASE_N((BEGIN(b))> DATE '2008-12-31',
                     (BEGIN(b))> DATE '2008-09-30',
                     (BEGIN(b))> DATE '2008-06-30',
                     (BEGIN(b))> DATE '2008-03-31',));
```

The following MERGE request merges rows from *t11* in to *t22*.

```
MERGE INTO t22
  USING t21
  ON t22.a = t21.a
  AND BEGIN(t22.b) = t21.b
  AND END(t22.b) = t21.c
WHEN MATCHED THEN
  UPDATE SET c=4
WHEN NOT MATCHED THEN
  INSERT VALUES (t21.a, PERIOD(t21.b, t21.c), 4);
```

## Example: Failure Because the Matching Condition is Defined on a PERIOD Bound Function

This example fails because the matching condition of the MERGE request is defined on a BEGIN Period bound function.

```
CREATE TABLE source(
  a INTEGER,
  b PERIOD(DATE),
  c INTEGER);

CREATE TABLE target(
  i INTEGER,
  j PERIOD(DATE),
  k INTEGER)
PRIMARY INDEX(i)
PARTITION BY Begin(j);

INSERT INTO source(1, PERIOD(DATE, UNTIL_CHANGED), 1);
```



```

MERGE INTO target
  USING source
  ON a=i AND BEGIN(j) = END(b)
WHEN MATCHED THEN
  UPDATE SET k=c ;

```

### Example: Invoking an SQL UDF From a MERGE Request

This example shows how to invoke an SQL UDF named *value\_expression* at several points within a MERGE request.

```

MERGE INTO t1
  USING (SELECT a2, b2, c2
        FROM t2
        WHERE test.value_expression(b2, c2))
        source_tbl(a2, b2, c2)
  ON a1 = source_tbl.a2 AND
     b1 = test.value_expression(source_tbl.b2, source_tbl.c2)
  WHEN MATCHED THEN
    UPDATE SET b1 = b2, c1 = test.value_expression(2,3)
  WHEN NOT MATCHED THEN
    INSERT (a2, test.value_expression(4,5), c2);

```

### Example: Merging into a Table with an Implicit Isolated Load Operation

For information on defining a load isolated table, see the WITH ISOLATED LOADING option for CREATE TABLE and ALTER TABLE in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

Following are the table definitions for the example:

```

CREATE TABLE ldi_table1,
  WITH CONCURRENT ISOLATED LOADING FOR ALL
  (a INTEGER,
   b INTEGER,
   c INTEGER)
PRIMARY INDEX ( a );

CREATE TABLE t1
  (c1 INTEGER,
   c2 INTEGER,
   c3 INTEGER)
PRIMARY INDEX ( c1 );

```

This statement performs a merge into the load isolated table ldi\_table1 from table t1 as an implicit concurrent load isolated operation:

```
MERGE WITH ISOLATED LOADING INTO ldi_table1
  USING t1 ON a=c1
  WHEN NOT MATCHED THEN INSERT
  VALUES (c1, c2, c3);
```

## Example: Merging into a Table with an Explicit Isolated Load Operation

For information on defining a load isolated table and performing an explicit isolated load operation, see the WITH ISOLATED LOADING option for CREATE TABLE and ALTER TABLE, in addition to Load Isolation Statements in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

Following are the table definitions for the example.

```
CREATE TABLE ldi_table1,
  WITH CONCURRENT ISOLATED LOADING FOR ALL
  (a INTEGER,
   b INTEGER,
   c INTEGER)
PRIMARY INDEX ( a );

CREATE TABLE t1
  (c1 INTEGER,
   c2 INTEGER,
   c3 INTEGER)
PRIMARY INDEX ( c1 );
```

This statement starts an explicit concurrent load isolated operation on table ldi\_table1:

```
BEGIN ISOLATED LOADING ON ldi_table1
  USING QUERY_BAND 'LDILoadGroup=Load1;';
```

This statement sets the session as an isolated load session:

```
SET QUERY_BAND='LDILoadGroup=Load1;' FOR SESSION;
```

This statement performs an explicit concurrent load isolated merge into table ldi\_table1 from table t1:

```
MERGE INTO ldi_table1
  USING t1 ON a=c1
  WHEN NOT MATCHED THEN INSERT
  VALUES (c1, c2, c3);
```

This statement ends the explicit concurrent load isolated operation:

```
END ISOLATED LOADING FOR QUERY_BAND 'LDILoadGroup=Load1;';
```

You can use this statement to clear the query band for the next load operation in the same session:

```
SET QUERY_BAND = 'LDILoadGroup=NONE;' FOR SESSION;
```

## Example: Executing a MERGE Update Request When Both the Target Table and the Source Table Have Row-Level Security Constraints

Assume that:

- The user logged onto this session has the `OVERRIDE UPDATE CONSTRAINT` row-level security privilege on `table_1`.
- Both `table_1` and `table_2` have the same set of row-level security constraints.

The update of `table_1` is valid and Teradata Database takes the constraint values for target table `table_1`, which are not specified in the request, from the constraint values defined for source table `table_2`.

```
MERGE INTO table_1 AS target
USING table_2 AS source
ON (target.col_1 = source.col_1)
WHEN MATCHED THEN
UPDATE SET level = source.col_2;
```

## Example: Application of Row-Level Security SELECT Constraints When User Lacks Required Privileges (MERGE Request)

This example shows how the `SELECT` constraint predicates are applied to the source rows when a user that does not have the required `OVERRIDE` privileges attempts to execute a `MERGE` request on a table that has the row-level security `SELECT` constraint. The `SELECT` constraint filters out the rows that the user submitting the `MERGE` request is not permitted to view. The `SELECT` constraint predicates are added to the `MATCH` condition.

An `EXPLAIN` statement is used to show the steps involved in the execution of the request and the outcome of the application of the constraint predicates.

The statements used to create the tables in this example are:

```
CREATE TABLE rls_src_tbl(
  col1 INT,
  col2 INT,
  classification_levels CONSTRAINT,
  classification_categories CONSTRAINT);
```

```
CREATE TABLE rls_tgt_tbl(
  col1 INT,
  col2 INT,
  classification_levels CONSTRAINT,
  classification_categories CONSTRAINT);
```

The user's sessions constraint values are:

```
Constraint1Name LEVELS
Constraint1Value 2
Constraint3Name CATEGORIES
Constraint3Value '90000000'xb
```

This EXPLAIN statement is used to show the steps involved in the execution of the MERGE request and the outcome of the application of the SELECT constraint predicates.

```
MERGE INTO rls_tgt_tbl
USING rls_src_tbl
ON (rls_tgt_tbl.col1=1)
WHEN MATCHED THEN
  UPDATE SET col2=3
WHEN NOT MATCHED THEN
  INSERT (1,1,rls_src_tbl.levels,rls_src_tbl.categories);
```

An EXPLAIN shows a RETRIEVE step from RS.rls\_src\_tbl with a condition of `((SYSLIB.SELECTCATEGORIES ('90000000'XB, RS.rls_src_tbl.categories ))= 'T') AND ((SYSLIB.SELECTLEVEL (2, RS.rls_src_tbl.levels ))= 'T'))`.

The results are merged with matched updates and unmatched inserts into RS.rls\_tgt\_tbl with a condition of `(((((SYSLIB.SELECTCATEGORIES ('90000000'XB, {RightTable}.CATEGORIES)) = 'T') AND ((SYSLIB.SELECTLEVEL (2, {RightTable}.LEVELS )) = 'T')) AND (RS.rls_tgt_tbl.col1 = Field_4)))`.

## ROLLBACK

### Purpose

Terminates and rolls back the current transaction.

For information about the temporal form of ROLLBACK, see *Teradata Vantage™ Temporal Table Support*, B035-1182.

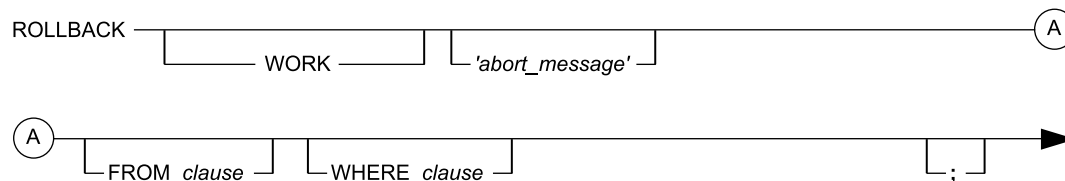
See also:

- [ABORT](#)
- [COMMIT](#)
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142

## Required Privileges

None.

## Syntax



## Syntax Elements

### WORK

Optional keyword.

### *abort\_message*

Text of the message to be returned when the transaction is terminated.

### FROM\_clause

A clause that is required only if the WHERE clause includes subqueries. You can code scalar subqueries as an expression within a derived table in FROM clause conditions in the same way you can specify scalar subqueries for a SELECT request. See [Scalar Subqueries](#) and [Rules for Using Scalar Subqueries in ABORT Statements](#). However, you cannot code a derived table as a scalar subquery.

See [FROM Clause](#).

### WHERE\_clause

A clause that introduces a conditional expression whose result must evaluate to TRUE if the transaction is to be rolled back. The expression can specify aggregate operations and scalar subqueries. For the rules and restrictions, see [Scalar Subqueries](#) and [Rules for Using Scalar Subqueries in ABORT Statements](#).

If the WHERE clause is omitted, termination is unconditional. See [WHERE Clause](#).

The WHERE condition specifies an expression whose result must evaluate to TRUE if termination is to occur. If the result is FALSE, transaction processing continues.

## ANSI Compliance

ROLLBACK is ANSI SQL:2011-compliant.

The following ROLLBACK options are Teradata extensions to the ANSI standard:

- *abort\_message*
- FROM\_clause

- WHERE\_clause

Other SQL dialects support similar non-ANSI standard statements with names such as the following:

- ROLLBACK TRANSACTION
- ROLLBACK WORK

## Usage Notes

### Definition and Termination of ANSI Transactions

In ANSI session mode, the first SQL request in a session initiates a transaction. The transaction is terminated by issuing either a COMMIT or a ABORT/ROLLBACK request. Request failures do not cause a rollback of the transaction, only of the request that causes them.

### ROLLBACK Is Explicit

There are no implicit transactions in ANSI session mode. More accurately, each ANSI transaction is initiated implicitly, but always completed explicitly. A COMMIT or ABORT/ROLLBACK request must always be explicitly stated and be the last request of a transaction in order for the transaction to terminate successfully.

In ANSI session mode, you must issue an ABORT/ROLLBACK (or COMMIT) request even when the only request in a transaction is a SELECT or SELECT AND CONSUME.

For a SELECT request, there is no difference between issuing a COMMIT or an ABORT/ROLLBACK.

For a SELECT AND CONSUME, there is a difference in the outcomes between issuing a COMMIT or ABORT/ROLLBACK because an ABORT or ROLLBACK request reinstates the subject queue table of the request to its former status, containing the rows that were pseudo-consumed by the aborted SELECT AND CONSUME request.

### Rules for Embedded SQL

ROLLBACK observes the following rules:

- ROLLBACK cannot be performed as a dynamic SQL statement.
- ROLLBACK is not valid when you specify the TRANSACT(2PC) option to the preprocessor.

### ROLLBACK and ABORT are Synonyms

With the exception of the presence of the WORK keyword, ROLLBACK is a synonym for ABORT.

The COMMIT statement also causes the current transaction to be terminated, but with commit rather than rollback.

See [ABORT](#) and [COMMIT](#).

## Actions Performed by ROLLBACK

ROLLBACK performs the following actions:

1. Backs out changes made to the database as a result of the transaction.
2. Deletes spooled output for the request.
3. Releases locks associated with the transaction.
4. If the transaction is in the form of a macro or a multistatement request, bypasses execution of the remaining statements.
5. Returns a failure response to the user.

## Actions Performed by ROLLBACK with Embedded SQL

ROLLBACK performs the following additional actions when used within an embedded SQL application:

1. Discards dynamic SQL statements prepared within the current transaction.
2. Closes open cursors.
3. Cancels database changes made by the current transaction.
4. Sets SQLCODE to zero when ROLLBACK is successful.
5. Sets the first SQLERRD element of the SQLCA to 3513 or 3514, depending on whether *abort\_message* is specified.
6. Returns an abort message in the SQLERRM field of the SQLCA if you specify a WHERE clause.
7. Terminates the application program connection (whether explicit or implicit) to Teradata Database if the RELEASE keyword is specified.

Environment	Preprocessor Action When a LOGON/CONNECT Request Does Not Precede the Next SQL Request
Mainframe-attached	Attempts to establish an implicit connection.
Workstation-attached	Issues a No Session Established error.

## ROLLBACK with a WHERE Clause

ROLLBACK tests each value separately, so a WHERE clause should not introduce both an aggregate and a nonaggregate value.

The aggregate value becomes, in effect, a GROUP BY value, and the mathematical computation is performed on the group aggregate results.

For example, assuming the following, the ROLLBACK statement that follows incorrectly terminates the transaction:

- The table Test contains several rows,
- The sum of Test.colA is 188, and
- Only one row contains the value 125 in Test.colB

```
ROLLBACK
WHERE (SUM(Test.colA) <> 188)
AND (Test.ColB = 125);
```

The preceding request is processed first by performing an all-rows scan with the condition (ColB = 125), which selects a single row and then computes intermediate aggregate results with the condition (SUM(ColA) <> 188).

The condition evaluates to TRUE because the value of ColA in the selected row is less than 188.

If ROLLBACK ... WHERE is used and the statement requires READ access to an object for execution, then user executing the ROLLBACK statement must have SELECT right to the data being accessed.

The WHERE condition of a ROLLBACK can include subqueries. The subqueries require FROM clauses and the ROLLBACK request should also have a FROM clause if you want the scope of a reference in a subquery to be the ROLLBACK condition.

## ROLLBACK with a UDT in the WHERE Clause

ROLLBACK supports comparisons of UDT expressions in a WHERE clause. A UDT expression is any expression that returns a UDT value.

If you specify a UDT comparison, then the UDT must have a defined ordering. See CREATE ORDERING in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## Rules for Using a Scalar UDF in a ROLLBACK Request

You can invoke a scalar UDF in the WHERE clause of a ROLLBACK request if the UDF returns a value expression. The scalar UDF must be invoked from within an expression that is specified as the search condition.

## Rules for Using Correlated Subqueries in a ROLLBACK Request

The following rules apply to correlated subqueries used in a ROLLBACK request:

- A FROM clause is required for a ROLLBACK request if the ROLLBACK references a table. All tables referenced in a ROLLBACK request must be defined in a FROM clause.
- If an inner query column specification references an outer FROM clause table, then the column reference must be fully qualified.

Correlated subqueries, scalar subqueries, and the EXISTS predicate are supported in the WHERE clause of a ROLLBACK request.



See and [SAMPLE Clause](#).

## Rules for Using Scalar Subqueries in ROLLBACK Requests

You can specify a scalar subquery in the WHERE clause of a ROLLBACK request in the same way you can specify one for a SELECT request.

You can also specify a ROLLBACK request with a scalar subquery in the body of a trigger. However, Teradata Database processes any noncorrelated scalar subquery you specify in the WHERE clause of a ROLLBACK statement in a row trigger as a single-column single-row spool instead of as a parameterized value.

## Multiple ROLLBACK Requests

If a macro or multistatement request contains multiple ROLLBACK requests, those requests are initiated in the order specified, even if the expressions could be evaluated immediately by the parser because the request does not access any tables. See [Two Types of ROLLBACK Requests](#). However, the system can process ROLLBACK requests in parallel.

If the system processes the ROLLBACK requests in parallel, and one of the requests actually does roll back, then the system reports that rollback regardless of its specified sequence in the macro or multistatement request.

You can examine the EXPLAIN report for the macro or multistatement request to determine whether the system is executing the ROLLBACK requests in parallel or not.

## Two Types of ROLLBACK Requests

There are two categories of ROLLBACK requests, those that can be evaluated by the parser and do not require access to a table and those that require table access.

If ROLLBACK expressions are processed that do not reference tables, and if their order of execution is not important relative to the other requests in a multistatement request or macro, then they should be placed ahead of any requests that reference tables so that the rollback operation can be done at minimum cost.

In the following example, the first two ROLLBACK requests can be evaluated by the parser and do not require access to tables. The third ROLLBACK request requires access to a table.

```
CREATE MACRO macro_name (
  p1 INTEGER,
  p2 INTEGER)
AS . . .
ROLLBACK 'error' WHERE :p1 < 0;
ROLLBACK 'error' WHERE :p2 < 0;
```

```
SELECT. . .
ROLLBACK 'error' WHERE tab.c1 = :p1;
```

## ROLLBACK with BTEQ

If you use ROLLBACK in a BTEQ script with either the .SESSION or the .REPEAT command, you must send the ROLLBACK statement and the repeated SQL statement as one request.

If you send the repeated request without the ROLLBACK, one of the requests is eventually blocked by other sessions and the job hangs because of a deadlock.

## Examples

### Example: ROLLBACK with a UDT in the WHERE Clause

The following examples show correct use of UDT expressions in the WHERE clause of a ROLLBACK request:

```
ROLLBACK WHERE (tab1.euro_col < CAST(0.0 AS euro));

ROLLBACK WHERE (tab1.cir_col.area() < 10.0);
```

### Example: Using an SQL UDF in the WHERE Clause of a ROLLBACK Request

The following ROLLBACK request specifies an SQL UDF in its WHERE clause search condition.

```
ROLLBACK FROM t1
WHERE a1 = test.value_expression(2,3);
```

## UPDATE

### Purpose

Modifies column values in existing rows of a table.

The UPDATE statement can take the following forms:

- Basic - updates one or more rows from a table.  
The Basic form can be specified with or without a FROM clause.
- Basic for Joined Tables

The Join Condition form updates rows from a table when the WHERE condition directly references columns in tables other than the one from which rows are to be updated; that is, if the WHERE condition includes a subquery or references a derived table.

- Positioned - see “UPDATE (Positioned Form)” in *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148.
- Temporal - see *Teradata Vantage™ ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ Temporal Table Support*, B035-1182.
- Upsert - see [UPDATE \(Upsert Form\)](#).

See also:

- [INSERT/INSERT ... SELECT](#)
- [MERGE](#)
- “UPDATE (Positioned Form)” in *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148
- [UPDATE \(Upsert Form\)](#)
- *Teradata Vantage™ ANSI Temporal Table Support*, B035-1186
- *Teradata Vantage™ Temporal Table Support*, B035-1182

## Required Privileges

The following privilege rules apply to the UPDATE statement:

- You must have the UPDATE privilege on the table or columns to be updated.
- To update any UDT column, you must also have the UDTUSAGE privilege on that column.
- When executing an UPDATE that also requires READ access to an object, you must have the SELECT privilege on the data being accessed.

For example, in the following request, READ access is required by the WHERE condition.

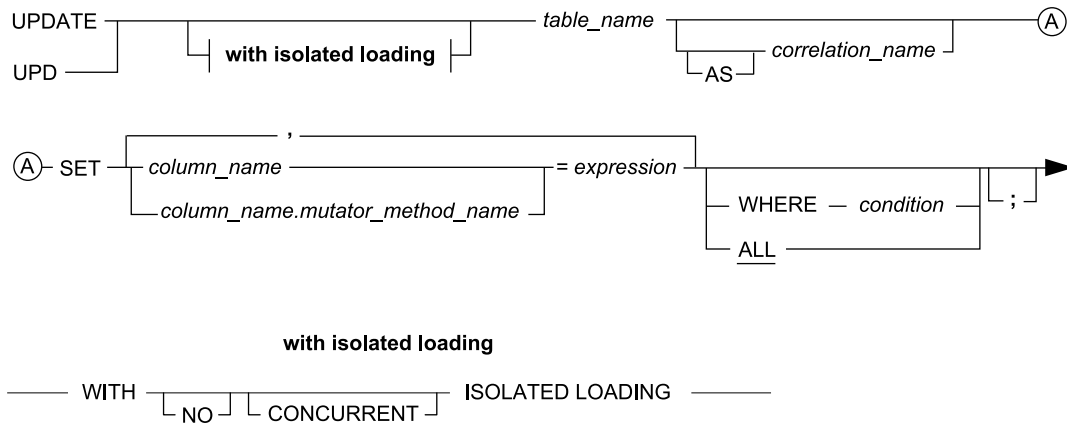
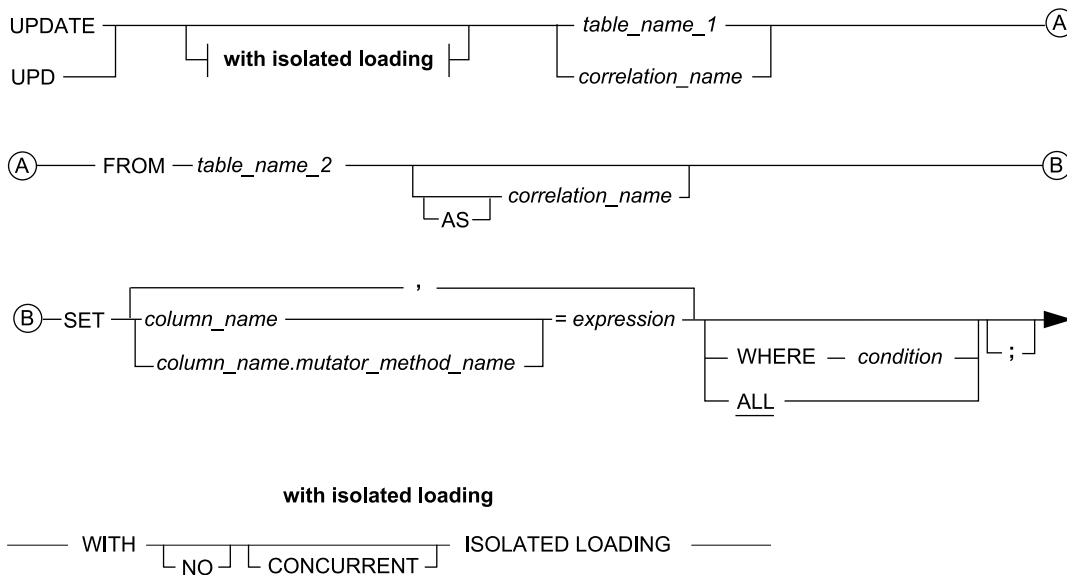
```
UPDATE table_1
SET column_1 = 1
WHERE column_1 < 0;
```

Similarly, the following request requires READ access because you must read *column\_1* values from *table\_1* in order to compute the new values for *column\_1*.

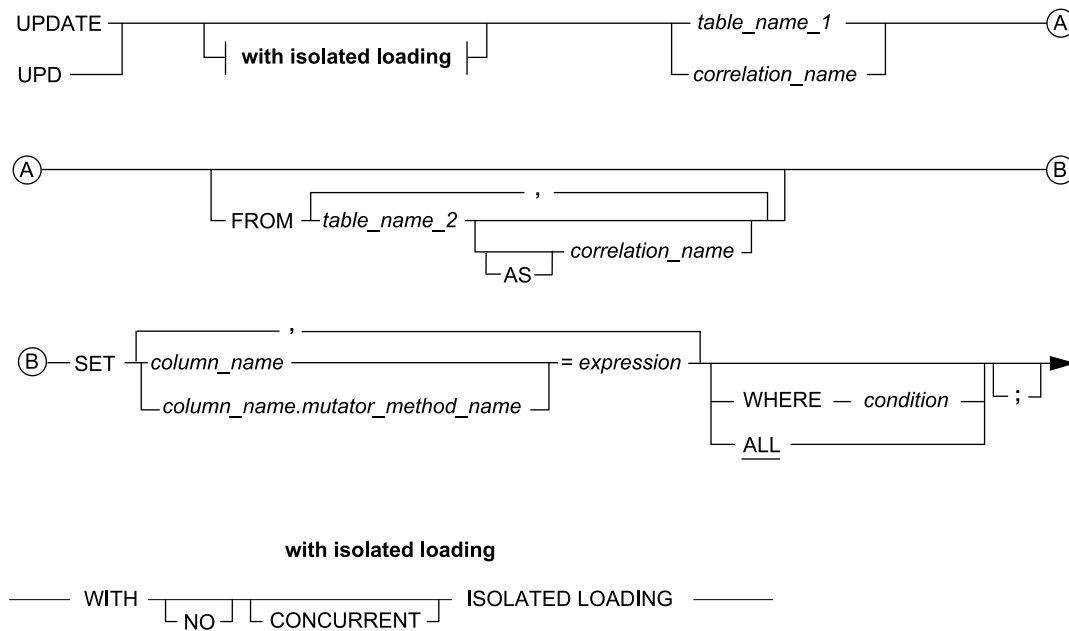
```
UPDATE table_1
SET field_1 = column_1 + 1;
```

The following request does not require SELECT privileges:

```
UPDATE table_1
SET column_3 = 0 ALL;
```

**Syntax - Basic Form, No FROM Clause****Syntax - Basic Form, FROM Clause**

## Syntax - Joined Tables Form



## Syntax Elements

### Isolated Loading Options

#### WITH ISOLATED LOADING

The **UPDATE** can be performed as a concurrent load isolated operation.

#### NO

The **UPDATE** is not performed as a concurrent load isolated operation.

#### CONCURRENT

Optional keyword that you can include for readability.

### Target Table Options

#### *table\_name\_1*

Name of the base table, queue table, or derived table to be updated, or the name of a view through which the table is accessed.

If you specify a correlation name for *table\_name\_1* in the **FROM** clause, then you must specify that correlation name for the updated table instead of *table\_name\_1*.

See [Example: UPDATE Specifying a Correlation Name for the Updated Table in the FROM Clause](#).

***correlation\_name***

An alias for *table\_name\_1*.

You cannot specify a correlation name for the table if the UPDATE statement includes the FROM clause.

Correlation names are also referred to as range variables.

The *correlation\_name* option is a Teradata extension to the ANSI SQL:2011 standard.

**FROM Clause****FROM**

Keyword introducing a table list of the updated table and any other tables from which field values are taken to update the updated table.

When you use an UPDATE syntax that requires a FROM clause, you should specify the names of all outer tables in the clause.

The UPDATE statement FROM clause is a Teradata extension to the ANSI SQL:2011 standard.

***table\_name\_2***

Name of one or more base tables, queue tables, derived tables, or views.

*table\_name\_1* must be a member of the *table\_name\_2* table list.

If you do not specify a correlation name for a *table\_name\_2* list object, or if you define a correlation name for it instead of *table\_name\_1*, then *table\_name\_2* cannot be a derived table.

If any *table\_name\_2* list member is a view, then that view must be updatable.

If you do not specify a database name, the system assumes the current database.

***correlation\_name***

Alias for a member of the *table\_name\_2* table list. Correlation names are also referred to as range variables.

A *correlation\_name* must be specified for at least one of the tables in a self-join operation.

If you specify a correlation name for *table\_name\_1* in the *table\_name\_2* table list using the joined tables syntax, you must specify that correlation name instead of the true name for *table\_name\_1*. See [Example: UPDATE Specifying a Correlation Name for the Updated Table in the FROM Clause](#).

**SET Clause****SET**

Names of one or more columns whose data is to be updated, and the expressions that are used for update.

If you are updating a UDT column, then you must use the mutator SET clause syntax. See [Updating Structured UDTs Using a Mutator SET Clause](#).

***column\_name***

Name of a column whose value is to be set to the value of the specified *expression*.

The *column\_name* field is for a column name only.

Do not use fully-qualified column name forms such as `databaseName.tableName.columnName` or `tableName.columnName`.

You cannot specify a derived period column name.

***mutator\_method\_name***

Name of a mutator method that is to perform some update operation on *column\_name*.

A mutator method name is the same name as the attribute name that it modifies. Within the mutated set clause, parentheses following the attribute name are not valid.

To update a structured UDT column, you must use the mutator SET clause syntax. See [Updating Structured UDTs Using a Mutator SET Clause](#).

***expression***

An expression that produces the value for which *column\_name* is to be updated.

*expression* can include constants, nulls (specified by the reserved word NULL), a DEFAULT function, or an arithmetic expression for calculating the new value. Values in a targeted row before the update can be referenced in an expression.

You can specify a scalar UDF for *expression* if it returns a value expression.

For join updates, you can reference columns in *expression* from rows participating in the join.

When host variables are used in the SET clause, they must always be preceded by a COLON character.

**WHERE**

A conditional clause. For more information see [WHERE Clause](#).

You can only specify a scalar UDF for *search\_condition* if it is invoked within an expression and returns a value expression.

If you specify a WHERE clause, you must have SELECT access on the searched objects.

***search\_condition***

Conditional expression to be used for determining rows whose values are to be updated. The condition can reference multiple tables or specify a scalar subquery. See [Scalar Subqueries](#) and [Rules for Using Scalar Subqueries in UPDATE Requests](#).

**ALL**

Indicates that all rows in the specified table are to be updated.

The ALL option is a Teradata extension to ANSI SQL.

**ANSI Compliance**

UPDATE is ANSI SQL:2011-compliant.

## Usage Notes

### Locks and Concurrency

An UPDATE operation sets a WRITE lock for the table, partitions, or row being updated. For a nonconcurrent isolated update on load isolated table, the update operation sets an EXCLUSIVE lock.

The lock set for SELECT subquery operations depends on the isolation level for the session, the setting of the AccessLockForUncomRead DBS Control field, and whether the subquery is embedded within a SELECT operation or within an UPDATE request.

Transaction Isolation Level	DBS Control AccessLockForUncomRead Field Setting	Default Locking Severity for Outer SELECT and Ordinary SELECT Subquery Operations	Default Locking Severity for SELECT Operations Embedded Within an UPDATE Request
SERIALIZABLE	FALSE	READ	READ
	TRUE		READ
READ UNCOMMITTED	FALSE		READ
	TRUE		ACCESS

For more information, see:

- “SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ - Database Utilities*, B035-1102

### Activity Count

The activity count in the success response (or ACTIVITY\_COUNT system variable for a stored procedure) for an UPDATE request reflects the total number of rows updated. If no rows qualify for update, then the activity count is zero.

### Duplicate Rows and UPDATE

It is not possible to distinguish among duplicate rows in a MULTiset table. Because of this, when a WHERE condition identifies a duplicate row, all of the duplicate rows are updated.



## Duplicate Row Checks

Unless a table is created as **MULTISET** (and without **UNIQUE** constraints) to allow duplicate rows, the system always checks for duplicate rows during the update process. The order in which updates are executed can affect the result of a transaction.

Consider the following example:

```
CREATE SET TABLE t1 (
  a INTEGER,
  b INTEGER)
PRIMARY INDEX (a);

INSERT INTO t1 VALUES (1,1);

INSERT INTO t1 VALUES (1,2);

UPDATE t1
SET b = b + 1
WHERE a = 1; /* fails */
UPDATE t1
SET b = b - 1
WHERE a = 1; /* succeeds */
```

The first **UPDATE** request fails because it creates a duplicate row.

If the order of the **UPDATE** requests is reversed, then both **UPDATE** requests succeed because the **UPDATE** does not result in duplicate rows.

```
CREATE SET TABLE t1 (
  a INTEGER,
  b INTEGER)
PRIMARY INDEX (a);

INSERT INTO t1 VALUES (1,1);

INSERT INTO t1 VALUES (1,2);

UPDATE t1
SET b = b - 1
WHERE a = 1; /* succeeds */
UPDATE t1
SET b = b + 1
WHERE a = 1; /* succeeds */
```

This mode is characteristic of both simple and join updates. Updates that affect primary or secondary index values, on the other hand, are implemented as discrete delete and insert operations.

## Large Objects and UPDATE

The behavior of truncated LOB updates differs in ANSI and Teradata session modes. The following table explains the differences in truncation behavior.

In this session mode ...	The following behavior occurs when non-pad bytes are truncated on insertion ...
ANSI	an exception condition is raised. The UPDATE fails.
Teradata	no exception condition is raised. The UPDATE succeeds: the truncated LOB is stored.

## UPDATE Processing Time

Processing time for an UPDATE operation is longer under the following conditions:

- When the FALLBACK option is specified for the table, because the rows in the secondary copy of the table must also be updated.
- When a column on which one or more indexes (secondary or primary) are defined is updated.

You can shorten the processing time for an UPDATE operation by using an indexed column in the WHERE clause of the UPDATE request.

Processing time can also vary between different syntaxes used to perform the identical update operation. Use the EXPLAIN modifier to determine which syntax form produces optimal processing time.

## Rules for Embedded SQL and Stored Procedures

The following rules apply to the searched form of the UPDATE statement:

- If host variable references are used, a COLON character must precede the host variable in the SET and WHERE clauses. The colon is optional otherwise, but strongly recommended.
- If the UPDATE request is specified with a WHERE clause and the specified search condition selects no rows, then the value +100 is assigned to SQLCODE and no rows are updated.
- The updated row is formed by assigning each update value to the corresponding column value of the original row. The resulting updated row must be NULL for any column that has the NOT NULL attribute or SQLCODE is set to -1002 and the row is not updated.

Update values are set in the corresponding row column values according to the rules for host variables.

- If the table identified by *table\_name* is a view specifying WITH CHECK OPTION, all values updated through that view must satisfy any constraints specified in any WHERE clause defined in the view.

## Queue Tables and UPDATE

The best practice is to avoid using the UPDATE statement on a queue table because the operation requires a full table scan to rebuild the internal queue table cache. You should reserve this statement for exception handling.

An UPDATE statement cannot be in a multistatement request that contains a SELECT and CONSUME request for the same queue table.

For more information on queue tables and the queue table cache, see *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

## Rules for Updating Partitioning Columns of a Row-Partitioned Table

The following rules apply to updating the partitioning columns of a row-partitioned table:

- For an UPDATE request that attempts to update the partitioning columns, the partitioning expression must result in a value between 1 and the number of partitions defined for that level.
- For an UPDATE request that attempts to insert a row, the partitioning expression for that row must result in a value between 1 and the number of partitions defined for that level.
- You cannot update the system-derived columns PARTITION and PARTITION#L1 through PARTITION#L62.
- Errors, such as divide by zero, can occur during the evaluation of a partitioning expression. The system response to such an error varies depending on the session mode in effect at the time the error occurs.

Session Mode	Expression evaluation errors roll back this work unit ...
ANSI	request that contains the aborted request.
Teradata	transaction that contains the aborted request.

Take care in designing your partitioning expressions to avoid expression errors.

- For the update to succeed, the session mode and collation at the time the table was created do not need to match the current session mode and collation. This is because the partition that contains a row to be updated is determined by evaluating the partitioning expression on partitioning column values using the table's session mode and collation.
- In updating rows in a table defined with a character partitioning, if the collation for the table is either MULTINATIONAL or CHARSET\_COLL and the definition for the collation has changed since the table was created, the system aborts any request that attempts to update a row in the table and returns an error to the requestor.

- If the partitioning expression for a table involves Unicode character expressions or literals and the system has been backed down to a release that has Unicode code points that do not match the code points that were in effect when the table or join index was defined, Teradata Database aborts any attempts to update rows in the table and returns an error to the requestor.

## Rules for Updating a Table with a Row-Partitioned Join Index

- For an UPDATE request that attempts to insert or update a row in a base table that causes an insert into a join index with row partitioning, the partitioning expression for that index row must result in a value between 1 and the number of partitions defined for that level.
- For an UPDATE request that attempts to insert or update a row in a base table that causes an update of an index row in a join index with row partitioning, the partitioning expression for that index row after the update must result in a value between 1 and the number of partitions defined for that level.
- Updating a base table row does not always cause inserts or updates to a join index on that base table. For example, you can specify a WHERE clause in the CREATE JOIN INDEX statement to create a sparse join index for which only those rows that meet the condition of the WHERE clause are inserted into the index, or, for the case of a row in the join index being updated in such a way that it no longer meets the conditions of the WHERE clause after the update, cause that row to be deleted from the index. The process for this activity is as follows:

1. The system checks the WHERE clause condition for its truth value after the update to the row.

Condition Result	Description
FALSE	System deletes the row from the sparse join index.
TRUE	System retains the row in the sparse join index and proceeds to stage b.

2. The system evaluates the new result of the partitioning expression for the updated row.

Partitioning Expression	Result
<ul style="list-style-type: none"> <li>◦ evaluates to null, or</li> <li>◦ is an expression that is not CASE_N or RANGE_N</li> </ul>	<p>Not between 1 and 65535 for the row.</p> <p>The system aborts the request. It does not update the base table or the sparse join index, and returns an error.</p>
<ul style="list-style-type: none"> <li>◦ evaluates to a value, and</li> <li>◦ is an expression that is not CASE_N or RANGE_N</li> </ul>	<p>Between 1 and 65535 for the row.</p> <p>The system stores the row in the appropriate partition, which might be different from the partition in which it was previously, and continues processing requests.</p>

- In updating rows in a table defined with a character partitioning, if a noncompressed join index with a character partitioning under either an MULTINATIONAL or CHARSET\_COLL collation sequence is defined on a table and the definition for the collation has changed since the join index was created, Teradata Database aborts any request that attempts to update a row in the table and returns an error

to the requestor whether the update would have resulted in rows being modified in the join index or not.

- If the partitioning expression for a noncompressed join index involves Unicode character expressions or literals and the system has been backed down to a release that has Unicode code points that do not match the code points that were in effect when the table or join index was defined, Teradata Database aborts any attempts to update rows in the table and returns an error to the requestor.
- You cannot update a base table row that causes an insert into a join index with row partitioning such that a partitioning expression for that index row does not result in a value between 1 and the number of row partitions defined for that level.
- You cannot update a base table row that causes an update of an index row in a join index with row partitioning such that a partitioning expression for that index row after the update does not result in a value between 1 and the number of row partitions defined for that level.

## Rules for Updating Column-Partitioned Tables

The update may be made in place if the columns being updated are only in ROW format column partitions of a nontemporal column-partitioned table, instead of being treated as a delete of the old row and an insert of the new row. This includes row-level security constraints that are implicitly updated due to the row being updated. An update in place can be done, with the following restrictions:

- A column of a primary AMP index or primary index must not be updated.
- A column of a unique secondary index defined on the target table must not be updated. Note that columns of a unique join index (UJI) defined (explicitly or implicitly) on the target table may be updated.
- All columns of any updated nonunique secondary index defined on the target table must belong to the same target column partition.
- A partitioning column must not be updated.
- All columns of any updated foreign or parent keys must belong to the same target column partition.
- All columns referenced in a check constraint where at least one of these columns is updated must all belong to the same target column partition.
- Any source column values from the target table used in the expression for the value to assign to a target column must come from the same target column partition as the target column.
- Cannot include updatable cursors.

## Identity Columns and UPDATE

You cannot update a GENERATED ALWAYS identity column.

## Updating of GENERATED ALWAYS Identity Columns and PARTITION Columns

You cannot update the following set of system-generated columns:

- GENERATED ALWAYS identity columns
- PARTITION
- PARTITION#L *n*

## Updating Distinct UDT Columns

To update a distinct type column, either of the following must be true:

- The updated value must be of the same distinct type.
- There must exist a cast that converts the type of the updated value to the distinct type of the column and the cast is defined with the AS ASSIGNMENT option (see "CREATE CAST" in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144).

By default, a distinct type has a system-generated cast of this type.

For example, suppose you have the following table definition:

```
CREATE TABLE table_1 (
  column1 euro,
  column2 INTEGER)
UNIQUE PRIMARY INDEX(column2);
```

Then the information in the following table is true:

Example	Comment
<pre>UPDATE table_1   SET column1 = 4.56  WHERE column2 = 5;</pre>	Valid if there is a cast defined with the AS ASSIGNMENT option, either system-generated or user-defined, that converts DECIMAL to euro.
<pre>UPDATE table_1   SET column1 = CAST(4.56 AS euro)  WHERE column2 = 5;</pre>	Valid if there is a cast defined with or without the AS ASSIGNMENT option, either system-generated or user-defined, that converts DECIMAL to euro.
<pre>USING (price decimal(6,2)) UPDATE table_1   SET column1 = (CAST (:price AS     euro)).roundup(0);</pre>	Valid if the roundup() method returns the euro type and if there is an appropriate cast definition, either system-defined or user-defined, that converts DECIMAL to euro. Because an explicit cast operation is used, the UDT cast need not have been defined using the AS ASSIGNMENT option. <ol style="list-style-type: none"> <li>1. Host variable :price is converted to euro.</li> <li>2. The roundup() method is invoked.</li> </ol>

Example	Comment
<pre>UPDATE table_1   SET column1 = column1.roundup(0) ;</pre>	Valid if the roundup() method returns the euro type.
<pre>UPDATE table_1   SET column1 = column3;</pre>	Valid column reference.
<pre>UPDATE table_1   SET column1 = NULL   WHERE column2 = 10;</pre>	Valid setting of a distinct column to NULL.

## Updating Structured UDT Columns

### Updating Structured UDTs Using a Mutator SET Clause

Mutator SET clauses provide a syntax for updating structured type columns. A mutator SET clause can only be used to update structured UDT columns (the specified column\_name in a mutator SET clause must identify a structured UDT column). Each mutator method name you specify must be a valid mutator method name for the respective structured type value.

A mutator method name is the same name as the attribute name that it modifies. Within the mutated set clause, parentheses following the attribute name are not valid.

There is one additional restriction on mutator SET clauses.

Consider the following example:

```
SET mycol.R = x,
    mycol.y = mycol.R() + 3
```

As implemented by Teradata, any column references in an expression refer to the value of the column in the row *before* the row is updated. The system converts the two example clauses to the following single equality expression:

```
mycol = mycol.R(x).y(mycol.R() + 3)
```

This is a deviation from the ANSI SQL:2011 standard.

According to the ANSI SQL:2011 standard, the column reference to *mycol* in the second example equality expression of the mutator SET clause should reflect the change made to it from the first equality expression of the mutator SET clause, the assignment of *x*.

The two equality expressions are converted to the following single equality expression:

```
mycol = mycol.R(x).y(mycol.R(x).R() + 3)
```

## Rules for Updating Rows Using Views

To update rows using a view through which the table is accessed, observe the following rules:

- You must have the UPDATE privilege on the view.

The immediate owner of the view (that is, the containing database for the view) must have the UPDATE privilege on the underlying object (view, base table, or columns) whose columns are to be updated, and the SELECT privilege on all tables that are specified in the WHERE clause.

- Each column of the view must correspond to a column in the underlying table (that is, none of the columns in the view can be derived using an expression).
- The data type definitions for an index column should match in the view definition and in the base table to which the view refers.

Although you can generally convert the data type of a view column (for example, from VARCHAR to CHARACTER), if that converted column is a component of an index, then the Optimizer does not use that index when the base table is updated because the data type of the recast column no longer matches the data type of the index column.

The resulting all-AMP, all-row scan defeats the performance advantages the index was designed for.

- No two view columns can reference the same table column.
- If the request used to define a view contains a WHERE clause WITH CHECK OPTION, then all values inserted through that view must satisfy the constraints specified in the WHERE clause.
- If a view includes a WHERE clause and does not specify WITH CHECK OPTION, then data can be inserted through the view that will not be visible through that view.

## Rules for Using Scalar Subqueries in UPDATE Requests

The following rules apply to using scalar subqueries in UPDATE requests:

- You can specify scalar subqueries in the FROM and WHERE clauses of an UPDATE request in the same way as you would for a SELECT request. See [Scalar Subqueries](#). You can only specify a scalar subquery in the FROM clause of an UPDATE request as an expression within a derived table. You *cannot*, however, code a derived table as a scalar subquery.
- You can specify scalar subqueries in the SET clause of an UPDATE request.
- When you specify a correlated scalar subquery in the SET clause, even if the request has no FROM clause, Teradata Database treats the update as a joined update. See [Example: UPDATE With a Noncorrelated Subquery in its WHERE Clause](#).
- You can specify an UPDATE statement with scalar subqueries in the body of a trigger.

However, Teradata Database processes any noncorrelated scalar subqueries specified in the FROM, WHERE, or SET clauses of an UPDATE statement in a row trigger as a single-column single-row spool instead of as a parameterized value.



## Rules for Using a Scalar UDF in an UPDATE Request

A scalar UDF can be invoked from both the SET clause and the WHERE clause of an UPDATE request. See [Example: UPDATE Using an SQL UDF Invocation in the SET and WHERE Clauses](#).

Clause	Usage rules are the same as those for invoking a scalar UDF from the ...
SET	Select list of a SELECT request. You must use the alias to reference the result of a scalar UDF invocation that has an alias.
WHERE	WHERE clause of a SELECT request. The scalar UDF must be invoked from within an expression that is specified as the search condition. Otherwise, the system returns an error. See <a href="#">WHERE Clause</a> .

## Rules for Using the DEFAULT Function With Update

The following rules apply to using the DEFAULT function with an UPDATE statement:

- The DEFAULT function takes a single argument that identifies a relation column by name. The function evaluates to a value equal to the current default value for the column. For cases where the default value of the column is specified as a current built-in system function, the DEFAULT function evaluates to the current value of system variables at the time the request is executed.

The resulting data type of the DEFAULT function is the data type of the constant or built-in function specified as the default unless the default is NULL. If the default is NULL, the resulting data type of the DEFAULT function is the same as the data type of the column or expression for which the default is being requested.

- The DEFAULT function has two forms. It can be specified as DEFAULT or DEFAULT (*column\_name*). When no column name is specified, the system derives the column based on context. If the column context cannot be derived, the request aborts and an error is returned to the requestor.
- You can specify a DEFAULT function without a column name argument as the expression in the SET clause. The column name for the DEFAULT function is the column specified as the *column\_name*. The DEFAULT function evaluates to the default value of the column specified as *column\_name*.
- You cannot specify a DEFAULT function without a column name argument as part of the expression. It must be specified by itself. This rule is defined by the ANSI SQL:2011 specification.
- You can specify a DEFAULT function with a column name argument in the source expression. The DEFAULT function evaluates to the default value of the column specified as the input argument to the DEFAULT function.

For example, DEFAULT(col2) evaluates to the default value of col2. This is a Teradata extension to the ANSI SQL:2011 specification.

- You can specify a DEFAULT function with a column name argument anywhere in an update expression. This is a Teradata extension to the ANSI SQL:2011 specification.
- When no explicit default value has been defined for a column, the DEFAULT function evaluates to null when that column is specified as its argument.

See *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145 for more information about the DEFAULT function.

## Rules for Using a PERIOD Value Constructor With UPDATE

See *Teradata Vantage™ SQL Date and Time Functions and Expressions*, B035-1211 for the rules on using PERIOD value constructors. Also see [Example: INSERT Using a PERIOD Value Constructor](#) for two examples of how to use PERIOD value constructors in UPDATE requests.

## Nonvalid Uses of UPDATE

An UPDATE operation causes an error message to be returned when any of the following conditions occur:

- The operation attempts to update a field using a value that violates a constraint (for example, UNIQUE or referential) declared for the column.
- The operation attempts to update a field using a value that is of a different numeric type than that declared for the column and the value cannot be converted correctly to the correct type.
- The operation attempts to update a VARCHAR column, and that operation causes the row to become identical to another row (except for the number of trailing pad characters), for a table not permitting duplicate rows.
- The operation attempts to update a CHARACTER column with a value that is not in the repertoire of the destination character data type.
- If in ANSI session mode, updating character data, where in order to comply with maximum length of the target column, non-pad characters are truncated from the source data. This update is valid in Teradata session mode.
- The operation attempts to update a row by using values that will create a duplicate of an existing row, for a table not allowing duplicate rows.
- The operation references objects in multiple databases without using fully-qualified names. Name resolution problems may occur if referenced databases contain tables or views with identical names and these objects are not fully qualified. Name resolution problems may even occur if the identically named objects are not themselves referenced.
- A JSON entity reference was used in the target portion of the SET clause.

## FROM Clause and UPDATE

The optional FROM list included in the UPDATE syntax is a Teradata extension to support correlated subqueries and derived tables in the search conditions for UPDATE.

Specify the FROM clause for the following reasons only:

- To provide the outer scope of a table with columns referenced in a subquery, making the subquery a correlated subquery.
- To permit references to a derived table.
- To specify joined tables for an update operation.

If a table is listed in the FROM clause for the UPDATE and not in the FROM clause for a subquery, then field references in the subquery are scoped at the outer level, making it a correlated subquery.

The following rules apply to the use of correlated subqueries in the FROM clause of an UPDATE request:

- If a FROM clause is specified for the UPDATE syntax you are using, then any correlation name used must be specified in the FROM clause.
- If a correlation name is specified for the updated table name in the FROM clause, this correlation name, rather than the original name, must be used as the *table\_name* that follows the UPDATE keyword. This FROM clause is optional if no joined tables are specified for an UPDATE.
- If an inner query column specification references an outer FROM clause table, then the column reference must be fully qualified.
- If the FROM clause is omitted, you can specify a correlation name for the *table\_name* that follows the UPDATE keyword.

Also see [Correlated Subqueries](#).

## UPDATEs With a Join

If a row from the updated table is joined with a row from another table in the FROM clause, and the specified WHERE condition for the request evaluates to TRUE for that row, then the row in the updated table is updated with columns referenced in the SET clause from the joined row.

When an UPDATE request specifies a join operation, the join is more efficient if the WHERE condition uses values for indexed columns in the joined tables.

Be aware that the order in which join updates are executed can affect the result of a transaction. See [Duplicate Row Checks](#).

## UPDATE Support for Load Isolated Tables

A nonconcurrent load isolated update operation on a load isolated table updates the matched rows in-place.

A concurrent load isolated update operation on a load isolated table logically deletes the matched rows and inserts the rows with the modified values. The rows are marked as deleted and the space is not reclaimed until you issue an ALTER TABLE statement with the RELEASE DELETED ROWS option. See *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## Examples

### Example: UPDATE to Set a Specific Value

The following request updates the yrsexp column in the employee table for employee Greene, who is employee number 10017:

```
UPDATE employee
SET yrs_exp = 16
WHERE empno = 10017;
```

### Example: UPDATE by a Percentage

The following request updates the employee table to apply a 10 percent cost of living increase to the salary for all employees:

```
UPDATE employee
SET salary = salary * 1.1 ALL ;
```

### Example: UPDATE to Set a Null Value

The following request places a null in the salary column for employee number 10001:

```
UPDATE employee
SET salary = NULL
WHERE emp_no = 10001 ;
```

### Example: Updating a Nonpartitioned Column with an Equality Constraint

This example shows an update of a nonpartitioned column with an equality constraint on the partitioning column. With partition level locking, an all-AMP PartitionRange lock is used. The partition list contains a single partition pair.

The table definition for this example is as follows:

```
CREATE TABLE HLSDS.SLPPIT1 (PI INT, PC INT, X INT, Y INT)
  PRIMARY INDEX (PI)
  PARTITION BY (RANGE_N(PC BETWEEN 1 AND 10 EACH 1));
```

An EXPLAIN of the UPDATE statement shows the partition lock:

```
Explain UPDATE HLSDS.SLPPIT1 SET X = 3 WHERE PC = 10;
1) First, we lock HLSDS.SLPPIT1 for write on a reserved rowHash in
   a single partition to prevent global deadlock.
2) Next, we lock HLSDS.SLPPIT1 for write on a single partition.
3) We do an all-AMPs UPDATE from a single partition of HLSDS.SLPPIT1
   with a condition of ("HLSDS.SLPPIT1.PC = 10") with a residual
   condition of ("HLSDS.SLPPIT1.PC = 10"). The size is estimated
   with no confidence to be 1 row. The estimated time for this
   step is 0.05 seconds.
4) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.
```

## Example: UPDATE Specifying a Correlation Name for the Updated Table in the FROM Clause

Because the following UPDATE request specifies the correlation name e for employee in its FROM clause, you must specify e in place of employee immediately following the UPDATE keyword.

```
UPDATE e
FROM employee AS e, department AS d
SET salary = salary * 1.05
WHERE e.emp_no = d.emp_no;
```

## Example: UPDATE With a Subquery in its WHERE Clause

The following examples perform the same UPDATE operation using either a subquery join with a derived table or a simple equality join on column\_1 in table\_1 and table\_2.

```
UPDATE table_1
SET column_1 = 1
WHERE column_1 IN (SELECT column_2
                   FROM table_2);

UPDATE table_1
SET column_1 = 1
WHERE table_1.column_1=table_2.column_1;
```

### Example: UPDATE With a Noncorrelated Subquery in its WHERE Clause

The following UPDATE operation uses a subquery to decrease the list price on an electroacoustic CD budget line for labels containing the string 'erstwhile':

```
UPDATE disc
SET price = price * .85
WHERE label_no IN (SELECT label_no
                   FROM label
                   WHERE label_name LIKE '%erstwhile%')
AND   line = 'NEA';
```

You can obtain the same result by writing the query using a join between the disc and label tables on label\_no:

```
UPDATE disc
SET price = price * .85
WHERE disc.label_no = label.label_no
AND   label_name LIKE '%erstwhile%'
AND   line = 'NEA';
```

### Example: UPDATE With a Scalar Noncorrelated Subquery in its SET Clause

You can specify a scalar subquery in the same way that you specify a column or constant in the SET clause of an UPDATE request.

The following example specifies a scalar subquery in its SET clause:

```
UPDATE sales_sum_table AS sst
SET  total_sales = (SELECT SUM(amount)
                   FROM sales_table AS s
                   WHERE s.day_of_sale BETWEEN sst.period_start
                                                AND   sst.period_end);
```

### Example: UPDATE With a Scalar Correlated Subquery in its SET Clause

When a scalar correlated subquery is specified in the SET clause, even if the UPDATE request does not have a FROM clause, Teradata Database treats the update as a joined update because of the scalar correlated subquery.

The following example requires salesumtable to be joined with salestable, and the update is done using a merge update operation via a spool.

```
UPDATE sales_sum_table AS sst
SET total_sales = (SELECT SUM(amount)
                  FROM sales_table AS s
                  WHERE s.day_of_sale BETWEEN sst.period_start
                                      AND sst.period_end);
```

### Example: UPDATE With a Correlated Subquery in its WHERE Clause

For a noncorrelated subquery, change the subquery to include all tables it references in the inner FROM clause.

```
UPDATE publisher
SET pubnum = NULL
WHERE 0 = (SELECT COUNT(*)
          FROM book, publisher
          WHERE book.pubnum = publisher.pubnum);
```

The request does not contain a correlated subquery and the condition in the subquery has a local defining reference. The count, determined once, is nonzero, and no rows are deleted.

For additional examples of correlated and noncorrelated subqueries, see [Correlated Subqueries](#).

### Example: UPDATE With a Noncorrelated Subquery in its WHERE Clause

The following UPDATE operation uses a subquery to decrease the list price on an electroacoustic CD budget line for labels containing the string 'erstwhile':

```
UPDATE disc
SET price = price * .85
WHERE label_no IN (SELECT label_no
                  FROM label
                  WHERE label_name LIKE '%erstwhile%')
AND line = 'NEA';
```

You can obtain the same result by writing the query using a join between the disc and label tables on label\_no:

```
UPDATE disc
SET price = price * .85
WHERE disc.label_no = label.label_no
AND label_name LIKE '%erstwhile%'
AND line = 'NEA';
```

## Example: UPDATE With a Join

The following example updates the employee table to give each employee a 5 percent salary increase using a join between the employee and department tables.

```
UPDATE employee
FROM department AS d
SET salary = salary * 1.05
WHERE employee.dept_no = d.dept_no
AND salary_pool > 25000;
```

## Example: UPDATE Using an SQL UDF Invocation in the SET and WHERE Clauses

Assume you have created an SQL UDF named value\_expression. The following example updates table t1 using the result of value\_expression when it is passed values for columns a1 and b1 to update columns a1 and b1 for rows where the invocation of value\_expression using the values for columns c1 and d1 are greater than 5.

```
UPDATE t1
SET b1 = test.value_expression(t1.a1, t1.b1)
WHERE test.value_expression(t1.c1, t1.d1) > 5;
```

## Example: UPDATE With a Mutator SET Clause

The mutator SET clause syntax permits you to write UPDATE requests for structured UDTs in a form of shorthand notation. Consider the following structured data type definitions:

```
CREATE TYPE person AS (
  last_name VARCHAR(20),
  first_name VARCHAR(20),
  birthdate DATE)
...
;

CREATE TYPE school_record AS (
  school_name VARCHAR(20),
  gpa          FLOAT)
INSTANTIABLE
...
;
```



```
CREATE TYPE college_record AS (
    school  school_record,
    major   VARCHAR(20),
    minor   VARCHAR(20))
INSTANTIABLE
...
;
```

Suppose you have the following table definition:

```
CREATE TABLE student_record (
    student_id  INTEGER,
    student     person,
    high_school school_record,
    college     college_record);
```

Without the mutator SET clause notation, UPDATES must be written in forms of chained or nested mutator method invocations. For example:

- The following UPDATE request changes student John Doe to Natsuki Tamura:

```
UPDATE student_record
SET student = student.Last_name('Tamura').First_name('Natsuki')
WHERE student.First_name() = 'John'
AND student.Last_name() = 'Doe';
```

- The following UPDATE request makes the following updates to the college record of student Steven Smith:
  - school\_name = 'UCLA'
  - GPA = 3.20
  - major = 'Computer Science'

```
UPDATE student_record
SET college =
    college.school(college.school().school_name('UCLA')
    .GPA(3.20)).major('Computer Science')
WHERE student.First_name() = 'Steven'
AND student.Last_name() = 'Smith';
```

Formulating the necessary chained or nested mutator method invocations can be very complex. However, you can use the mutator SET clause notation to make writing UPDATE requests for a structured type column simpler and more intuitive.

Consider these rewrites of the two previous UPDATE requests (these use the mutator SET clause notation):

- The following UPDATE request changes student John Doe to Natsuki Tamura:

```
UPDATE student_record
  SET student.Last_name = 'Tamura',
      student.First_name = 'Natsuki',
  WHERE student.First_name() = 'John'
  AND   student.Last_name() = 'Doe';
```

- The following UPDATE request makes the following updates to the college record of student Steven Smith:
  - school\_name = 'UCLA'
  - GPA = 3.20
  - major = 'Computer Science'

```
UPDATE student_record
  SET college.school.school_name = 'UCLA',
      college.school.GPA = 3.20,
      college.major = 'Computer Science'
  WHERE student.First_name() = 'Steven'
  AND   student.Last_name() = 'Smith';
```

## Example: Update Requests Using the DEFAULT Function

You can specify a DEFAULT function without a column name as the expression in the SET clause.

The column name for the DEFAULT function is the column specified as the column name. The DEFAULT function evaluates to the default value of the column specified as *column\_name*.

The examples below assume the following table definition:

```
CREATE TABLE table_11 (
  col_1 INTEGER,
  col_2 INTEGER DEFAULT 10,
  col_3 INTEGER DEFAULT 20,
  col_4 CHARACTER(60));
```

The following UPDATE requests are valid:

The following example updates the values of col3 to 20 (its default value) for all rows:

```
UPDATE table_11
  SET col_3 = DEFAULT;
```

The following example updates the values of Col3 to 20 (its default value) for rows where the value of col1=5.

```
UPDATE table_11
  SET col_3 = DEFAULT
  WHERE Col1 = 5;
```

Assume the following table definition for the next example:

```
CREATE TABLE table_12 (
  x INTEGER,
  y INTEGER);
```

The following example updates the values of col3 to 20 (its default value) depending on whether the WHERE condition evaluates to true or not.

```
UPDATE table_11
SET col_3 = DEFAULT
WHERE 5 < ANY
(SELECT y
FROM table_12);
```

You can specify a DEFAULT function with a column name in the source expression. The DEFAULT function evaluates to the default value of the column name specified as the input argument to the DEFAULT function. For example, DEFAULT(col\_2) evaluates to the default value of col\_2. This is a Teradata extension.

The following UPDATE request is valid. The input argument to the DEFAULT function is col\_2. Therefore, the DEFAULT function evaluates to the default value of the col\_2 and then set col\_3 to this value. Specifically, it updates the values of col\_3 to 10 (the default value of col\_2) for all rows.

```
UPDATE table_11
SET col3 = DEFAULT(col_2);
```

The following example updates the values of col\_3 to 10 (the default value of col\_2) for rows where the value of col\_1 is 5.

```
UPDATE table_11
SET col_3 = DEFAULT(col_2)
WHERE col_1 = 5;
```

You can specify a DEFAULT function with a column name anywhere in the update expression. This is a Teradata extension to the ANSI SQL:2011 standard.

The following UPDATE request is valid. The input argument to the DEFAULT function is col\_2; therefore, the DEFAULT function evaluates to the default value of col\_2. The request then updates the value of col\_3 to 15 (10+5, the default value of col\_2 + 5) for all rows.

```
UPDATE table_11
SET col_3 = DEFAULT(col_2) + 5;
```

The following example updates the value of col3 to 15 (the default value of col2+5) for all rows.

```
UPDATE table_11
SET col_3 = DEFAULT(col_2) + 5 ALL;
```

The following example updates the values of col3 to 15 (the default value of col\_2 + 5) for rows where the value of col\_1=20.

```
UPDATE table_11
  SET col_3 = DEFAULT(col_2)+5
  WHERE col_1 = 20;
```

When there is no explicit default value associated with the column, the DEFAULT function evaluates to null.

Assume the following table definition for the examples that follow:

```
CREATE TABLE table_13 (
  col_1 INTEGER,
  col_2 INTEGER NOT NULL,
  col_3 INTEGER NOT NULL DEFAULT NULL,
  col_4 INTEGER CHECK (col_4>100) DEFAULT 99 );
```

In the following example, col\_1 is nullable and does not have an explicit default value associated with it; therefore, the DEFAULT function evaluates to null.

```
UPDATE table_13
  SET col_1 = DEFAULT;
```

The following UPDATE requests are equivalent. For both requests, the DEFAULT function evaluates to the default value of col\_3 for rows where the value of col\_1 is 5.

```
UPDATE table_11
  SET col_3 = DEFAULT(c3)
  WHERE col_1 = 5;
UPDATE table_11
  SET col_3 = DEFAULT
  WHERE col_1 = 5;
```

## Example: UPDATE Using a PERIOD Value Constructor

The following example uses tables t1 and t2, which are defined as follows:

```
CREATE TABLE t1 (
  c1 INTEGER
  c2 PERIOD(DATE))
  UNIQUE PRIMARY INDEX (c1);

CREATE TABLE t2 (
  a INTEGER
  b DATE
```

```
c DATE)
UNIQUE PRIMARY INDEX (a);
```

The following two UPDATE requests both use a PERIOD value constructor:

```
UPDATE t1
SET c2 = PERIOD(DATE '2007-02-03', DATE '2008-02-04'));
UPDATE t1 FROM t2
SET c2 = PERIOD(b,c)
WHERE t2.a = 2;
```

### Example: Updating a NoPI Table

The following UPDATE request updates the NoPI table nopi012\_t1 aliased as t1.

```
UPDATE t1
FROM nopi012_t1 AS t1, nopi012_t2 AS t2
SET c3 = t1.c3 * 1.05
WHERE t1.c2 = t2.c2;
```

### Example: Updating a Table with an Implicit Isolated Load Operation

For information on defining a load isolated table, see the WITH ISOLATED LOADING option for CREATE TABLE and ALTER TABLE in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

Following is the table definition for the example.

```
CREATE TABLE ldi_table1,
  WITH CONCURRENT ISOLATED LOADING FOR ALL
  (a INTEGER,
   b INTEGER,
   c INTEGER)
PRIMARY INDEX ( a );
```

This statement performs an update on the load isolated table ldi\_table1 as an implicit concurrent load isolated operation:

```
UPDATE WITH CONCURRENT ISOLATED LOADING ldi_table1 SET b = b + 1;
```

An EXPLAIN shows that the UPDATE step is performed as concurrent load isolated. The isolated load begins and performs a concurrent load isolated UPDATE of ldi\_table1 (Load Uncommitted) with a condition of ("(ldi\_table1.TD\_ROWLOADID\_DEL = 0) AND ((1=1))").

## Example: Updating a Table with an Explicit Isolated Load Operation

For information on defining a load isolated table and performing an explicit isolated load operation, see the WITH ISOLATED LOADING option for CREATE TABLE and ALTER TABLE, in addition to Load Isolation Statements in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

Following is the table definition for the example:

```
CREATE TABLE ldi_table1,
  WITH CONCURRENT ISOLATED LOADING FOR ALL
  (a INTEGER,
   b INTEGER,
   c INTEGER)
PRIMARY INDEX ( a );
```

This statement starts an explicit concurrent load isolated operation on table ldi\_table1:

```
BEGIN ISOLATED LOADING ON ldi_table1 USING QUERY_BAND 'LDILoadGroup=Load1;';
```

This statement sets the session as an isolated load session:

```
SET QUERY_BAND='LDILoadGroup=Load1;' FOR SESSION;
```

This statement performs an explicit concurrent load isolated update on table ldi\_table1:

```
UPDATE ldi_table1 SET c = c + 1;
```

This statement ends the explicit concurrent load isolated operation:

```
END ISOLATED LOADING FOR QUERY_BAND 'LDILoadGroup=Load1;';
```

You can use this statement to clear the query band for the next load operation in the same session:

```
SET QUERY_BAND = 'LDILoadGroup=NONE;' FOR SESSION;
```

## Example: Application of Row-Level Security SELECT and UPDATE Constraints When User Lacks Required Privileges (UPDATE Request)

This example shows how the SELECT and UPDATE constraints are applied when a user that does not have the required privileges submits an UPDATE request in an attempt to update the classification level value for a row. The SELECT constraints filter out the rows that the user is not permitted to access and the UPDATE constraints restrict the user from executing the update operation on the target row.

The classification level value is stored in the classification\_level column, one of the constraint columns. The other constraint column is classification\_categories.

The statement used to create the table in this example is:

```
CREATE TABLE rls_tbl(
  col1 INT,
```

```
col2 INT,
classification_levels CONSTRAINT,
classification_categories CONSTRAINT);
```

The user's sessions constraint values are:

```
Constraint1Name LEVELS
Constraint1Value 2
Constraint3Name CATEGORIES
Constraint3Value '90000000'xb
```

Following is the UPDATE statement:

```
UPDATE rls_tbl SET col1=2 where col1=1;
```

The EXPLAIN shows the outcome of the SELECT and UPDATE constraints. A RETRIEVE step on RS.rls\_tbl is performed by way of the primary index "RS.rls\_tbl.col1 = 1" with a residual condition of ("((SYSLIB.SELECTLEVEL (2, RS.rls\_tbl.levels ))= 'T') AND ((SYSLIB.SELECTCATEGORIES ('90000000'XB, RS.rls\_tbl.categories ))='T')").

Next, there is a MERGE DELETE of the results to RS.rls\_tbl with the updated rows constrained by (RS.rls\_tbl.levels = SYSLIB.UPDATELEVEL (2, {LeftTable}.levels)), (RS.rls\_tbl.categories = SYSLIB.UPDATECATEGORIES ('90000000'XB, {LeftTable}.categories)).

Then, there is a MERGE into RS.rls\_tbl of the results.

## UPDATE (Upsert Form)

### Purpose

Updates column values in a specified row and, if the row does not exist, inserts the row into the table with a specified set of initial column values. The table must have a primary index and cannot be column partitioned.

For details on the temporal form, see *Teradata Vantage™ Temporal Table Support*, B035-1182.

See also:

- [INSERT/INSERT ... SELECT](#)
- [MERGE](#)
- [UPDATE](#)

### Required Privileges

The following privilege rule applies to both the UPDATE and INSERT portions of this statement:

- You must have *both* update and insert privileges on the base table, view, or column set to be updated regardless of which portion of the request is performed.

The following privilege rules applies to the UPDATE portion of this statement:

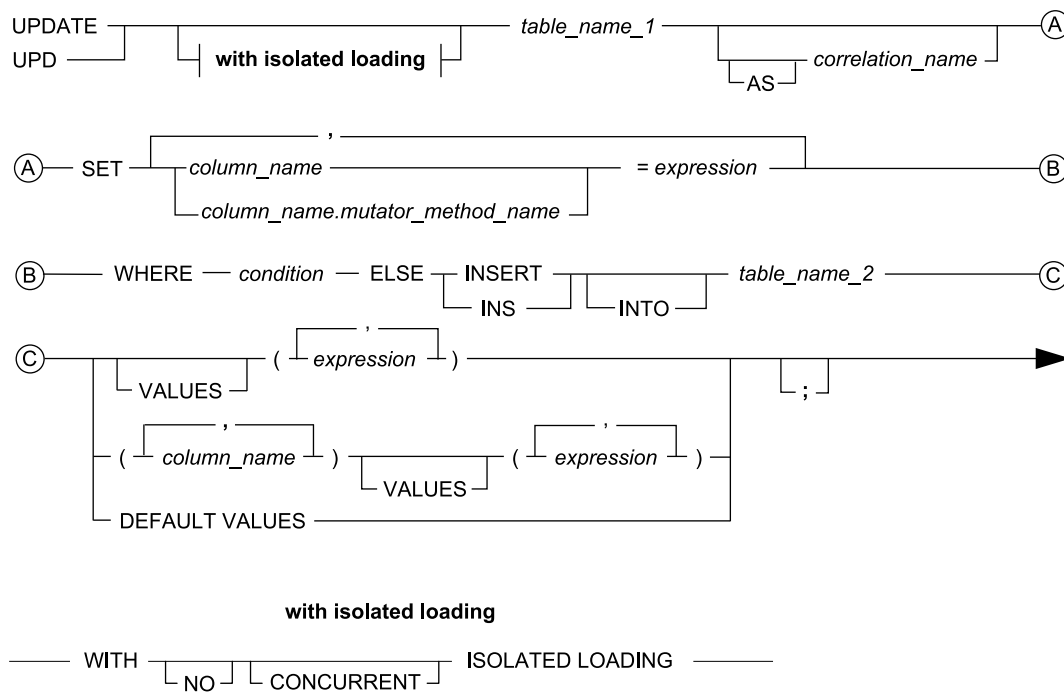
- You must have the UPDATE privilege on the base table, view, or column set to be updated.
- To update rows through a view, you must have the UPDATE privilege on the view.
- Also, the immediate owner of the view (that is, the database in which the view resides) must have the UPDATE privilege on the underlying object (view or base table).

The following privilege rules apply to the INSERT portion of this statement:

- You must have the INSERT privilege on the referenced table.
- To insert rows into a table through a view, you must have the INSERT privilege on the view.

Also, the immediate owner of the view (that is, the database in which the view resides) must have the INSERT privilege on the underlying object (view or base table).

## Syntax



## Syntax Elements

### Isolated Loading Options

#### WITH ISOLATED LOADING

The UPDATE can be performed as a concurrent load isolated operation.

#### NO

The UPDATE is not performed as a concurrent load isolated operation.



**CONCURRENT**

Optional keyword that can be included for readability.

**Target Table Options*****table\_name\_1***

Name of the table or queue table in which the row set is to be updated, or the name of the view through which the base table is accessed. The table must have a primary index and can be row-partitioned but cannot be column partitioned.

This must match the specification for *table\_name\_2*.

**SET Clause*****column\_name = expression***

Value expression to be updated in column *column\_name*. This can be a DEFAULT function.

Host variables in the SET clause must always be preceded by a COLON character.

You cannot specify a derived period column name.

***mutator\_method\_name***

name of a mutator method that is to perform some update operation on *column\_name*.

See [Updating Structured UDTs Using a Mutator SET Clause](#).

**WHERE Clause*****condition***

The predicate that specifies the row to be updated:

- If the UPDATE target is a SET table with a UPI, then only one row can be updated per request.
- If the UPDATE target is a SET table with a NUPI or a MULTiset table, then multiple rows can be updated per request.

***table\_name\_2***

Name of the table or queue table in which the row set is to be inserted, or the name of the view through which the base table is accessed.

This name must match the name specified for *table\_name\_1*.

**VALUES Clause*****expression***

The non-default set of values to be inserted into *table\_name\_2*.

You must specify values or nulls for *each* column defined for the table named *table\_name\_2* or the insert fails.

## DEFAULT

### DEFAULT (*column\_name*)

The DEFAULT function should update the column with its default value in the position by which it is called in the expression list.

If you specify DEFAULT (*column\_name*), then the DEFAULT function updates the default value for the column specified by *column\_name*, and the position of the DEFAULT call in the expression list is ignored.

### *column\_name*

Column set into which the values specified by *expression* are to be inserted.

## DEFAULT VALUES

A row consisting of default values is to be added to *table\_name*.

If any column does not have a defined DEFAULT phrase, then the process inserts a null for that column.

## ANSI Compliance

The upsert form of UPDATE is a Teradata extension to the ANSI SQL:2011 standard.

## Usage Notes

### Definition of Upsert

A simple upsert is a standalone update operation for which a subsequent standalone insert operation is coded that is performed only if the specific row to be updated is found not to exist. These actions require the performance of two individual SQL requests in the case where no row is found to update. This was the only means of performing a conditional insert on an update failure prior to the introduction of the upsert form of the UPDATE statement.

The more high-performing atomic upsert operation, represented by the upsert form of the UPDATE statement, is a single SQL request that includes both UPDATE and INSERT functionality. The specified update operation performs first, and if it fails to find a row to update, then the specified insert operation performs automatically.

### Purpose of the Atomic Upsert Operation

The single-pass upsert is described as atomic to emphasize that its component UPDATE and INSERT SQL statements are grouped together and performed as a *single*, or atomic, SQL request.

Existing Teradata Parallel Data Pump scripts that use the upsert feature in its Teradata Parallel Data Pump-specific form, using paired UPDATE and INSERT requests, do not have to be changed to take advantage of the atomic upsert capability because Teradata Parallel Data Pump automatically converts an UPDATE ... INSERT request pair in its older syntax into a corresponding atomic upsert request whenever appropriate.

The atomic upsert syntax can also be used by any CLv2-based applications and is particularly useful when coded directly into BTEQ scripts.

## Rules for Using Atomic Upsert

The following table describes the fundamental constraints on an atomic upsert operation.

Constraint	Explanation
The UPDATE and INSERT components of the upsert operation must specify the same table.	The purpose of an atomic upsert is to first attempt to update a particular row in a table and then, if that row is not found, to insert it into the table. By specifying different tables for the UPDATE and INSERT components of the statement, the intent of the operation is violated.
The table must have a primary index, which can be row partitioned but not column partitioned.	A primary index is needed to search for the row.
<p>The UPDATE and INSERT components of the upsert operation must specify the same row.</p> <p>In other words, the primary index value for the inserted row is identical to the primary index value for the targeted update row.</p>	<p>If an upsert operation cannot find the specific row it targets, then it inserts the row it would have otherwise updated, inserting the specific “update” values into the fields that would have been updated had the row existed in the table.</p> <p>This constraint is met when the primary index value specified by the WHERE clause of the UPDATE component matches the primary index value implied by the column values specified in the INSERT component. Because the value of the number generated for an INSERT into an identity column is not knowable in advance, you cannot perform an upsert operation into a target table that has an identity column as its primary index.</p> <p>Upserts into identity column tables for which the identity column is not the primary index are valid.</p>
The UPDATE component fully specifies the primary index value to ensure that all accesses are one-AMP operations.	<p>When the UPDATE component of the statement fully specifies the primary index, the system accesses any targeted row with a single-AMP hashed operation.</p> <p>This rule is applied narrowly in the Teradata Parallel Data Pump case, where it is taken to mean that the primary index is specified in the WHERE clause of the UPDATE component as equality constraints using simple constraint values (either constants or USING clause references to imported data fields). Simple values are also assumed for constraints on non-index columns and for the update values in the SET clause, avoiding any costly subqueries or FROM clause references. Similarly, TPump has a performance-driven preference to avoid subqueries in the INSERT.</p>

When you perform an upsert UPDATE to a row-partitioned table, the following rules must be followed or an error is returned:

- The values of the partitioning columns must be specified in the WHERE clause of the UPDATE clause of the statement.
- The INSERT clause of the statement must specify the same partition as the UPDATE clause.
- The UPDATE clause must not modify a partitioning column.

The outcome of partitioning expression evaluation errors, such as divide-by-zero errors, depends on the session mode.

In this session mode ...	Expression evaluation errors roll back this work unit ...
ANSI	request.
Teradata	transaction.

## Locks and Concurrency

An UPDATE (Upsert form) operation sets a WRITE lock for the row being updated. For a nonconcurrent load isolated update on load isolated table, the update operation sets an EXCLUSIVE lock.

## UPDATE (Upsert Form) Insert Operations and Row-Partitioned Tables

The rules for the INSERT component of the Upsert form of an UPDATE statement for a target row-partitioned table are as follows:

- The target table can be row partitioned but cannot be column partitioned.
- The outcome of partitioning expression evaluation errors, such as divide-by-zero errors, depends on the session mode.

Session Mode	Work Unit Rolled Back by Expression Evaluation Errors
ANSI	Request that contains the aborted request.
Teradata	Transaction that contains the aborted request.

- To insert a row, the partitioning expression must produce a partition number that results in a value between 1 and 65,535 (after casting to INTEGER, if it does not have an INTEGER data type).
- The INSERT clause must specify the same partitioning column values as the UPDATE clause. This also applies when the UPDATE condition specifies conditions based on Period bound functions.
- A Period column in an upsert request must be a term in an equality condition.
- If you specify a Period column as part of the partitioning expression, you can only specify equality conditions on that column for an upsert request on that table. If you submit an upsert request that

specifies inequality conditions on a Period column that is specified in a partitioning expression for the table, then Teradata Database aborts the request and returns an error.

However, you can specify a Period column that is not defined as part of a partitioning expression for both equality and inequality conditions on that column for an upsert request.

- If a partitioning expression specifies a Period bound function, then an equality condition on the bound function is treated as a partitioning bound matching condition.

If both BEGIN and END bounds are specified in a partitioning expression of the table, then only an equality condition on both bounds is treated as a partitioning bound matching condition. This must result in a single partition.

- The conditions IS UNTIL\_CHANGED and IS UNTIL\_CLOSED are treated as equality conditions only for the END bound function.
- Collation mode has the following implications for upsert operations that insert rows into tables defined with a character partitioning. If the collation for a row-partitioned table is either MULTINATIONAL or CHARSET\_COLL and the definition for the collation has changed since the table was created, Teradata Database aborts any request that attempts to insert or update a row in the table and returns an error to the requestor.

## UPDATE (Upsert Form) Update Operations and RPPI Tables

The rules for the UPDATE component of the Upsert form of an UPDATE statement in a target row-partitioned table are as follows.

The target table must have a primary index and can be row partitioned but cannot be column partitioned.

You cannot submit an UPDATE request to update the partitioning columns of a table with row partitioning such that a partitioning expression does not result in a value between 1 and the number of partitions defined for that level.

You must specify the same partition of the combined partitioning expression for the INSERT component in the upsert form of an UPDATE request that you specify for the UPDATE component.

You must specify the values of the partitioning columns in the WHERE clause of the upsert form of an UPDATE request.

You also cannot modify the values of those partitioning columns; otherwise, the system aborts the request and returns an error message to the requestor.

You cannot update the system-derived columns PARTITION and PARTITION#L1 through PARTITION#L62.

Expression evaluation errors, such as divide by zero, can occur during the evaluation of a partitioning expression. The system response to such an error varies depending on the session mode in effect at the time the error occurs.

In this session mode ...	Expression evaluation errors roll back this work unit ...
ANSI	request that contains the aborted request.

In this session mode ...	Expression evaluation errors roll back this work unit ...
Teradata	transaction that contains the aborted request.

Take care in designing your partitioning expressions to avoid expression errors.

The INSERT clause must specify the same partitioning column values as the UPDATE clause. This also applies when the UPDATE condition specifies conditions based on Period bound functions.

If you specify a Period column as part of the partitioning expression, you can only specify equality conditions on that column for an upsert request on that table. An upsert request cannot specify inequality conditions on a Period column that is specified in a partitioning expression for the table.

However, an upsert request can specify a Period column that is not defined as part of a partitioning expression for equality and inequality conditions on that column.

If you specify a Period column in an upsert request, it must be a term in an equality condition.

If the partitioning expression for a target table specifies a Period bound function, then an equality condition on the bound function is treated as a partitioning bound matching condition.

If BEGIN and END bounds are specified in the partitioning expression of the table, only an equality on both bounds is processed as a partitioning bound matching condition. This must result in a single partition.

The conditions IS UNTIL\_CHANGED and IS UNTIL\_CLOSED are treated as equality conditions only for the END bound function.

## UPDATE (Upsert Form) and Join Indexes

For a row inserted into a base table that causes an insert into a row-partitioned join index or an update of an index row in a row-partitioned join index:

- the partitioning expressions for that join index row cannot evaluate to null, and
- the partitioning expression must be an expression that is CASE\_N or RANGE\_N with a result between 1 and 65535 for the row.

Inserting a row into a base table does not always cause inserts or updates to a join index on that base table. For example, you can specify a WHERE clause in the CREATE JOIN INDEX statement to create a sparse join index for which only those rows that meet the condition of the WHERE clause are inserted into the index, or, for the case of a row in the join index being updated in such a way that it no longer meets the conditions of the WHERE clause after the update, cause that row to be deleted from the index.

The process for this is:

1. The system checks the WHERE clause condition for its truth value after the row insert.
2. The system evaluates the condition, then does one of the following:
3. If the condition evaluates to FALSE, the system deletes the row from the sparse join index.
4. If the condition evaluates to FALSE, the system retains the row in the sparse join index.

You cannot assign either a value or a null to the system-derived columns `PARTITION` or `PARTITION#L1` through `PARTITION#L62` in an insert operation.

Collation mode has the following implications for upsert operations that insert rows into tables defined with a character partitioning. If a noncompressed join index with a character partitioning under either an `MULTINATIONAL` or `CHARSET_COLL` collation sequence is defined on a table and the definition for the collation has changed since the join index was created, Teradata Database aborts any request that attempts to insert or update a row in the table and returns an error to the requestor whether the insert or update would have resulted in rows being modified in the join index or not.

If the partitioning expression for a table or noncompressed join index involves Unicode character expressions or literals, and the system has been backed down to a release that has a different Unicode character set than the one in effect when the table or join index was defined, you cannot insert or update rows in the table with an upsert operation.

Insert or update a row in a base table that causes an insert into a join index with row partitioning such that a partitioning expression for that index row does not result in a value between 1 and the number of partitions defined for that level.

Insert or update a row in a base table that causes an update of an index row in a join index with row partitioning such that a partitioning expression for that index row after the update does not result in a value between 1 and the number of partitions defined for that level.

Updating a base table row does not always cause inserts or updates to a join index on that base table. For example, you can specify a `WHERE` clause in the `CREATE JOIN INDEX` statement to create a sparse join index for which only those rows that meet the condition of the `WHERE` clause are inserted into the index, or, for the case of a row in the join index being updated in such a way that it no longer meets the conditions of the `WHERE` clause after the update, cause that row to be deleted from the index.

The process for this activity is as follows:

1. The system checks the `WHERE` clause condition for its truth value after the update to the row.

Condition	Result
FALSE	System deletes the row from the sparse join index.
TRUE	System retains the row in the sparse join index and proceeds to stage b.

2. The system evaluates the new result of the partitioning expression for the updated row.

Partitioning Expression	Result
<ul style="list-style-type: none"> <li>• Evaluates to null, or</li> <li>• Expression that is not <code>CASE_N</code> or <code>RANGE_N</code></li> </ul>	<p>Not between 1 and 65535 for the row.</p> <p>The system aborts the request. It does not update the base table or the sparse join index, and returns an error.</p>
<ul style="list-style-type: none"> <li>• Evaluates to a value, and</li> <li>• Expression that is not <code>CASE_N</code> or <code>RANGE_N</code></li> </ul>	<p>Between 1 and 65535 for the row.</p> <p>The system stores the row in the appropriate partition, which can be different from the partition in which it was stored, and continues processing requests.</p>

## UPDATE (Upsert Form) and Subqueries

The Upsert form of UPDATE does not support subqueries.

## Queue Tables and UPDATE (Upsert Form)

The best practice is to avoid using the UPDATE statement on a queue table because the operation requires a full table scan to rebuild the internal queue table cache. You should reserve this statement for exception handling.

An UPDATE statement cannot be in a multistatement request that contains a SELECT and CONSUME request for the same queue table.

For more information on queue tables and the queue table cache, see *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

## UPDATE (Upsert Form) As a Triggering Action

Triggers invoke the following behavior when fired as the result of an UPDATE (Upsert Form) request:

Trigger Defined on this Action	Description
UPDATE	Fired in all cases for qualified rows. No insert triggers are performed.
INSERT	Fired only if no row is updated and the INSERT is performed.

If no rows qualify, then no triggers are fired.

Triggers are fired as if the UPDATE and INSERT components in the atomic upsert request are performed as separate statements: unconditionally for UPDATE and conditionally for INSERT. The INSERT is performed only when no rows qualify for the UPDATE.

## UPDATE (Upsert Form) As a Triggered Action

UPDATE (Upsert Form) requests are supported as triggered actions. The rules for their use as triggered actions are the same as for any other SQL statement.

## Rules and Restrictions

The basic upsert constraints described in [Rules for Using Atomic Upsert](#) follow from the conventional definition of upsert functionality.



## Unsupported Syntax or Features

This table lists unsupported syntax or features. Attempt to use them causes an error.

Unsupported Syntax	Explanation
INSERT ... SELECT	None.
Positioned UPDATE	A WHERE clause cannot use an updatable cursor to do a positioned UPDATE.
UPDATE ... FROM	None.
UPDATE ... WHERE subquery	A WHERE clause cannot use a subquery either to specify the primary index or to constrain a non-index column.
UPDATE ... SET ... primary_index_column	You cannot update a primary index column because any change to the rowhash value for a row affects its distribution.
UPDATE ... SET ... partitioning_column	You cannot update a partitioning column because any change to the partitioning for a row affects its location.
UPDATE ... SET identity_column ...	You cannot update GENERATED ALWAYS identity column values.
UPDATE ... SET PARTITION ...	You cannot update system-derived PARTITION values. If PARTITION is a user-named column in a table, and not a GENERATED ALWAYS identity column, then it can be updated.

## Using UPDATE (Upsert Form) Requests with Scalar Subqueries

The rules for using scalar subqueries with UPDATE (Upsert Form) requests are the same as those for simple UPDATE requests, except for the following additional rule:

- Because the UPDATE component of an upsert operation must be a simple update, you cannot specify a correlated scalar subquery in its SET clause or its WHERE clause.

See [Rules for Using Scalar Subqueries in UPDATE Requests](#).

## Using UPDATE (Upsert Form) Requests on RPPI Tables

The following rules apply to using UPDATE (Upsert Form) requests on row-partitioned tables with a primary index.

- When an update to a base table row that causes an insert into a join index with row partitioning, a partitioning expression for that index row must result in a value between 1 and the number of row partitions defined for that level. Otherwise, Teradata Database aborts the request and returns an error.

- When an update to base table row that causes an update of an index row in a join index with row partitioning, a partitioning expression for that index row after the update must result in a value between 1 and the number of row partitions defined for that level. Otherwise, Teradata Database aborts the request and returns an error.

## Using UPDATE (Upsert Form) with the DEFAULT Function

The following rules apply to using the DEFAULT function with the upsert form of UPDATE:

- The DEFAULT function takes a single argument that identifies a relation column by name. The function evaluates to a value equal to the current default value for the column. For cases where the default value of the column is specified as a current built-in system function, the DEFAULT function evaluates to the current value of system variables at the time the request is executed.

The resulting data type of the DEFAULT function is the data type of the constant or built-in function specified as the default unless the default is NULL. If the default is NULL, the resulting data type of the DEFAULT function is the same as the data type of the column or expression for which the default is being requested.

- The DEFAULT function has two forms. It can be specified as DEFAULT or DEFAULT (*column\_name*). When no column name is specified, the system derives the column based on context. If the column context cannot be derived, the request aborts and an error is returned to the requestor.
- All of the rules listed in [Rules for Using the DEFAULT Function With Update](#) also apply for an update in the upsert form of the UPDATE statement.
- All of the rules listed in [Inserting using the DEFAULT Function Option, the DEFAULT VALUES Option, or Without Specifying a Value](#) also apply for an insert in the upsert form of the UPDATE statement.

See *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145 for more information about the DEFAULT function.

## UDTs and UPDATE (Upsert Form)

UDT expressions can be used in an upsert form of UPDATE statement in any way that is consistent with the general support of UDTs by INSERT and UPDATE and with existing semantics for UPSERT. See [Inserting into UDT Columns](#), [Updating Distinct UDT Columns](#), and [Updating Structured UDT Columns](#).

The UPDATE statement used in the upsert operation can also use a mutator SET clause if it updates a structured UDT column. See [Updating Structured UDTs Using a Mutator SET Clause](#).

## UPDATE (Upsert Form) Support for Load Isolated Tables

A nonconcurrent load isolated update operation on a load isolated table updates the matched rows in-place.

A concurrent load isolated update operation on a load isolated table logically deletes the matched rows and inserts the rows with the modified values. The rows are marked as deleted and the space is not

reclaimed until you issue an ALTER TABLE statement with the RELEASE DELETED ROWS option. See *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## Examples

### UPDATE Upsert Examples

This section provides examples that show how the atomic upsert form of UPDATE works, including error cases. All examples use the same table called *sales*, which is defined as:

```
CREATE TABLE sales, FALLBACK
  item_nbr   INTEGER NOT NULL,
  sale_date  DATE FORMAT 'MM/DD/YYYY' NOT NULL,
  item_count INTEGER)
PRIMARY INDEX (item_nbr);
```

Assume that the table has been populated with the following data.

```
INSERT INTO sales (10, '05/30/2000', 1);
```

### Example: Upsert Update

This example shows a valid upsert UPDATE request.

```
UPDATE sales
SET itemcount = item_count + 1
WHERE (item_nbr = 10
AND    sale_date = '05/30/2000')
ELSE INSERT
INTO sales (10, '05/30/2000', 1);
```

After all of the rules have been validated, the row with *item\_nbr* = 10 and *sale\_date* = '05/30/2000' gets updated.

A message indicates the successful update of one row.

### Example: Upsert Insert

This example shows a valid upsert INSERT request.

```
UPDATE sales
SET item_count = item_count + 1
WHERE (item_nbr = 20
```

```

AND    sale_date = '05/30/2000')
ELSE INSERT INTO sales (20, '05/30/2000', 1);

```

After all of the rules have been validated and no row was found that satisfies the compound predicate *item* = 20 and *sale\_date* = '05/30/2000' for the update, a new row is inserted with *item\_nbr* = 20.

A message indicates the successful insert of one row.

## Example: Upsert Specifying Different Tables

This example shows an upsert UPDATE request that does not specify the same table name for both the UPDATE part and the INSERT part of the request.

```

UPDATE sales
SET item_count = item_count + 1
WHERE (item_nbr = 10
AND    sale_date = '05/30/2000')
ELSE INSERT INTO new_sales (10, '05/30/2000', 1);

```

One of the rules of the upsert form of UPDATE is that only one table is processed for the request. Because the tables *sales* and *new\_sales* are not the same for this example the system returns an error to the user indicating that the name of the table must be the same for both the UPDATE and the INSERT.

## Example: Upsert on Primary Index

This example shows an upsert UPDATE request that does not specify the same primary index value for the UPDATE and INSERT parts of the request.

```

UPDATE sales
SET item_count = item_count + 1
WHERE (item_nbr = 10
AND    sale_date = '05/30/2000')
ELSE INSERT INTO sales (20, '05/30/2000', 1);

```

The primary index value specified for the UPDATE and the INSERT must be the same. Otherwise, the operation is looking at two different rows: one for UPDATE and the other for the INSERT. This is not the purpose of the upsert form of UPDATE.

Because the specified primary index values of 10 and 20 are not the same, this case returns an error to the user, indicating that the primary index value must be the same for both the UPDATE and the INSERT.

## Example: Upsert Without Specifying Primary Index

This example shows an upsert UPDATE request that does not specify the primary index in its WHERE clause.

```
UPDATE sales
SET item_count = item_count + 1
WHERE sale_date = '05/30/2000'
ELSE INSERT INTO sales (10, '05/30/2000', 1);
```

When the primary index is not specified in the UPDATE portion of an upsert request, the operation could have to perform an all-row scan to find rows to update. This is not the purpose of upsert form of UPDATE. This case returns an error.

### Example: Upsert Without ELSE Clause

This example shows an upsert UPDATE request that fails to specify the ELSE keyword.

```
UPDATE sales
SET item_count = item_count + 1
WHERE (item_nbr = 10
AND sale_date = '05/30/2000')
INSERT INTO sales (10, '05/30/2000', 1);
```

This case returns a syntax error to the user.

### Example: Upsert Update Using the DEFAULT Function

When the DEFAULT function is used for either the UPDATE operation or for the INSERT operation within the upsert, it evaluates to the default value of the referenced column. This is a Teradata extension.

Assume the following table definition for the examples:

```
CREATE TABLE table_19 (
  col_1 INTEGER,
  col_2 INTEGER DEFAULT 10,
  col_3 INTEGER DEFAULT 20,
  col_4 CHARACTER(60));

UPDATE table19
SET col_2 = DEFAULT
WHERE col1 = 10
ELSE INSERT table_19 (10, DEFAULT, DEFAULT, 'aaa');
```

This request updates *col\_2* to the DEFAULT value of *col\_2*, which is 10, depending on whether the WHERE condition evaluates to true or not.

If the row *does* exist, the updated row becomes the following: (10, 10, existing value, existing value).

If the row does *not* exist, the system inserts a new row with the a *col\_2* value of 10 (the default value of *col\_2*) and a *col\_3* value of 20 (the default value of *col\_3*). The newly inserted row is as follows: (10, 10, 20, 'aaa').

The following example is a correct use of the DEFAULT function within an UPDATE upsert request:

```
UPDATE table_19
  SET col_2 = DEFAULT(col3)
  WHERE col_1 = 10
  ELSE INSERT table_19 (10, DEFAULT, DEFAULT(col_2), 'aaa');
```

When the value of *col\_1* is 10, this upsert updates *col\_2* to the DEFAULT value of *col\_3*, which is 20, because the column name passed as the argument of the DEFAULT function is *col\_3*.

If the row does exist, the updated row becomes the following: (10, 20, existing value, existing value).

If the row does not exist, the system inserts a new row with a *col\_2* value of 10 (default value of *col\_2*) and a *col\_3* value of 10 (the default value of *col\_2*). The newly inserted row is as follows: (10, 10, 10, 'aaa').

## Example: Upsert Update Using a Period Bound Function

Suppose you define the following row-partitioned table using the END Period bound function.

```
CREATE SET TABLE testing.t33 (
  a INTEGER,
  b PERIOD(DATE),
  c INTEGER)
PRIMARY INDEX (a)
PARTITION BY CAST((END(b)) AS INTEGER);
```

This UPDATE upsert request inserts a new row into *t33*.

```
UPDATE t33
  SET c = 1
  WHERE a = 20
  AND   END(b) = DATE '1901-02-25'
  ELSE INSERT INTO t33 (20, PERIOD(DATE '1901-02-24',
    DATE '1901-02-25'), 1);
```

## USING Request Modifier

### Purpose

Defines one or more variable parameter names used to import data to Teradata Database from a client system or to export data from Teradata Database to a client system. Also specifies how imported or exported LOB data is to be handled.

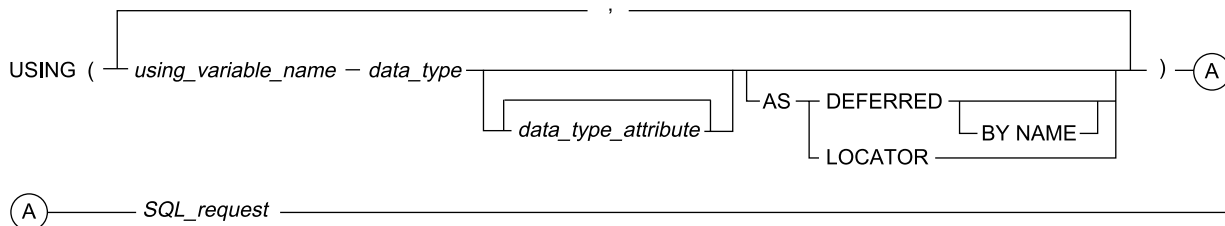
See also:

- *Basic Teradata® Query Reference*, B035-2414
- *Teradata® SQL Assistant for Microsoft Windows User Guide*, B035-2430

## Required Privileges

None.

## Syntax



## Syntax Elements

### *using\_variable\_name*

Name of a USING variable that is to be referenced as a parameter in the SQL request associated with the modifier.

Each name specified must be unique.

During processing, each *using\_variable\_name* in the SQL request is replaced by a constant value read from the client application.

You can also use dot notation to access DATASET and JSON data. See "Accessing DATASET Data Using Dot Notation" in *Teradata Vantage™ DATASET Data Type*, B035-1198 and "JSON Entity Reference (Dot Notation Syntax)" in *Teradata Vantage™ JSON Data Type*, B035-1150.

### *data\_type*

Data type of the constant value substituted for *using\_variable\_name*.

UDT types are not supported directly. See [USING Support for UDTs](#) for details and information about conversion workarounds.

This can be used to set a default value.

See *Teradata Vantage™ Data Types and Literals*, B035-1143 for a list of data types.

### *data\_type\_attribute*

One of the data type attributes listed below.

The only data type attributes that affect the handling of values imported by USING are the following:

- CASESPECIFIC
- NOT CASESPECIFIC

- UPPERCASE

The server character set attribute (for example CHARACTER SET LATIN) cannot be part of *data\_type\_attribute*.

Type attributes are Teradata extensions to the ANSI SQL:2011 standard.

### AS DEFERRED

LOB data is sent to Teradata Database from the client application when requested, and separately from other row data.

This option is only valid for LOB data.

See [USING Support for Large Objects](#) for more information.

### BY NAME

Interpret the data in the USING row as a client file name. When the system detects the BY NAME option following AS DEFERRED, the following events occur in the order indicated:

Teradata Database sends the file name back to the client.

The client opens the file.

The client sends the data in the file to Teradata Database.

### AS LOCATOR

The client application passes a locator for the LOB, generated from a previous SELECT request for the session, to Teradata Database.

This option is only valid for LOB data.

See [USING Support for Large Objects](#) for more information.

### SQL\_request

The SQL request with which the USING request modifier is associated.

This can be a multistatement request, any non-DDL single-statement request, or an explicit transaction.

You can pass the value for *n* in a TOP *n* operator in an SQL request by specifying a USING-defined parameter. See [Example: USING Request Modifier with TOP Operator](#).

All USING variable names in the SQL request must be preceded by a COLON character.

## ANSI Compliance

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

## Usage Notes

### USING Is Not Supported for Embedded SQL

Embedded SQL does not support the USING request modifier.



Declare and use input host variables in your embedded SQL applications where you would use the USING request modifier interactively.

## Actions Performed by USING

USING imports data rows from a client system as follows:

1. Each value is retrieved from a client system-resident data source, such as a disk file, that you must specify using a client utility such as BTEQ.

For example, if you submit a USING-modified request through BTEQ, you must precede the request with something like the following BTEQ command:

```
.IMPORT DATA FILE=c:\temp\t1
```

See *Basic Teradata® Query Reference*, B035-2414 for documentation on the .IMPORT command.

2. The values are passed to Teradata Database as a data parcel along with the text of the request. Iterated requests can pack multiple data parcels with the text of a request.
3. A value is substituted for each parameter name when the modified SQL request is processed.

You can import data from a client system by preceding any of the following SQL statements with a USING request modifier:

- DELETE
- INSERT
- MERGE
- SELECT
- UPDATE

For exporting Teradata Database data to a client, the process is essentially the reverse of that used for data import.

You can export data to a client system using a client utility such as BTEQ and specifying the BTEQ .EXPORT command, then preceding a SELECT request to retrieve the data to be exported from Teradata Database with an appropriately specified USING request modifier.

See *Basic Teradata® Query Reference*, B035-2414 for documentation on the .EXPORT command.

## USING Variables

The maximum number of USING variables you can specify in a USING request modifier is 2,536.

Each USING variable name must be unique in a USING request modifier.

The USING variable construct (*:using\_variable\_name*) is valid anywhere a character, numeric, or byte constant is valid. The constant values in the client system data record must match, item for item, the

USING description of those values. Also, the constants in the data record must appear in the order in which they are defined in the USING clause.

You cannot use a USING variable to specify a column position in a GROUP BY or ORDER BY clause.

One USING clause can only modify one request. If several requests require USING variables, each request must be associated with its own USING row identifier.

## USING Support for Scalar UDFs

You can pass USING variables in the arguments of a scalar UDF during invocation (see [Example: USING and SQL UDF Invocation](#)) when the data types of the USING variables match those of the UDF parameters, whether natively or after being explicitly cast. If the data types do not match, Teradata Database aborts the request and returns an error.

You can avoid this problem by casting the data types of the USING variables to the data types of the UDF parameters.

## Valid Request Types

A USING modifier can be associated with one of the following types of requests:

- Any single-statement request except a DDL or DCL statement. An iterated request is a special case of this query type and must always have an associated USING modifier.
- A multistatement request.
- An explicit transaction. If the modifier is associated with an explicit transaction, the USING keyword must appear as the first keyword in that transaction.

If the first statement or request references a USING variable, the USING clause should immediately precede the BEGIN TRANSACTION statement. See [Example: USING Request Modifier Reads Character Strings](#).

## USING and the EXPLAIN Request Modifier

If you specify both a USING request modifier and an EXPLAIN request modifier for the same request, the EXPLAIN request modifier must precede the USING request modifier. See [EXPLAIN Request Modifier](#).

## USING Support for UDTs

USING does not support UDTs explicitly. Client applications must transfer UDT data to the Teradata platform in its external type form, which is always a predefined data type. You can then use either an implicit or explicit cast to import UDTs by means of the USING request modifier.

Implicit casts are invoked during assignment operations, including: INSERT, UPDATE, and parameter passing operations.

When both of the following conditions are true, the system automatically converts the client value predefined data type to the appropriate Teradata platform value UDT by means of an implicit casting operation:

- A host variable is involved in an assignment operation and is assigned to a UDT target
- An implicit cast is defined. The implicit cast can be either of the following:
  - A system-generated implicit cast.
  - An explicitly created implicit cast.

See “CREATE CAST” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144 for details.

If no applicable implicit cast has been defined, you must perform explicit conversions using constructor methods, casts, UDFs, and so on, as indicated by the following examples:

- The following example shows an implicit cast conversion from a client predefined data type to a Teradata platform distinct UDT:

```
USING (europe_sales DECIMAL(9,2))
INSERT INTO tab1 VALUES (:europe_sales, 1);
```

- The following example shows an explicit conversion and construction operation using a constructor method:

```
USING (street VARCHAR(20), zip CHARACTER(5))
INSERT INTO tab2 VALUES (NEW address(:street, :zip), 2 );
```

- The following example shows conversion by means of method invocation:

```
USING (price DECIMAL(6,2))
UPDATE tab1
SET column1 = CAST(:price AS euro).roundup(0);
```

Best practices for UDT design recommend that you duplicate all transform group functionality to enable optimum support for the USING request modifier. You can do this by coding equivalent predefined external data type-to-UDT and UDT-to-predefined external data type casts. The simplest way to accomplish this is to reference the same routines in the equivalent CREATE CAST and CREATE TRANSFORM statements. For details about how to use these statements, see “CREATE CAST” and “CREATE TRANSFORM” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

If you follow these guidelines, then iterated array processing is also supported. UDTs cannot be referenced for array processing directly in a USING request modifier, but can be populated by using the implicit cast (from predefined data type-to-UDT) mechanism.

Distinct and structured UDTs differ in these requirements:

- For a distinct UDT, if you plan to use only its system-generated functionality, no work is required because the transform and implicit casting functionality has already been defined.

- For a structured UDT, you must define the transform and implicit casting functionality explicitly with CREATE CAST and CREATE TRANSFORM statements, as indicated by the following examples for a UDT named address:

- The following request creates the transform group:

```
CREATE TRANSFORM FOR address client_io (
  TO SQL WITH SPECIFIC FUNCTION SYSUDTLIB.stringToAddress,
  FROM SQL WITH SPECIFIC METHOD toString);
```

- The following request creates an implicit cast from VARCHAR(100) to address that duplicates the tosql functionality of the client\_io transform group:

```
CREATE CAST ( VARCHAR(100) AS address )
  WITH SPECIFIC FUNCTION SYSUDTLIB.stringToAddress
  AS ASSIGNMENT;
```

- The following request creates an implicit cast from address to VARCHAR(100) that duplicates the fromsql functionality of the client\_io transform group:

```
CREATE CAST (address AS VARCHAR(100))
  WITH SPECIFIC METHOD ToString AS ASSIGNMENT ;
```

## USING Support for Large Objects

The USING request modifier supports passing LOBs to the Teradata platform. The documentation for each Teradata Database-supported API specifies how this functionality is presented to the application.

You can specify three different modes for handling large objects by the USING request modifier:

- Inline
- Deferred
- Locator

Deferred and locator modes defer the transfer of data between client and server, transmitting non-LOB data separately from LOB data. The appropriate deferred mode depends on the application. Neither deferred or locator mode passes the entire LOB field with the non-LOB data from a row.

### Inline Mode

With inline mode, an entire LOB field is passed along with the other fields in the row. Inline mode is the default.

### Deferred Mode

Portions of the LOB are passed sequentially by the client application to Teradata Database until the entire LOB has been transmitted.

Deferred mode means that the system passes a 4-byte integer token identifying the LOB with the non-LOB data in the initial request. Then while in MultiPart Request mode, Teradata Database elicits the LOB data from the client with an ElicitData parcel, providing the token to identify the particular LOB data it wants. A maximum-sized LOB can be passed in this way without it being necessary to know the LOB data size in advance.

However, simple deferred mode does not work well with client utilities that might not know the location of the LOB data on the client. Because of this, it is best to use the BY NAME phrase whenever you specify the AS DEFERRED option.

When you specify the AS DEFERRED BY NAME option, the LOB data is represented in the data sent with the request as a VARCHAR value limited to 1,024 bytes in length. Teradata Database processes AS DEFERRED BY NAME in the same manner as if it were a simple AS DEFERRED, with the exception that when the AMP wants to elicit the LOB data from the client, it uses a ElicitDataByName parcel, which identifies the LOB by the name passed by the client in the initial request, rather than an ElicitData parcel, which identifies the LOB by the LOB token that was originally passed by the client.

## Locator Mode

A locator passes a LOB value reference from a client to the server application or user-defined function without passing the LOB value itself. Its purpose is to minimize message traffic, enhancing system performance in the process. Queries that request LOB data are serviced with intermediate results that contain locators to LOB data, not the LOB data itself.

When you use a locator as a parameter in a subsequent request, the result is exactly as if the associated LOB value had been used directly. Locators can be used in almost any context where a LOB value is valid.

Locators are instantiated in the result set of an SQL query and bound to an application program or UDF variable by an application-specific method.

Locators exist for a limited time within the session in which they are created. Their precise life span depends on the specific application.

## LOB Transfer Mode Comparison

The following table lists what the system passes in the various LOB modes:

Mode	Description
Inline	Entire LOB along with all non-LOB data. This is limited by the maximum request size of 1 MB, but even if that limit is not exceeded, the size of the transmitted LOB cannot exceed 64 Kbytes.
Deferred	A 64 Kbyte portion of the LOB. 64 Kbyte portions of the LOB are passed sequentially by the client application to Teradata Database until the entire LOB has been transmitted.

Mode	Description
	Consult the API document for your application to determine its parcel size limit for a deferred LOB. Some APIs support parcels as large as 1 MB.
Locator	Value reference to the LOB on the Teradata platform. Locators are generated by a previous SELECT request within the current session and passed to the client application, which later passes it back to the Teradata platform in a CLlv2 response.

The following table summarizes the uses for each mode:

Mode	Description
Inline	<p>Also known as non-deferred mode.</p> <p>Generally useful for small LOBs only because the entire LOB is transferred along with non-LOB data when transmitted to or from Teradata Database.</p> <p>Inline mode transfers LOBs in the same way it transfers other field values. This mode does not require a special clause in the USING text.</p> <p>You can use inline mode to transfer multiple rows.</p> <p>The maximum total size of the request, when transmitted from a client application to Teradata Database, is 1 MB, which is the system maximum request size. However, the size of the transmitted LOB cannot exceed 64 Kbytes.</p> <p>You cannot transmit more than 64 Kbytes of data from a client application to the Teradata platform because that is the maximum row length for Teradata Database.</p> <p>There is no restriction on the size of LOB data that can be transmitted to Teradata Database from a client application.</p>
Deferred	<p>Specify this mode with a required AS DEFERRED clause.</p> <p>Transfers LOBs sequentially from client-to-server in 64 Kbyte fragments. After Teradata Database receives each LOB fragment, it sends an acknowledgement to the application, which then either sends the next LOB fragment or, if all the data has been sent, terminates the request.</p> <p>You can only transmit single rows in deferred mode.</p> <p>There is no limit on the size of the request.</p> <p>In this mode, the Teradata platform requests the application to transmit the LOB after validating, but before completing, the SQL request. The Teradata platform returns a failure if an SQL request that contains a deferred LOB is performed on every AMP in the system (usually, but not always, excluding single AMP systems). Note that this refers only to all-AMP requests. The Teradata platform also prevents overlapping deferred LOB requests within one query.</p> <p>If there are multiple LOBs identified as being transferred in deferred mode, the Teradata platform might not request them in the order in which they are specified in the USING clause.</p> <p>You can append a BY NAME option to an AS DEFERRED clause, which provides a smoother interface for handling deferred data.</p>
Locator	<p>Specify this mode with a required AS LOCATOR clause.</p> <p>Transfers non-LOB data plus a locator to the Teradata platform-resident LOB data from client-to-server.</p> <p>You can transmit multiple rows in locator mode.</p> <p>Once instantiated, a locator can be used by other requests made within the same session.</p>

## Client Application Restrictions on USING With Large Objects

The following client applications support USING with large objects with no restrictions:

- CLIV2
- Archive/Recovery

The following client applications support USING with large objects with the noted restrictions:

Application	Restrictions
BTEQ	Inline mode only.
Teradata Parallel Transport	Deferred and Locator modes for these operators only: <ul style="list-style-type: none"> <li>• DataConnector</li> <li>• SQL Inserter</li> <li>• SQL Selector</li> </ul>

- The following Teradata Tools and Utilities applications do not support USING with large objects:
  - FastLoad
  - FastExport
  - MultiLoad
  - Embedded SQL
- Teradata Database stored procedures do not support USING with large objects.

Consult the appropriate documentation for additional restrictions on LOB use with Teradata features.

## USING and DateTime System Functions

In non-ANSI Teradata Database applications, when you access the system functions DATE or TIME, the values are obtained directly from the operating system and placed in a USING row for access by the query being performed. In this situation, the value for DATE is stored with type DATE and the value for TIME is stored with type REAL. These values are stored in known fixed fields in the USING row.

The ANSI system functions CURRENT\_DATE, CURRENT\_TIME, and CURRENT\_TIMESTAMP behave differently:

- CURRENT\_DATE is stored in the field previously used exclusively for DATE.
- CURRENT\_TIME is not stored in the field previously used for TIME because both its storage format and its content are different and are not interchangeable with those of TIME. Another system-value field is added to the USING row. Notice that CURRENT\_TIME includes TIME ZONE data in addition to the more simple TIME data, so the differences between the two extend beyond storage format.
- CURRENT\_TIMESTAMP is also stored as an additional separate field in the USING row.

You can use the legacy Teradata Database DATE and TIME functionality and the ANSI DateTime functionality with the USING request modifier.

## ANSI DateTime Considerations

Other than DATE values in INTEGERDATE mode, external values for DateTime and Interval data are expressed as fixed length CharFix character strings (in logical characters) in the designated client character set for the session.

The type provided in a USING request modifier for any client data to be used as a DateTime value can be defined either as the appropriate DateTime type or as CHARACTER(*n*), where *n* is a character string the correct length for the external form of a DateTime or Interval data type, and the character string is to be used internally to represent a DateTime or Interval value.

When a client creates USING data without being aware of the ANSI DateTime data types, those fields are typed as CHARACTER(*n*). Then when those USING values appear in the assignment lists for INSERTs or UPDATEs, the field names from the USING phrase can be used directly.

An example follows:

```
USING (TimeVal CHARACTER(11),
      NumVal INTEGER,
      TextVal (CHARACTER(5))
INSERT INTO TABLE_1 (:TimeVal, :NumVal, :TextVal);
```

When you import ANSI DateTime values with a USING request modifier and the values are to be used for actions other than an INSERT or UPDATE, you must explicitly CAST them from the external character format to the proper ANSI DateTime type.

An example follows:

```
USING (TimeVal CHARACTER(11),
      NumVal INTEGER)
UPDATE TABLE_1
SET TimeField=:TimeVal, NumField=:NumVal
WHERE CAST(:TimeVal AS TIME(2)) > TimeField;
```

While you can use TimeVal CHARACTER(11) directly for assignment in this USING request modifier, you must CAST the column data definition explicitly as TIME(2) in order to compare the field value TimeField in the table because TimeField is an ANSI TIME defined as TIME(2).

You can use both DateTime and Interval declarations to allow a USING request modifier to directly indicate that an external character string value is to be treated as a DateTime or Interval value. To import such values, you import their character strings directly into the USING request modifier.

If you move values into a USING request modifier and the character string data cannot be converted into valid internal DateTime or Interval values as indicated by their type definitions, then the system returns an error to the application.



## Example of ANSI DateTime and Interval With USING

The following USING request modifier expects to see an 11 character field containing a string such as '21:12:57.24'.

```
USING ( TimeVal TIME(2), ... )
```

You could import the same value using the following USING request modifier; however, it is unclear that the data is a time value with two decimal places:

```
USING ( TimeVal CHARACTER(11), ... )
```

## ANSI DateTime and Parameterized Requests

A CLIV2 StatementInfo parcel is used in both “Prepare then Execute” and “Parameterized Query” CLI modes to describe fields for export and import and to build a USING request modifier without a USING request modifier in the request text.

In this mode of operation, the system observes the following rules:

- DateTime and Interval exported values as seen by the client as CharFix data with the exception of DATE values when the session is in INTEGERDATE mode.
- Imported values that are to be DateTime or Interval values are seen by the system as CharFix values with the exception of DATE values when the session is in INTEGERDATE mode.

When used in an INSERT or UPDATE request, CharFix values can be implicitly cast in the appropriate DateTime or Interval value. In this case, the system ensures that the value being assigned is CharFix and that it has the right length to represent the DateTime or Interval value to which it is being assigned.

The system casts the value and accepts the result if the contents are a valid representation of a value of the indicated data type. In other cases, you must explicitly cast the USING parameter as the appropriate DateTime or Interval data type.

## Array Considerations for Specifying TOP $n$ as a USING Parameter

You cannot specify the  $n$  value of a TOP  $n$  specification as a USING parameter for arrays. Otherwise, an error is returned to the requestor. For an example, see [Example: Non-Support for Iterated Requests With TOP  \$n\$](#) .

## Character String Definitions in a USING Request Modifier

Default case specificity for character string comparisons depends on the mode defined for the current session.

IF the session is in this mode ...	THEN the values default to this declaration for string comparisons ...
ANSI	CASESPECIFIC
Teradata	NOT CASESPECIFIC

You can add the explicit attribute NOT CASESPECIFIC to the definition of a CHARACTER or VARCHAR field in the USING phrase to override the default.

The purpose of the NOT CASESPECIFIC attribute is to ease the transition to an ANSI session mode of operation. You should instead use the ANSI SQL:2011-compliant UPPER function to perform case blind comparisons.

### ***data\_type* Considerations for a Japanese Character Site**

Be very careful with Japanese character data regarding the information that will be sent in the data parcel in client form-of-use. Consider the following example for illustrating the issues involved.

```

USING (emp_name  CHARACTER(40) UPPERCASE,
      emp_number INTEGER)
INSERT INTO employee (:emp_name, :emp_number);

```

This request specifies that the data parcel contains 40 bytes of CHARACTER data for *emp\_name* and four bytes of INTEGER data for *emp\_number*.

Because the *data\_type* describes the layout of the data parcel in terms of client form-of-use, this means that CHARACTER(40) indicates 40 *bytes* of CHARACTER information rather than 40 *characters*.

For single-byte character external data sets such as ASCII and EBCDIC, bytes and characters represent the same byte count.

For character sets containing mixed single-byte and multibyte characters (such as KanjiEUC), or only 2-byte characters, however, the numbers of bytes and numbers of characters are *not* identical.

Notice that the GRAPHIC and VARGRAPHIC data types have character counts that are always half the number of bytes.

For Kanji character sets containing only GRAPHIC multibyte characters, such as KanjiEBCDIC, character and byte units are identical.

You can specify GRAPHIC USING data by specifying a [VAR]CHAR(*n*) CHARACTER SET GRAPHIC attribute for the *data\_type*. This is identical to specifying the *data\_type* as [VAR]GRAPHIC(*n*).

There is no equivalent extension to CHARACTER SET syntax to account for CHARACTER data, because the data is in client rather than internal form-of-use.

## Non-GRAPHIC Character Data Representation

Non-GRAPHIC CHARACTER(*n*) and VARCHAR(*n*) data types in a USING request modifier are defined in terms of bytes, where the maximum value of *n* is 64K.

In all other SQL declarations, such as a column definition within a CREATE TABLE statement, non-GRAPHIC CHARACTER(*n*) and VARCHAR(*n*) data types are defined as follows:

This server character set...	Is defined in terms of ...	And the maximum value of <i>n</i> is ...
Latin	characters	64K
Unicode		32K
Kanji1	bytes	64K
Kanjisjis		32K

### 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.

For details on the process of importing character data with a USING request modifier, see [Character Data Import Process](#).

## Character String Assignment and GRAPHIC Columns

The following table describes the behavior of character strings assigned to GRAPHIC and non-GRAPHIC columns.

You cannot import non-GRAPHIC data into character columns typed as GRAPHIC, nor can you import GRAPHIC data into character columns that are not typed as GRAPHIC.

IF USING describes a character string for assignment to a column of this type ...	THEN it must describe the USING string as this type ...
GRAPHIC	<ul style="list-style-type: none"> <li>• GRAPHIC(<i>n</i>)</li> <li>• VARGRAPHIC(<i>n</i>)</li> <li>• CHARACTER(<i>n</i>) CHARACTER SET GRAPHIC</li> <li>• VARCHAR(<i>n</i>) CHARACTER SET GRAPHIC</li> </ul>
LATIN UNICODE	<ul style="list-style-type: none"> <li>• CHARACTER(<i>n</i>)</li> <li>• VARCHAR(<i>n</i>)</li> </ul>

IF USING describes a character string for assignment to a column of this type ...	THEN it must describe the USING string as this type ...
KANJISJIS KANJI1	

## CHARACTER and GRAPHIC Server Character Set Limitations for INSERT Operations

This table describes the data limits on INSERT operations for various CHARACTER and GRAPHIC server character sets.

Client Character Set Type	USING Data Type	Maximum Size for USING Data	Server Character Set	Maximum Column Size for INSERT (Characters)
Single byte	CHARACTER	64,000 bytes	LATIN	64,000
			KANJI1	
			UNICODE If the USING data row contains more than 32K logical characters (or more than 64K <i>bytes</i> when mixed single-byte-multibyte strings are involved), the characters that exceed that limit are truncated. Note that this includes Unicode data in both UTF-8 and UTF-16 sessions.	32,000
			KANJISJIS If the USING data row contains more than 32K logical characters (or more than 64K <i>bytes</i> when mixed single-byte-multibyte strings are involved), the characters that exceed that limit are truncated.	
Multibyte	CHARACTER	64,000 bytes	LATIN If the USING data row contains more than 32K logical characters (or more than 64K <i>bytes</i> when mixed single-byte-multibyte strings are involved), the system rejects the row. Although a Latin field can be defined as 64K, you cannot insert more than 32K multibyte characters into it because Teradata	32,000

Client Character Set Type	USING Data Type	Maximum Size for USING Data	Server Character Set	Maximum Column Size for INSERT (Characters)
			Database uses Unicode internally, and Unicode has a 32K limit. The load utilities are similarly limited.	
			KANJI1	64,000
			UNICODE If the USING data row contains more than 32K logical characters (or more than 64K <i>bytes</i> when mixed single-byte-multibyte strings are involved), the characters that exceed that limit are truncated. Note that this includes Unicode data in both UTF-8 and UTF-16 sessions.	32,000
Multibyte	CHARACTER	64,000 bytes	KANJISJIS If the USING data row contains more than 32K logical characters (or more than 64K <i>bytes</i> when mixed single-byte-multibyte strings are involved), the characters that exceed that limit are truncated.	32,000
	GRAPHIC	32,000 characters	GRAPHIC	32,000
			LATIN If the USING data row contains more than 32K logical characters (or more than 64K <i>bytes</i> when mixed single-byte-multibyte strings are involved), the system rejects the row. Although a Latin field can be defined as 64K, you cannot insert more than 32K multibyte characters into it because Teradata Database uses Unicode internally, and Unicode has a 32K limit. The load utilities are similarly limited.	
			KANJI1	
			UNICODE If the USING data row contains more than 32K logical characters (or more than 64K <i>bytes</i> when mixed single-byte-multibyte strings are involved), the characters that exceed that limit are truncated.	

Client Character Set Type	USING Data Type	Maximum Size for USING Data	Server Character Set	Maximum Column Size for INSERT (Characters)
			Note that this includes Unicode data in both UTF-8 and UTF-16 sessions.	
			KANJISJIS If the USING data row contains more than 32K logical characters (or more than 64K <i>bytes</i> when mixed single-byte-multibyte strings are involved), the characters that exceed that limit are truncated.	

## Character Data Import Process

Consider the following table definition:

```
CREATE TABLE table_1 (
  cunicode CHARACTER(10) CHARACTER SET UNICODE,
  clatin CHARACTER(10) CHARACTER SET LATIN,
  csjis CHARACTER(10) CHARACTER SET KANJISJIS,
  cgraphic CHARACTER(10) CHARACTER SET GRAPHIC,
  cgraphic2 CHARACTER(10) CHARACTER SET GRAPHIC);
```

Suppose the session character set is KanjiShift-JIS and you submit the following request:

```
USING
  cunicode (CHARACTER(10)),
  clatin (CHARACTER(10)),
  csjis (CHARACTER(10)),
  cgraphic (GRAPHIC(10)),
  cgraphic2 (CHARACTER(10) CHARACTER SET GRAPHIC))
INSERT INTO table_1(:cunicode, :clatin, :csjis, :cgraphic,
                   :cgraphic2);
```

The USING request modifier indicates that the data parcel contains the following:

- 10 KanjiShift-JIS *bytes* for the Unicode column
- 10 bytes for the Latin column
- 10 bytes for the KanjiSJIS column
- 20 bytes for the first GRAPHIC column
- 20 bytes for the second GRAPHIC column

Individual fields in the data parcel are converted from the client form-of-use to the server form-of-use. After conversion, the first three fields are treated as Unicode literals and the last two fields are treated as GRAPHIC literals (see *Teradata Vantage™ Data Types and Literals*, B035-1143).

The column data is then converted to the target fields using implicit translation according to the rules listed in *Teradata Vantage™ Data Types and Literals*, B035-1143.

The conversion process is described by the following table.

Stage	Process			
	These bytes are converted to server form-of-use ...	Treated as this kind of literal ...	Converted to this character data type ...	And stored in this column ...
1	first 10 (1 - 10)	UNICODE	none	cunicode
2	next 10 (11 - 20)	UNICODE	LATIN	clatin
3	next 10 (21-30)	UNICODE	KANJISJIS	csjis
4	next 20 (31 - 50)	GRAPHIC	GRAPHIC	cgraphic
5	last 20 (51 - 70)	GRAPHIC	GRAPHIC	cgraphic2

Note that the meaning of the USING clause is independent of the session character set.

## UPPERCASE Option and Character Parameter Definition in USING

When you specify UPPERCASE for a character parameter in a USING request modifier, it applies only to the single-byte characters of any mixed single-byte-multibyte character set.

To ensure uniform formatting as uppercase, do one of the following:

- Define the column with the UPPERCASE attribute in CREATE TABLE or ALTER TABLE (see “ALTER TABLE” and “CREATE TABLE” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ Data Types and Literals*, B035-1143 for details).
- Use the UPPER function to convert the lowercase characters to uppercase (see *Teradata Vantage™ SQL Functions, Expressions, and Predicates*, B035-1145).

See [Character Data Import Process](#).

## Examples

### Example: USING Request Modifier

In this example, the USING request modifier establishes three variable parameters whose constant values are used both for data input and as WHERE clause predicates in a multistatement request:

```
.SET RECORDMODE ON
.IMPORT DATA FILE = r13sales.dat;

USING (var_1 CHARACTER, var_2 CHARACTER, var_3 CHARACTER)
INSERT INTO testtabu (c1) VALUES (:var_1)
;INSERT INTO testtabu (c1) VALUES (:var_2)
;INSERT INTO testtabu (c1) VALUES (:var_3)
;UPDATE testtabu
    SET c2 = c1 + 1
    WHERE c1 = :var_1
;UPDATE testtabu
    SET c2 = c1 + 1
    WHERE c1 = :var_2
;UPDATE testtabu
    SET c2 = c1 + 1
    WHERE c1 = :var_3;
```

### Example: USING Request Modifier with Variables

In this example the USING request modifier defines the variables *:emp\_name* and *:emp\_number* as, a CHARACTER constant and an INTEGER numeric constant, respectively. The USING variables are replaced by values from a client system data record when the system processes the accompanying INSERT request.

```
.SET RECORDMODE ON
.IMPORT DATA FILE = r13sales.dat;

USING (emp_name CHARACTER(40),
      emp_number INTEGER)
INSERT INTO employee (name, empno)
VALUES (:emp_name, :emp_number);
```

The INSERT request (in Record Mode on an IBM mainframe) is transmitted to Teradata Database with an appended 44-byte data record consisting of a 40-byte EBCDIC character string followed by a 32-bit integer.

### Example: USING Request Modifier Reads Character Strings

In this example, the USING request modifier defines a variable parameter for use with an explicit transaction that reads character strings from a disk file and inserts them in signed zoned decimal format.

The USING request modifier precedes the BEGIN TRANSACTION statement, while the BEGIN TRANSACTION statement and the request associated with the USING clause are entered as one multistatement request.



```
.SET RECORDMODE ON
.IMPORT DATA FILE = r13sales.dat;

USING (zonedec CHARACTER(4))
BEGIN TRANSACTION
;INSERT INTO dectest (colz = :zonedec (DECIMAL(4),
FORMAT '9999S')) ;

USING (zonedec CHARACTER(4))
INSERT INTO Dectest
    (colz = :zonedec (DECIMAL(4), FORMAT '9999S')) ;

USING (zonedec CHARACTER(4))
INSERT INTO Dectest
    (colz = :zonedec (DECIMAL(4), FORMAT '9999S')) ;
END TRANSACTION;
```

In BTEQ applications, you can combine USING request modifiers with the .REPEAT command to perform multiple insertions automatically.

## Example: Inline Mode Processing of a Large Object

The following example passes the BLOB values for column *b* inline:

```
.SET INDICDATA ON
.IMPORT DATA FILE=mar08sales.dat

USING (a INTEGER,
      b BLOB(60000))
INSERT INTO mytable VALUES (:a, :b);
```

## Example: Deferred Mode Processing of a Large Object

The following example passes the CLOB values for column *b* in deferred chunks:

```
.SET INDICDATA ON
.IMPORT DATA FILE=mar08sales.dat

USING (a INTEGER,
      b CLOB AS DEFERRED)
INSERT INTO mytable VALUES (:a, :b);
```

## Example: Deferred Mode Processing of CLOBs Using the DEFERRED BY NAME Phrase

The following example shows the use of the BY NAME phrase for deferred mode processing of large objects using BTEQ and CLOBs.

```
.SET INDICDATA ON

CREATE TABLE tabf (
  i1 INTEGER,
  v1 VARCHAR(256));

INSERT INTO tabf (1, 'c:\temp\vconfig.txt');

.EXPORT INDICDATA FILE=scg0720.dat

SELECT *
FROM tabf;

.EXPORT reset

CREATE TABLE tabc (
  i1 INTEGER,
  c1 CLOB);

.IMPORT INDICDATA FILE=scg0720.dat

USING (a INTEGER,
      b CLOB AS DEFERRED BY NAME)
INSERT INTO tabc (:a, :b);
```

## Example: DEFERRED MODE Processing of BLOBs Using the DEFERRED BY NAME Phrase

The following example shows the use of the BY NAME phrase for deferred mode processing of large objects using BTEQ and BLOBs.

```
CREATE TABLE tabf2 (
  i1 INTEGER,
  v1 VARCHAR(256));

INSERT INTO tabf2 (1, 'c:\temp\data.dat');
```

```

.SET INDICDATA ON
.EXPORT INDICDATA FILE=scg0720.dat

SELECT *
FROM tabf;

.EXPORT RESET

CREATE TABLE tabb (
  i1 INTEGER,
  c1 BLOB);

.IMPORT INDICDATA FILE=scg0720.dat

USING (a INTEGER,
       b BLOB AS DEFERRED BY NAME)
INSERT INTO tabb (:a, :b);

```

### Example: Locator Mode Processing of a Large Object

The first example shows how locator *b* is used to copy an existing, Teradata platform-resident BLOB from its current base table location into a base table named mytable without any data transferred to the client.

```

USING (a INTEGER,
       b BLOB AS LOCATOR)
INSERT INTO mytable VALUES (:a, :b);

```

The second example shows the BLOB data identified by locator *b* being returned to the client.

```

.SET INDICDATA ON
.EXPORT INDICDATA FILE=udbsales.dat

USING (b BLOB AS LOCATOR)
SELECT :b;

.EXPORT RESET

```

### Example: Using a Locator Multiple Times Within a Session

This example shows the same locator being used in more than one request within a session:

```

USING (a CLOB AS LOCATOR)
SELECT :a;

```

```

USING (a INTEGER,
       b CLOB AS LOCATOR)
INSERT INTO tab2 (:a, :b);

```

## Example: Iterated Requests

The following example shows one way of performing an iterated request in BTEQ:

```

.IMPORT DATA FILE = r13sales.dat;
.REPEAT RECS 200 PACK 100

USING (pid INTEGER, pname CHAR(12))
INSERT INTO ptable VALUES(:pid, :pname);

```

The `.REPEAT` command specifies that BTEQ should read up to 200 data records and pack a maximum of 100 data records with each request.

## Example: USING Request Modifier with TOP Operator

This example passes the value for  $n$  in the `TOP  $n$`  operator in its `SELECT` request using the `INTEGER` parameter  $a$  as defined in the `USING` request modifier.

```

.SET RECORDMODE ON
.IMPORT DATA FILE=employee.dat

USING (a INTEGER)
SELECT TOP :a *
FROM employee;

```

## Example: Non-Support for Iterated Requests With TOP $n$

The following request indicates the lack of support for specifying a `TOP  $n$`  value as a parameterized variable in a `USING` request modifier for iterated arrays.

```

.REPEAT 1 PACK 5
BTEQ -- Enter your DBC/SQL request or BTEQ command:

USING (a INTEGER, b INTEGER, c INTEGER)
SELECT TOP :a *
FROM t1;

*** Starting Row 0 at Tue Aug 05 11:46:07 2008

```

```
*** Failure 6906Iterated request:Disallowed statement type (TOP N).
Statement# 1, Info =0
*** Total elapsed time was 1 second.
```

## Example: Dynamic UDT Expressions

The following example shows one way to use a dynamic UDT expression in a table UDF with a USING request modifier:

```
.IMPORT INDICDATA FILE lobudt003.data

USING(p1 INTEGER, p2 INTEGER, p3 BLOB)
SELECT *
FROM TABLE(dyntbf003(:p2,
                      NEW VARIANT_TYPE(:p2 AS a,:p3 AS b))) AS t1
ORDER BY 1;

*** Query completed. 2 rows found. 3 columns returned.
*** Total elapsed time was 1 second.
```

[illegible]

## Example: USING and SQL UDF Invocation

This example invokes the SQL UDF *value\_expression* in a USING clause-based SELECT request.

```

USING (a INTEGER, b INTEGER)
SELECT test.value_expression(:a, :b) AS cve
FROM t1
WHERE t1.a1 = :a
AND    t1.b1 = :b;

```

This example invokes the SQL UDF *value\_expression* in a USING clause-based DELETE request.

```
USING (a INTEGER, b INTEGER)
DELETE FROM t1
WHERE test.value expression(:a, :b) > t1.a1;
```

This example, which invokes the SQL UDF *value\_expression* in a USING clause-based SELECT request with mismatches of the data types of the arguments, aborts and returns an error.

```
USING (a CHARACTER(10), b CHARACTER(10))  
SELECT test.value_expression(:a, :b) AS cve  
FROM t1  
WHERE t1.a1 = :a  
AND   t1.b1 = :b;
```

# Query and Workload Analysis Statements

## Overview

These topics describe Teradata SQL DML statements used to collect or analyze data demographics and statistics. The collected data is used to populate user-defined QCD tables with data used by various Teradata utilities to analyze query workloads as part of the ongoing process to reengineer the database design process. For example, the Teradata Index Wizard determines optimal secondary and single-table join index sets to support your query workloads.

For information about query capture databases (QCDs), see *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

The Teradata Index Wizard is described in the following documents:

- *Teradata® Index Wizard User Guide*, B035-2506
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142

For information about Teradata Database data control language (DCL) statements, see *Teradata Vantage™ SQL Data Control Language*, B035-1149.

For information about HELP and SHOW statements, see *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

For information about database query logging, see *Teradata Vantage™ - Database Administration*, B035-1093.

## COLLECT DEMOGRAPHICS

### Purpose

Collects various table demographic estimates and writes the data to the *DataDemographics* table of a user-defined QCD database for subsequent analysis by the Teradata Index Wizard.

For more information about index analysis, see:

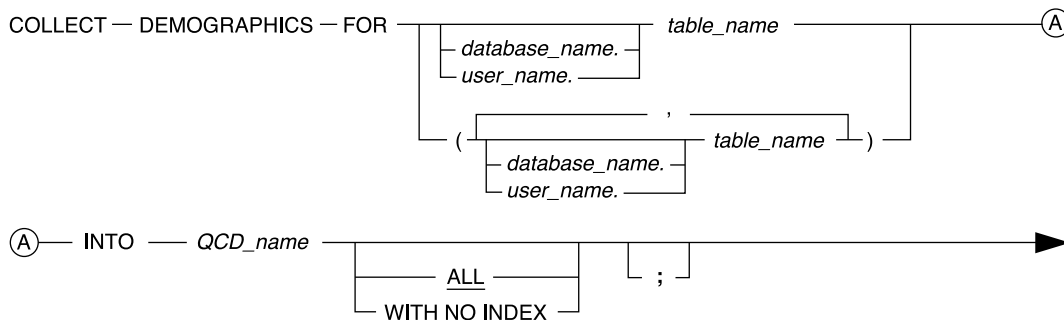
- [COLLECT STATISTICS \(QCD Form\)](#)
- [INITIATE INDEX ANALYSIS](#)
- [INSERT EXPLAIN](#)
- [RESTART INDEX ANALYSIS](#)
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142
- *Teradata® Index Wizard User Guide*, B035-2506

## Required Privileges

You must have all of the following privileges to perform COLLECT DEMOGRAPHICS:

- DELETE on the *DataDemographics* table in *QCD\_name*.
- INSERT on the *DataDemographics* table in *QCD\_name* or INSERT on the *QCD\_namedatabase*.
- SELECT on the specified tables or containing databases or users.

## Syntax



## Syntax Elements

**database\_name**

**user\_name**

Containing database or user for *table\_name* if something different from the current database or user.

**table\_name**

Names of tables for which data demographic estimates are to be collected.

**QCD\_name**

Name of the QCD database into which the collected data demographic estimates are to be written.

**ALL**

Collect primary and index subtable data demographic estimates.

This is the default.

**WITH NO INDEX**

Exclude index subtable data demographic estimates from the collection and collect only primary data estimates.

## ANSI Compliance

`COLLECT DEMOGRAPHICS` is a Teradata extension to the ANSI SQL:2011 standard.

## Invocation

Normally invoked using the Teradata Index Wizard utility or the Visual Explain tool.



## Demographics Collected

COLLECT DEMOGRAPHICS determines and writes the following information for each specified table into the DataDemographics table of the specified query capture database:

- Subtable type
- Subtable ID
- Estimated cardinality
- Estimated average row length
- Various system-related information

One row of information is collected in the appropriate DataDemographics subtable of the specified QCD for each AMP that is online at the time the request is performed.

COLLECT DEMOGRAPHICS does not capture information for the QCD table *TableStatistics*. *TableStatistics* is used only during the collection of statistics invoked by performing an INSERT EXPLAIN ... WITH STATISTICS or COLLECT STATISTICS (QCD Form) request.

For more detailed information, see *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

## Relationship to INSERT EXPLAIN WITH STATISTICS AND DEMOGRAPHICS

The information retrieved for COLLECT DEMOGRAPHICS and INSERT EXPLAIN WITH STATISTICS AND DEMOGRAPHICS is identical. The principal differences are:

- INSERT EXPLAIN WITH STATISTICS AND DEMOGRAPHICS uses an SQL query to collect the information, while COLLECT DEMOGRAPHICS obtains the information directly.
- Demographics captured by INSERT EXPLAIN WITH STATISTICS AND DEMOGRAPHICS are automatically deleted whenever you delete the relevant query plans.

Demographics captured by COLLECT DEMOGRAPHICS are not deleted when you perform associated DROP actions on the subject table and must be deleted explicitly.

## COLLECT DEMOGRAPHICS Not Supported From Macros

You cannot specify a COLLECT DEMOGRAPHICS request from a macro. If you execute a macro that contains a COLLECT DEMOGRAPHICS request, Teradata Database aborts the request and returns an error.

## Example: Collect Demographic Estimates

Collect demographic estimates on *table\_1* in database *db\_1* and write the results into the *DataDemographics* table of user-defined QCD *MyQCD*.

```
COLLECT DEMOGRAPHICS FOR db_1.table_1 INTO MyQCD;
```

### Example: Update Demographic Estimates

An entry already exists in the *DataDemographics* table for the specified table, then it is updated with the currently collected demographic estimates. The value of *TimeStamp* is updated to reflect the current time.

```
COLLECT DEMOGRAPHICS FOR db_1.table_1 INTO MyQCD; /* Inserts data
into DataDemographics table of MyQCD */
COLLECT DEMOGRAPHICS FOR db_1.table_1 INTO MyQCD; /* Updates data
in DataDemographics table of MyQCD */
```

## COLLECT STATISTICS (QCD Form)

### Purpose

Collects demographic data for one or more columns and indexes of a table, computes a statistical profile of the collected data, and stores the synopsis in the TableStatistics table of the specified QCD database.

Statistics collected by this statement are used for index analysis and validation tasks performed by database query analysis tools. These statistics are not used by the Optimizer to process queries. For information on COLLECT STATISTICS (Optimizer Form), see *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

For more information about index analysis, see:

- [COLLECT DEMOGRAPHICS](#)
- [INITIATE INDEX ANALYSIS](#)
- [INSERT EXPLAIN](#)
- [RESTART INDEX ANALYSIS](#)
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142
- [DROP STATISTICS \(QCD Form\)](#)
- “HELP STATISTICS (Optimizer Form)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184

For more information about client-based query analysis tools, see:

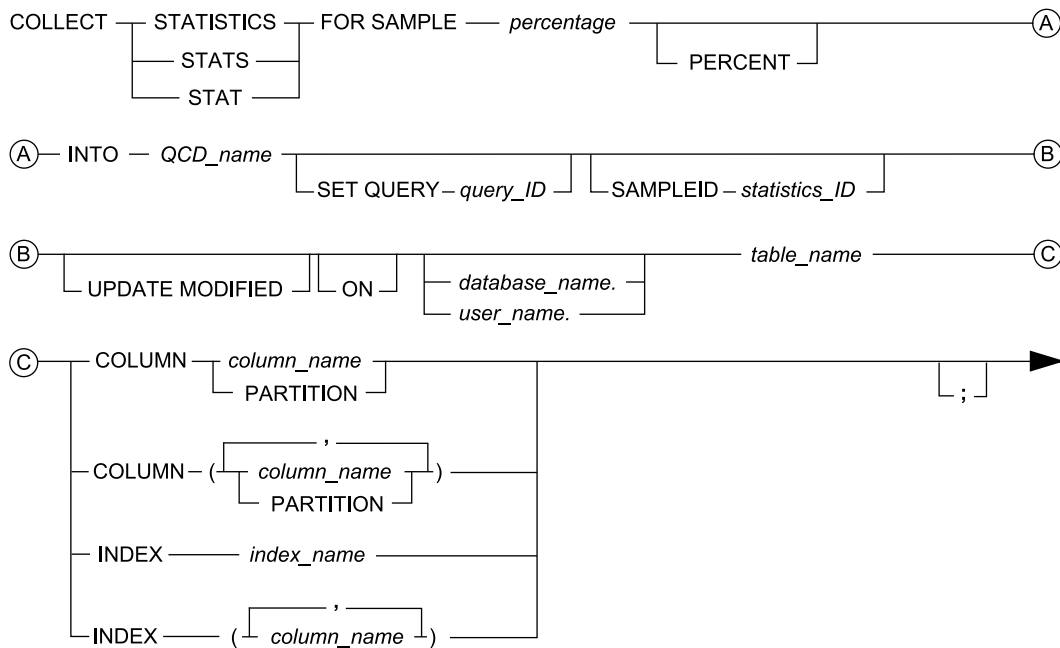
- *Teradata® Index Wizard User Guide*, B035-2506
- *Teradata® Viewpoint User Guide*, B035-2206, Stats Manager topic

### Required Privileges

You must have the following privileges to perform COLLECT STATISTICS (QCD Form):

- INDEX or DROP TABLE on *table\_name* or its containing database.
- INSERT and UPDATE on the TableStatistics table or its containing QCD database.

## Syntax



## Syntax Elements

### FOR SAMPLE *percentage*

### FOR SAMPLE *percentage* PERCENT

Percentage of table rows to be sampled to build the statistics for the specified column and index sets. The percentage value must be less than 100.

For example, if you specify a value of 5, then 5 percent of the rows for *table\_name* on each AMP are read to collect their statistics.

This option is not recognized for single-column statistics requests for the PARTITION column of a row-partitioned table. The system automatically increases the percentage to 100.

The restriction for COLLECT STATISTICS (Optimizer Form) on using sampling with columns of a row-partitioned table that are also members of the partitioning expression for that table does not apply to COLLECT STATISTICS (QCD Form).

PERCENT is an optional keyword to indicate that the value specified for *percentage* is a percentage.

### INTO *QCD\_name*

Name of the QCD database into which the collected sampled statistics are to be written.

The sampled statistics are written to the TableStatistics table of the specified QCD database.

### SET QUERY *query\_ID*

Value inserted into QCD.TableStatistics.QueryId.

Each unique composite of *query\_ID* and *statistics\_ID* enables you to store a separate set of statistics in QCD.TableStatistics for a particular column or index.

The default value is 0.

#### **SAMPLEID *statistics\_ID***

Value to be inserted into QCD.TableStatistics.StatisticsId.

Each unique composite of *query\_ID* and *statistics\_ID* enables you to store a separate set of statistics in QCD.TableStatistics for a particular column or index.

The default value is 1.

#### **UPDATE MODIFIED**

Store modified statistics stored in QCD.TableStatistics.ModifiedStats.

If you do not specify the UPDATE MODIFIED option, the system stores unmodified statistics in QCD.TableStatistics.StatisticsInfo.

#### ***database\_name user\_name***

Name of the containing database or user for *table\_name* if different from the current database or user.

#### ***table\_name***

Name of the table for which the specified column and index sampled statistics and sample size are to be collected.

You cannot collect QCD statistics on volatile, journal, global temporary, or global temporary trace tables.

#### **COLUMN *column\_name***

Name of a column set on which sampled statistics are to be collected.

You can collect statistics on both a column set and on the system-derived PARTITION column in the same COLLECT STATISTICS request.

You cannot collect statistics on UDT or LOB columns.

#### **COLUMN PARTITION**

Statistics are to be collected on the system-derived PARTITION column for a table. The value collected is the system-derived partition number, which ranges from 1 through 65,535 inclusive, or is zero for non-partitioned.

You can collect statistics on both a column set and on the system-derived PARTITION column in the same COLLECT STATISTICS request.

Even though *table\_name* need not identify a partitioned table, and the system does not return an error if you request PARTITION column statistics on a nonpartitioned table, the operation serves no purpose.

You cannot collect PARTITION statistics on the following:

- Join indexes
- Global temporary tables

- Volatile tables

**INDEX *index\_name***

Name of the index on which sampled statistics are to be collected.

**INDEX *column\_name***

Names of the columns in the column set on which sampled statistics are to be collected.

**ANSI Compliance**

COLLECT STATISTICS (QCD Form) is a Teradata extension to the ANSI SQL:2011 standard.

## Usage Notes

### Invocation

Normally invoked by client-based Teradata Database query analysis tools.

### Where Column and Index Statistics Are Stored

Table statistics collected by the QCD form of COLLECT STATISTICS are stored in the QCD table named TableStatistics. See *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

### Rules and Guidelines for COLLECT STATISTICS (QCD Form)

Unless otherwise noted, the rules and guidelines for collecting QCD statistics are the same as those for Optimizer statistics. See “Rules and Guidelines for COLLECT STATISTICS (Optimizer Form)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

### Collecting QCD Statistics on Multiple Columns

If you have defined a column set as a composite index, it makes no difference whether you collect statistics on the composite using the COLUMN keyword or the INDEX keyword. The result is identical.

For example, suppose you have created an index on the column set (x1,y1) of table t1. The following COLLECT STATISTICS requests gather the identical statistics for the named QCD into the identical QCD table columns:

```
COLLECT STATISTICS FOR SAMPLE 30 PERCENT ON t1
  INTO myqcd COLUMN (x1,y1);
```

```
COLLECT STATISTICS FOR SAMPLE 30 PERCENT ON t1
  INTO myqcd INDEX (x1,y1);
```

## Collecting QCD Statistics on the PARTITION Column of a Table

For information about why you should collect PARTITION statistics for the Optimizer, see “Collecting Statistics on the System-Derived PARTITION Column and the Row Partitioning Column Set of a Partitioned Table” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

Because PARTITION is not a reserved keyword, you can use it as the name for any column in a table definition, but you should not. This practice is *strongly* discouraged in all cases, and particularly for partitioned tables, because if you name a non-PARTITION column *partition*, the system resolves it as a regular column. As a result, you cannot collect statistics on the system-derived PARTITION column of any table that also has a user-defined column named *partition*.

If a table is partitioned by a single-column expression, its column statistics are inherited as PARTITION statistics. In this special case, you need not collect single-column PARTITION statistics.

For example, assume the following table definition:

```
CREATE TABLE table_1 (
  col_1 INTEGER,
  col_2 INTEGER)
PRIMARY INDEX(col_1) PARTITION BY col_2;
```

If you collect individual column statistics on *col\_2*, which is the partitioning column, then those statistics are also inherited as PARTITION statistics.

You can also collect PARTITION statistics on NPPI tables, which quickly provides up to date statistics on the number of rows in the table. This is because the partition number for every row in an NPPI table is 0.

You can specify a USING SAMPLE clause to collect single-column PARTITION statistics, but the specification is ignored. The system automatically resets the sampling percentage to 100. The sampling field in a detailed statistics report is always reported as 0 to document this behavior. Note that all valid sampling percentages specified for a USING SAMPLE clause *are* honored for multicolumn PARTITION statistics.

There is a limit of 32 sets of multicolumn statistics that can be collected on a given table. Because single-column PARTITION statistics use index ids in the same range as multicolumn statistics (between 129 and 160, inclusive), collecting PARTITION statistics effectively reduces the limit on the number of multicolumn statistics that can be collected to 31.

The system ignores user-specified column ordering for multicolumn PARTITION statistics. This is consistent with non-PARTITION multicolumn statistics and multicolumn indexes. The columns are ordered based on their internal field id. Because the system-derived PARTITION column has field id value of 0, it always takes the first position in multicolumn statistics.

## **COLLECT STATISTICS (QCD Form) and UDTs**

You cannot collect statistics on UDT columns.

## **COLLECT STATISTICS (QCD Form) And Large Objects**

You cannot collect statistics on BLOB or CLOB columns.

## **COLLECT STATISTICS (QCD Form) Does Not Collect Statistics For The Optimizer**

Unlike the standard `COLLECT STATISTICS` statement, `COLLECT STATISTICS (QCD Form)` does not write the statistics it collects into the data dictionary for use by the Optimizer. The `COLLECT STATISTICS (QCD Form)` writes its statistics into the `TableStatistics` table of the specified QCD database, where the information is used by various database query analysis tools to perform index analysis and validation.

Because `COLLECT STATISTICS (QCD Form)` does not write its statistics to the data dictionary, it does not place locks on dictionary tables. Through appropriate control of the sample size, you can collect sample statistics for index analysis during production hours without a noticeable negative effect on system performance.

## **Implicit Recollection of QCD Statistics Is Not Supported**

Unlike `COLLECT STATISTICS (Optimizer Form)`, you cannot recollect statistics implicitly on indexes and columns for which statistics have been collected in the past. You must specify an explicit index or column name each time you collect statistics using `COLLECT STATISTICS (QCD Form)`.

## **COLLECT STATISTICS (QCD Form) Is Not Treated As DDL By The Transaction Manager**

You can place a `COLLECT STATISTICS (QCD Form)` request anywhere within a transaction because it is not treated as a DDL statement by the Transaction Manager.

## **COLLECT STATISTICS (QCD Form) Collects New Statistics Even If Statistics Exist In the Data Dictionary or QCD**

`COLLECT STATISTICS (QCD Form)` always collects its own fresh statistics on candidate index columns, even if a statistical profile of the specified columns or index columns already exists in the data dictionary or QCD for *table\_name*, to ensure that index analyses are always performed using current statistics.

Fresh statistics are also collected whenever you perform an INSERT EXPLAIN request and specify the WITH STATISTICS clause. See [INSERT EXPLAIN](#) for more information.

## Quality of Statistics As a Function of Sample Size

The larger the sample size used to collect statistics, the more likely the sampled statistics are an accurate representation of population statistics. There is a trade-off for this accuracy: the larger the sample size, the longer it takes to collect the statistics.

You cannot use sampling to collect COLUMN statistics for the partitioning columns of row-partitioned tables and you should not use sampling to collect INDEX statistics for those columns. The collect statistics on the system-derived PARTITION or PARTITION#*Ln* columns.

## Examples

### Example: Collect Statistics on a Single-Column NUSI

This example collects statistics on a single-column NUSI named *orderDateNUSI* on the *order\_date* column from a random sample of 10 percent of the rows in the *orders* table and writes them to the *TableStatistics* table of the QCD database named *MyQCD*.

```
COLLECT STATISTICS FOR SAMPLE 10 PERCENT
ON orders INDEX orderDateNUSI
INTO MyQCD;
```

### Example: Collect Statistics On Index Using an Alternate Syntax

This example collects sampled statistics on the same index as [Example: Collect Statistics on a Single-Column NUSI](#) using different syntax:

```
COLLECT STATISTICS FOR SAMPLE 10 PERCENT
ON orders INDEX (order_date)
INTO MyQCD;
```

### Example: Collecting Single-Column PARTITION Statistics

This example collects statistics on the system-derived PARTITION column for the row-partitioned orders table and writes them to a user-defined QCD called *myqcd*.

Even though you have specified a sampling percentage of 30 percent, the system ignores it and uses a value of 100 percent. See [Collecting QCD Statistics on the PARTITION Column of a Table](#).



```
COLLECT STATISTICS FOR SAMPLE 30 PERCENT
ON orders COLUMN PARTITION
INTO myqcd;
```

## Example: Collecting Multi-Column PARTITION Statistics

This example collects multi-column sampled statistics on the following column set for the row-partitioned orders table and writes them to a user-defined QCD called *myqcd*:

- System-derived PARTITION column
- *quant\_ord*
- *quant\_shpd*

Because you are requesting multicolumn statistics, the system honors the specified sampling percentage of 20 percent, unlike the case for single-column PARTITION statistics. See [Collecting QCD Statistics on the PARTITION Column of a Table](#) and [Example: Collect Statistics on a Single-Column NUSI](#).

```
COLLECT STATISTICS FOR SAMPLE 20 PERCENT
ON orders COLUMN (quant_ord, PARTITION, quant_shpd)
INTO myqcd;
```

## DROP STATISTICS (QCD Form)

### Purpose

Drops sampled statistics on the specified table from the TableStatistics table in the specified *QCD\_name*.

See these related statements for more information:

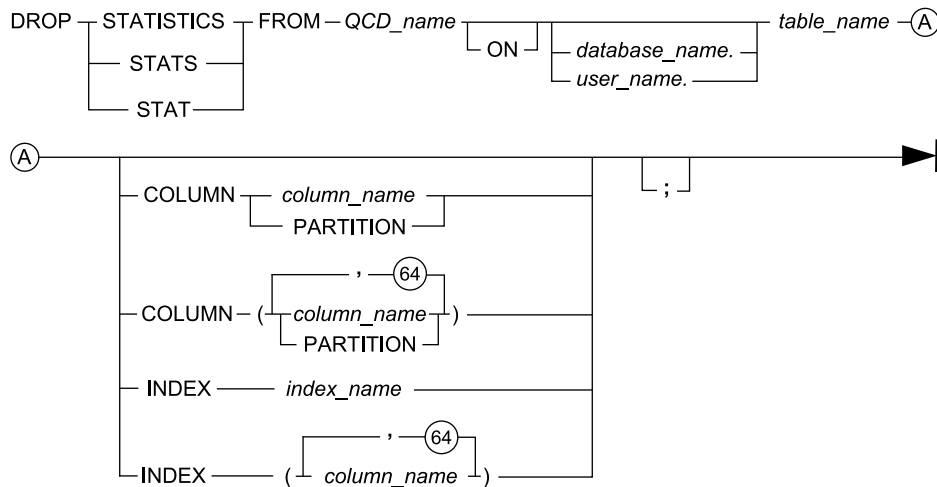
- [COLLECT STATISTICS \(QCD Form\)](#)
- “HELP STATISTICS (QCD Form)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

### Required Privileges

You must have the following privileges to perform DROP STATISTICS (QCD Form):

- INDEX or DROP TABLE on *table\_name* or its containing database
- DELETE on the TableStatistics table or its containing QCD database

## Syntax



## Syntax Elements

### FROM QCD\_name

Name of the QCD database from which statistics on the specified table columns and indexes are to be dropped from the *TableStatistics* table.

### database\_name | user\_name

Name of the containing database or user for *table\_name* if different from the current database or user.

### table\_name

Name of the table for which sampled column and index statistics are to be dropped.

### COLUMN column\_name

Set of non-indexed columns for which sampled statistics are to be dropped.

You cannot drop statistics on a UDT column.

### COLUMN PARTITION

Statistics are to be dropped on the system-derived PARTITION column for a table.

You cannot reference the PARTITION#Ln columns of a table in a DROP STATISTICS request.

For more information about the PARTITION column, see [Collecting QCD Statistics on the PARTITION Column of a Table](#).

### INDEX index\_name

Name of the index for which sampled statistics are to be dropped.

### INDEX column\_name

Column set for which sampled statistics are to be dropped.

## ANSI Compliance

DROP STATISTICS (QCD Form) is a Teradata extension to the ANSI SQL:2011 standard.

## Invocation

Normally invoked using client-based database query analysis tools.

## Difference Between DROP STATISTICS (QCD Form) and DROP STATISTICS (Optimizer Form)

Unlike the DROP STATISTICS (Optimizer Form) statement, DROP STATISTICS (QCD Form) does not drop the statistics kept in the data dictionary for use by the Optimizer. The DROP STATISTICS (QCD Form) drops statistics from the TableStatistics table of the specified QCD database.

## DROP STATISTICS (QCD Form) and INSERT EXPLAIN WITH STATISTICS

DROP STATISTICS (QCD Form) does not drop statistics from the QCD that were captured using an INSERT EXPLAIN WITH STATISTICS request. These statistics are dropped automatically whenever their corresponding query plan is deleted from the QCD.

## DROP STATISTICS (QCD Form) and Transaction Processing

You can place a DROP STATISTICS (QCD Form) request anywhere within a transaction because it is not treated as a DDL statement by the Transaction Manager.

## Dropping Statistics On UDTs

You cannot DROP STATISTICS on a UDT column.

## Example: Drop Statistics on All Columns

The following example drops statistics on all columns and indexes for the orders table from the TableStatistics table of the QCD database named *MyQCD*.

```
DROP STATISTICS FROM MyQCD
ON orders;
```

## Example: Drop Statistics on a Single Column

The following example drops statistics on a single-column NUSI named *orderDateNUPI* from the TableStatistics table of the QCD database named *MyQCD*.

```
DROP STATISTICS FROM MyQCD
ON orders INDEX orderDateNUPI;
```

## Example: Dropping PARTITION Statistics

The following DROP STATISTICS request drops statistics on the PARTITION column only for the table named *table\_1* from the TableStatistics table in the QCD named *QCD\_11*:

```
DROP STATISTICS FROM QCD_11 ON table_1
COLUMN PARTITION;
```

The following DROP STATISTICS request drops statistics on the column named *column\_1* and the system-derived PARTITION column for the table named *table\_2* from the *TableStatistics* table in the QCD named *QCD\_12*:

```
DROP STATISTICS FROM QCD_12 ON table_2
COLUMN (column_1, PARTITION);
```

The following table-level DROP STATISTICS request drops all the statistics, including statistics on the system-derived PARTITION column, for the table named *table\_3* from the *TableStatistics* table in the QCD named *QCD\_13*:

```
DROP STATISTICS FROM QCD_13 ON table_3;
```

## DUMP EXPLAIN

### Purpose

Captures the Optimizer plan information for a query and returns it to the requestor in the form of a script containing a series of INSERT requests.

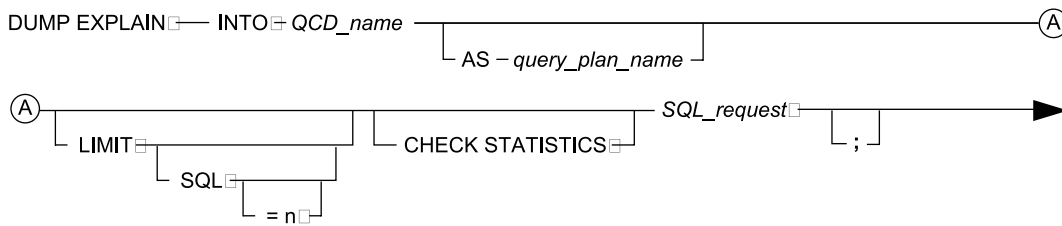
For more information about index analysis, see:

- “BEGIN QUERY LOGGING” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144
- “REPLACE QUERY LOGGING” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144
- [COLLECT DEMOGRAPHICS](#)
- [COLLECT STATISTICS \(QCD Form\)](#)
- [INITIATE INDEX ANALYSIS](#)
- [INSERT EXPLAIN](#)
- [RESTART INDEX ANALYSIS](#)
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ - Database Design*, B035-1094
- *Teradata® Index Wizard User Guide*, B035-2506
- *Teradata® Viewpoint User Guide*, B035-2206, Stats Manager topic
- *Teradata® Visual Explain User Guide*, B035-2504
- *Teradata® System Emulation Tool User Guide*, B035-2492

### Required Privileges

You must have INSERT privileges on all the tables in the specified query capture database.

## Syntax



## Syntax Elements

### INTO QCD\_name

Name of a user-defined query capture database.

The database named *QCD\_name* need not exist on the target system. However, a database named *QCD\_name* must exist on the test system on which the generated script is performed.

Use the Control Center feature of the Visual Explain tool to create your QCD databases.

### AS query\_plan\_name

Optional user-defined name under which the query plan information is to be stored. For information on object naming, see *Teradata Vantage™ SQL Fundamentals*, B035-1141.

If you do not specify a *query\_plan\_name*, the query plan information is stored with a null name. Each query plan is stored with a unique non-null Query ID, enabling you to distinguish among the query plans in a particular database. See *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142. Query IDs are not unique across databases.

*query\_plan\_name* is not constrained to be unique.

You can store *query\_plan\_name* as *query\_plan\_name* if you enclose the pad character-separated words in APOSTROPHE (u+0027) characters as “query plan name”.

### LIMIT

#### LIMIT SQL

#### LIMIT SQL=*n*

Place a limit on the size of the query, DDL, view text, and predicate text captured for the QCD tables Query, Relation, ViewText, and Predicate, respectively.

If you do not specify this clause, then the system captures complete text.

The value for *n* indicates the upper limit on the amount of text to capture.

If you specify either LIMIT by itself or LIMIT SQL without a value, then the clause is equivalent to LIMIT SQL = 0, and no text is captured.

### CHECK STATISTICS

Capture COLLECT STATISTICS recommendations for *SQL\_request* in the StatsRecs QCD table. See *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

**SQL\_request**

DML statement whose Optimizer plan information is to be captured and returned to the requestor as a series of INSERT requests.

*SQL\_request* is limited to the following statements:

- DELETE
- EXEC (Macro Form)
- INSERT
- MERGE
- SELECT
- UPDATE

**ANSI Compliance**

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

**Invocation**

Normally invoked using client-based database query analysis tools.

**DUMP EXPLAIN Actions Without An Existing QCD**

Ensure that there is an existing *QCD\_name* database on the test system before running the script produced by DUMP EXPLAIN for the first time. You can use the Control Center feature of the Visual Explain tool to create your QCD databases.

If a *QCD\_name* database or any of its tables does not exist, or if you do not have INSERT privileges on one or more *QCD\_name* database tables when you perform the script produced by DUMP EXPLAIN, then the system displays an appropriate error message and terminates performance of the script.

**Actions Performed by DUMP EXPLAIN**

DUMP EXPLAIN performs the following actions in the order indicated.

1. Runs an EXPLAIN on the SQL DML request specified by *SQL\_statement*.
2. Captures the Optimizer plan output of that EXPLAIN.
3. Returns the output to the requestor as a script containing a series of INSERT requests designed to be used to update the appropriate tables in the user-specified query capture database.

You might want to use DUMP EXPLAIN rather than INSERT EXPLAIN if you are collecting information from several different machines and you want to ensure that only the selected QCD on the appropriate machine is updated with the results or if you do not want to update the QCD during heavy workload periods. In this case, you could submit the DUMP requests as part of a batch job during a less burdened workload period.

**DUMP EXPLAIN Is Not Valid Within a Multistatement Request**

You cannot submit an DUMP EXPLAIN request as part of a multistatement request, though you can submit an DUMP EXPLAIN request *for* a multistatement request.

If you attempt to submit a multistatement request that contains an DUMP EXPLAIN request, the multistatement request aborts and returns an error.

Note that while the DUMP EXPLAIN request in the following example superficially appears to be a valid statement in a multistatement request, it actually captures the query plan for a multistatement request that *follows* it, and is not itself part of that multistatement request. As a result, the request is treated like any other DUMP EXPLAIN and completes successfully.

```
DUMP EXPLAIN INTO qcd SELECT * FROM d1.t1
;SELECT * FROM d1.t1;

*** Insert completed. One row added.
*** Total elapsed time was 1 second.
```

### DUMP EXPLAIN Does Not Support Capturing Output in XML

Unlike INSERT EXPLAIN and the EXPLAIN request modifier, the DUMP EXPLAIN statement does *not* support the capture of query plan data as an XML document.

### Effects of Request Cache Peeking on DUMP EXPLAIN Outcomes

When a data parcel is submitted with an DUMP EXPLAIN request, the plan might be generated with peeked USING and CURRENT\_DATE or DATE values, or both. If any of these values are peeked, then the query plan shows them.

If no data parcel is submitted with an DUMP EXPLAIN request, the resulting plan is generated without peeking at USING or CURRENT\_DATE, or DATE values, so it is a generic plan by definition. Note that the Visual Explain, Teradata System Emulation Tool, and Teradata Index Wizard do not accept USING data as input while capturing query plans using DUMP EXPLAIN requests unless those requests are submitted using BTEQ or Teradata SQL Assistant.

The Teradata Index Wizard internally generates plans for workload queries in order to estimate workload costs, which are used to determine optimal index recommendations. When queries in the workloads specify USING request modifiers, the plan is generated without peeking at USING, CURRENT\_DATE, or DATE values. Because of these factors, Request Cache peeking has no impact on the resulting index. Given that workload analyses should be independent of USING values, this behavior is correct.

### Examples

The following example set uses the same SELECT request throughout to illustrate how the different syntax options of DUMP EXPLAIN produce different results.

Each DUMP EXPLAIN request generates a set of INSERT requests for the query plan produced by the Optimizer for the SELECT request that it modifies.

### Example: DUMP EXPLAIN Request

The output of this DUMP EXPLAIN request is referenced under the query plan name *EmployeeSmithQuery* in the *TLE\_queries* database.

```
DUMP EXPLAIN INTO TLE_queries AS EmployeeSmithQuery
SELECT emp_id, emp_address
FROM employee
WHERE emp_name = 'Smith';
```

### Example: DUMP EXPLAIN Request With Null Query Plan Name

The output of this DUMP EXPLAIN request has a null query plan name in the *TLE\_queries* database. The difference between this result and that of [Example: DUMP EXPLAIN Request](#) is that the query plan for this example has no name.

```
DUMP EXPLAIN INTO TLE_queries
SELECT emp_id, emp_address
FROM employee
WHERE emp_name = 'Smith';
```

### Example: DUMP EXPLAIN Request With Query Plan Name

The output of this DUMP EXPLAIN request is referenced under the query plan name *Employee Smith Query* in the *TLE\_queries* database.

```
DUMP EXPLAIN INTO TLE_queries AS "Employee Smith Query"
SELECT emp_id, emp_address
FROM employee
WHERE emp_name = 'Smith';
```

## EXPLAIN Request Modifier

### Purpose

Reports a summary of the query plan generated by the SQL query optimizer to process any valid SQL request, that is, the steps the system would use to resolve a request. The Optimizer processes an explained request in the same way that the request would be processed without the EXPLAIN modifier, except that the SQL within the request is not actually executed. However, for a dynamic plan, the request is partially executed.

Optionally, you can display the output as XML text instead of plain English text.

For additional information about how to interpret the reports generated by an EXPLAIN request modifier or the related Visual Explain utility, see:

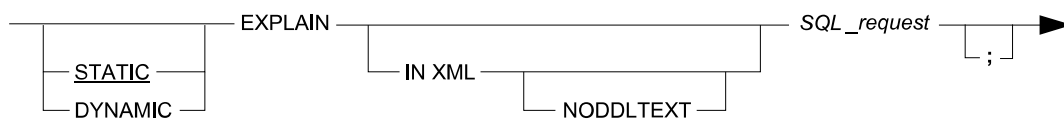
- [INSERT EXPLAIN](#)
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142
- *Teradata® Visual Explain User Guide*, B035-2504

### Required Privileges

To EXPLAIN a request, you must have the permissions that are required to execute that request.



## Syntax



## Syntax Elements

### STATIC

An EXPLAIN for the static plan is generated. This is the default. If applicable, the static plan indicates the request is eligible for incremental planning and execution (IPE).

### IN XML

Return the output as XML text rather than as plain English text. For more information, see *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

### NO DDLTEXT

Do not capture the DDL text in the XML EXPLAIN text.

### SQL\_request

SQL request for which to return Optimizer processing information.

## ANSI Compliance

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

Other SQL dialects support similar non-ANSI standard statements with names such as EXPLAIN PLAN.

## Usage Notes

### EXPLAIN and the USING Request Modifier

If you specify both an EXPLAIN request modifier and a USING request modifier for the same request, the EXPLAIN modifier must precede the USING modifier. See [USING Request Modifier](#). A dynamic plan cannot be displayed for requests with a USING request modifier.

### Capturing EXPLAIN Text Output in XML Format

The IN XML option returns the EXPLAIN text as XML text rather than as plain English text. See *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142 for the XML plan attributes and the XML schema used by the XML output produced by the IN XML option.

The EXPLAIN IN XML request modifier can also specify that the output be returned in one of two possible formats:

- As XML text with any associated DDL text. You specify this with the IN XML option.
- As XML text, but without any associated DDL text. You specify this with the IN XML NODDLTEXT option.

A complementary option for the BEGIN QUERY LOGGING and REPLACE QUERY LOGGING statements optionally creates a compact XML version of the information that can be stored in a DBQL table. See “BEGIN QUERY LOGGING” and “REPLACE QUERY LOGGING” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## EXPLAIN Report Overview

EXPLAIN provides a summary of the access and join plans generated by the Optimizer for the query SQL requests. The report details which indexes would be used to process the request, identifies any intermediate spools that would be generated, indicates the types of join to be performed, shows whether the requests in a transaction would be dispatched in parallel, and so on.

When you perform an EXPLAIN against any SQL request, that request is parsed and optimized, and the plan generated by the Optimizer is returned to the requestor in the form of a text file that explains the steps of the plan for the request. The EXPLAIN report reproduces the execution strategy determined by the Optimizer, but does not explain why it makes the choices it does.

EXPLAIN helps you to evaluate complex queries and to develop alternative, more efficient, processing strategies.

Only the first 255 characters of a conditional expression are displayed in an EXPLAIN. The entire conditional expression is enclosed in APOSTROPHE characters.

Although the times reported in the EXPLAIN output are presented in units of seconds, they are actually arbitrary units of time. These numbers are valuable because they permit you to compare alternate coding formulations of the same query with respect to relative performance, but do not correlate with clock time. Times and row estimates are not reported for some steps.

Save the EXPLAIN results for future reference.

## Substitute Characters in Object Names and Literals Returned in an EXPLAIN Report

If an object name or literal returned in an EXPLAIN report includes a character not in the session character set, or is otherwise unprintable, the name or literal is shown as a UNICODE delimited identifier or literal.

Each non-representable character is escaped to its hexadecimal equivalent.

The system uses BACKSLASH (U+005C), if available, as the escape character. If BACKSLASH is not available:

- If either YEN SIGN (U+00A5) or WON SIGN (U+20A9) replaces BACKSLASH in the session character set, the system uses the replacement as the escape character.
- If none of the preceding characters is available, the system uses NUMBER SIGN (U+0023).

Object names that are translatable to the session character set, but are not lexically distinguishable as names are enclosed in double quotation marks.

For information on object naming, see *Teradata Vantage™ SQL Fundamentals*, B035-1141.

## Standard Form of Display for EXPLAIN

The standard output of an EXPLAIN request modifier displays character constant strings as follows:

Type of Constant	Format
Printable single byte character.	Teradata Latin.
Anything else.	Internal Teradata hexadecimal.

## EXPLAIN Processes SQL Requests Only

You can modify any valid Teradata SQL request with EXPLAIN.

You cannot EXPLAIN a USING request modifier, a WITH RECURSIVE view modifier, or another EXPLAIN modifier. See [USING Request Modifier](#) and [WITH Modifier](#). You cannot explain individual functions, procedures, or methods.

## Teradata Visual Explain Utility

The Teradata Visual Explain tool provides a graphic display of Optimizer plans and also permits you to compare the plans for queries that return the same result. This feature can simplify the interpretation of EXPLAIN reports.

For more information about Teradata Visual Explain, see *Teradata® Visual Explain User Guide*, B035-2504. Related supporting information appears in *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

## EXPLAIN and Embedded SQL

Use EXPLAIN only with data returning requests. EXPLAIN returns multiple rows of a single column whose data type is VARCHAR(80). Because it causes multiple rows to be returned, EXPLAIN can only be performed with a cursor.

For static execution of EXPLAIN, see *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148, substituting an EXPLAIN-modified SQL request for the *query\_expression* clause shown there.

Do not perform EXPLAIN request modifiers dynamically because the result is unpredictable.

## EXPLAIN and Procedures

You cannot EXPLAIN procedures.

For example, if you compile the procedure `update_orders` and perform the following EXPLAIN, an error occurs because `update_orders` is not a valid SQL statement.

```
EXPLAIN update_orders;
```

You should use EXPLAIN while building your procedures. You must extract the SQL text from the procedure body and EXPLAIN each of the individual statements. This can require modification of the

procedure SQL text in some cases. For example, you might have to remove an INTO clause or add a USING request modifier to represent procedure variables and parameters.

### Effect of Request Cache Peeking on EXPLAIN Reports

If you specify USING data, or if you specify a DATE or CURRENT\_DATE built-in function in a request, or both, the system can invoke Request Cache peeking. See *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142. In this case, the EXPLAIN text indicates the peeked literal values.

If you do not specify USING data, or if the USING variables, DATE or CURRENT\_DATE values, or both are not peeked, then there is no impact on either the generated plan or the generated EXPLAIN text.

Note that parameterized requests specified without a USING request modifier, but using either CLlv2 data parcel flavor 3 (Data) or CLlv2 parcel flavor 71 (DataInfo), cannot be explained using any of the following request modifiers or statements:

- EXPLAIN
- DUMP EXPLAIN
- INSERT EXPLAIN

For details about data parcel formats, see *Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems*, B035-2417 or *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418.

### Using EXPLAIN to Determine the Database Objects that a View Accesses

Explaining a request does not necessarily report the names of all the underlying database objects accessed by that request, but it does provide the names of all the base tables accessed.

To determine the objects that a particular view accesses, including any nested views, use the SHOW statement. See “SHOW” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

For example, the following request reports the create text and containing databases for all underlying tables and views accessed by *view\_name*.

```
SHOW QUALIFIED SELECT *
FROM view_name;
```

On the other hand, the following request reports only the underlying tables accessed by *view\_name*, and not any nested views that are accessed.

```
EXPLAIN SELECT *
FROM view_name;
```

### 2PC Session Mode

If you specify an EXPLAIN modifier with a request in 2PC session mode, and that request is followed by a 2PC function, then the SQL portion of the request is explained, but the 2PC function is not.

For example, if the following multistatement request were submitted, the INSERT and SELECT would be explained; the VOTE would not be explained. The VOTE is, however, sent to the AMPs:

```
EXPLAIN INSERT tab1(1,2)
;SELECT *
  FROM tab1
;VOTE
```

VOTE is specified by the system administrator, not by users submitting requests in 2PC mode.

## Usage Notes

### EXPLAIN and the USING Request Modifier

If you specify both an EXPLAIN request modifier and a USING request modifier for the same request, the EXPLAIN modifier must precede the USING modifier. See [USING Request Modifier](#). A dynamic plan cannot be displayed for requests with a USING request modifier.

### Capturing EXPLAIN Text Output in XML Format

The IN XML option returns the EXPLAIN text as XML text rather than as plain English text. See *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142 for the XML plan attributes and the XML schema used by the XML output produced by the IN XML option.

The EXPLAIN IN XML request modifier can also specify that the output be returned in one of two possible formats:

- As XML text with any associated DDL text. You specify this with the IN XML option.
- As XML text, but without any associated DDL text. You specify this with the IN XML NODDLTEXT option.

A complementary option for the BEGIN QUERY LOGGING and REPLACE QUERY LOGGING statements optionally creates a compact XML version of the information that can be stored in a DBQL table. See “BEGIN QUERY LOGGING” and “REPLACE QUERY LOGGING” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

### EXPLAIN Report Overview

EXPLAIN provides a summary of the access and join plans generated by the Optimizer for the query SQL requests. The report details which indexes would be used to process the request, identifies any intermediate spools that would be generated, indicates the types of join to be performed, shows whether the requests in a transaction would be dispatched in parallel, and so on.

When you perform an EXPLAIN against any SQL request, that request is parsed and optimized, and the plan generated by the Optimizer is returned to the requestor in the form of a text file that explains the steps of the plan for the request. The EXPLAIN report reproduces the execution strategy determined by the Optimizer, but does not explain why it makes the choices it does.

EXPLAIN helps you to evaluate complex queries and to develop alternative, more efficient, processing strategies.

Only the first 255 characters of a conditional expression are displayed in an EXPLAIN. The entire conditional expression is enclosed in APOSTROPHE characters.

Although the times reported in the EXPLAIN output are presented in units of seconds, they are actually arbitrary units of time. These numbers are valuable because they permit you to compare alternate coding formulations of the same query with respect to relative performance, but do not correlate with clock time. Times and row estimates are not reported for some steps.

Save the EXPLAIN results for future reference.

## Substitute Characters in Object Names and Literals Returned in an EXPLAIN Report

If an object name or literal returned in an EXPLAIN report includes a character not in the the session character set, or is otherwise unprintable, the name or literal is shown as a UNICODE delimited identifier or literal.

Each non-representable character is escaped to its hexadecimal equivalent.

The system uses BACKSLASH (U+005C), if available, as the escape character. If BACKSLASH is not available:

- If either YEN SIGN (U+00A5) or WON SIGN (U+20A9) replaces BACKSLASH in the session character set, the system uses the replacement as the escape character.
- If none of the preceding characters is available, the system uses NUMBER SIGN (U+0023).

Object names that are translatable to the session character set, but are not lexically distinguishable as names are enclosed in double quotation marks.

For information on object naming, see *Teradata Vantage™ SQL Fundamentals*, B035-1141.

## Standard Form of Display for EXPLAIN

The standard output of an EXPLAIN request modifier displays character constant strings as follows:

Type of Constant	Format
Printable single byte character.	Teradata Latin.
Anything else.	Internal Teradata hexadecimal.

## EXPLAIN Processes SQL Requests Only

You can modify any valid Teradata SQL request with EXPLAIN.

You cannot EXPLAIN a USING request modifier, a WITH RECURSIVE view modifier, or another EXPLAIN modifier. See [USING Request Modifier](#) and [WITH Modifier](#). You cannot explain individual functions, procedures, or methods.

## Teradata Visual Explain Utility

The Teradata Visual Explain tool provides a graphic display of Optimizer plans and also permits you to compare the plans for queries that return the same result. This feature can simplify the interpretation of EXPLAIN reports.

For more information about Teradata Visual Explain, see *Teradata® Visual Explain User Guide*, B035-2504. Related supporting information appears in *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

## EXPLAIN and Embedded SQL

Use EXPLAIN only with data returning requests. EXPLAIN returns multiple rows of a single column whose data type is VARCHAR(80). Because it causes multiple rows to be returned, EXPLAIN can only be performed with a cursor.

For static execution of EXPLAIN, see *Teradata Vantage™ SQL Stored Procedures and Embedded SQL*, B035-1148, substituting an EXPLAIN-modified SQL request for the *query\_expression* clause shown there.

Do not perform EXPLAIN request modifiers dynamically because the result is unpredictable.

## EXPLAIN and Procedures

You cannot EXPLAIN procedures.

For example, if you compile the procedure `update_orders` and perform the following EXPLAIN, an error occurs because `update_orders` is not a valid SQL statement.

```
EXPLAIN update_orders;
```

You should use EXPLAIN while building your procedures. You must extract the SQL text from the procedure body and EXPLAIN each of the individual statements. This can require modification of the procedure SQL text in some cases. For example, you might have to remove an INTO clause or add a USING request modifier to represent procedure variables and parameters.

## Effect of Request Cache Peeking on EXPLAIN Reports

If you specify USING data, or if you specify a DATE or CURRENT\_DATE built-in function in a request, or both, the system can invoke Request Cache peeking. See *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142. In this case, the EXPLAIN text indicates the peeked literal values.

If you do not specify USING data, or if the USING variables, DATE or CURRENT\_DATE values, or both are not peeked, then there is no impact on either the generated plan or the generated EXPLAIN text.

Note that parameterized requests specified without a USING request modifier, but using either CLlv2 data parcel flavor 3 (Data) or CLlv2 parcel flavor 71 (DataInfo), cannot be explained using any of the following request modifiers or statements:

- EXPLAIN
- DUMP EXPLAIN
- INSERT EXPLAIN

For details about data parcel formats, see *Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems*, B035-2417 or *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418.

## Using EXPLAIN to Determine the Database Objects that a View Accesses

Explaining a request does not necessarily report the names of all the underlying database objects accessed by that request, but it does provide the names of all the base tables accessed.

To determine the objects that a particular view accesses, including any nested views, use the SHOW statement. See “SHOW” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

For example, the following request reports the create text and containing databases for all underlying tables and views accessed by *view\_name*.

```
SHOW QUALIFIED SELECT *
FROM view_name;
```

On the other hand, the following request reports only the underlying tables accessed by *view\_name*, and not any nested views that are accessed.

```
EXPLAIN SELECT *
FROM view_name;
```



## 2PC Session Mode

If you specify an EXPLAIN modifier with a request in 2PC session mode, and that request is followed by a 2PC function, then the SQL portion of the request is explained, but the 2PC function is not.

For example, if the following multistatement request were submitted, the INSERT and SELECT would be explained; the VOTE would not be explained. The VOTE is, however, sent to the AMPs:

```
EXPLAIN INSERT tab1(1,2)
;SELECT *
  FROM tab1
;VOTE
```

VOTE is specified by the system administrator, not by users submitting requests in 2PC mode.

## INITIATE INDEX ANALYSIS

### Purpose

Analyzes a query workload and generates a set of recommended secondary and single-table join indexes for optimizing its processing.

For more information about QCDs and index analysis, see:

- [COLLECT DEMOGRAPHICS](#)
- [COLLECT STATISTICS \(QCD Form\)](#)
- [INITIATE PARTITION ANALYSIS](#)
- [RESTART INDEX ANALYSIS](#)
- [INSERT EXPLAIN](#)
- “CREATE TABLE (Index Definition Clause)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ - Database Design*, B035-1094
- *Teradata® Index Wizard User Guide*, B035-2506

### Required Privileges

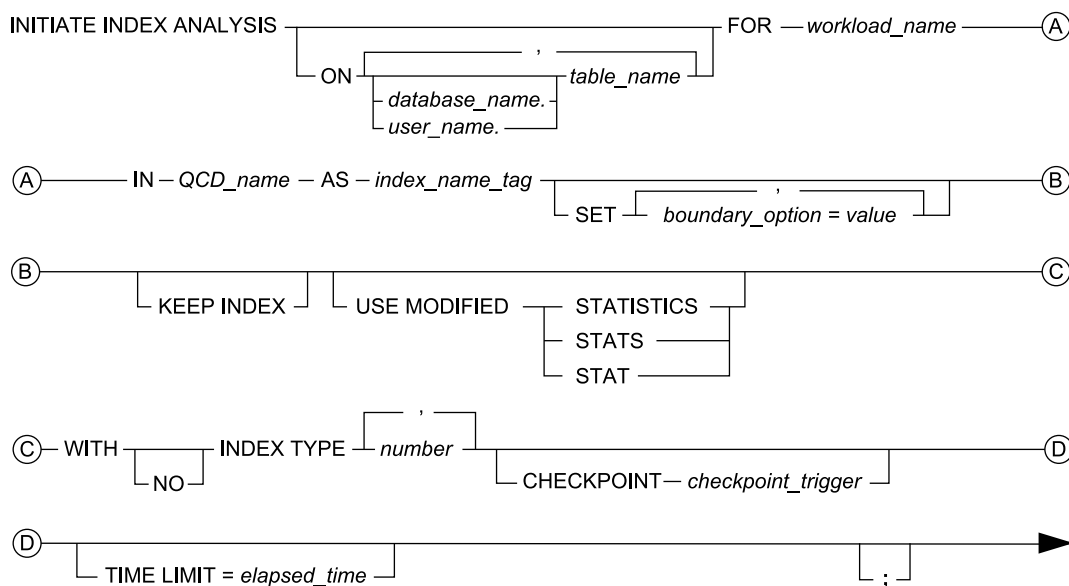
The privileges required to submit an INITIATE INDEX ANALYSIS request are the following:

Privilege Required	Table Set in Database <i>QCD_name</i>
<ul style="list-style-type: none"> <li>• INSERT</li> <li>and</li> <li>• SELECT</li> </ul>	IndexRecommendations
INSERT	<ul style="list-style-type: none"> <li>• AnalysisStmts</li> </ul>

Privilege Required	Table Set in Database <i>QCD_name</i>
	<ul style="list-style-type: none"> <li>• IndexMaintenance</li> <li>• IndexColumns</li> <li>• JoinIndexColumns</li> </ul>

If you specify a checkpoint trigger, you must have both INSERT and DELETE privileges on the AnalysisLog table in *QCD\_name*.

## Syntax



## Syntax Elements

**database\_name**

**user\_name**

Containing database or user for *table\_name* if something other than the current database or user.

**table\_name**

Table set to be analyzed for index recommendations.

If no table is specified, then all tables referenced in the specified *workload\_name* are analyzed for index recommendations.

This option permits you to implicitly instruct **INITIATE INDEX ANALYSIS** not to recommend indexes on certain tables.

**workload\_name**

Name of the workload to which the queries to be analyzed belong.

**QCD\_name**

QCD workload database in which *workload\_name* exists.

INITIATE INDEX ANALYSIS stores the index recommendations for the specified workload in this QCD database.

***index\_name\_tag***

Name to be assigned to the index recommendations within *QCD\_name*.

***boundary\_option = value***

Clause that sets upper bounds on the specified option.

The following options and maximum bounds are valid specifications.

- CHANGERATE
- COLUMNSPERINDEX
- COLUMNSPERJOININDEX
- INDEXMAINTMODE
- INDEXESPERTABLE
- SEARCHSPACE

For the definitions and maximum bounds for these options, see [Boundary Options](#).

**KEEP INDEX**

Index recommendations are not to contain any DROP INDEX or DROP STATISTICS recommendations.

The default is to recommend DROP INDEX and DROP STATISTICS recommendations when the analysis indicates their usefulness.

**USE MODIFIED STATISTICS**

Perform the index analysis with the user-modified statistics stored in the ModifiedStats column of QCD table TableStatistics rather than the statistics gathered with COLLECT STATISTICS (QCD Form).

If the ModifiedStats column is not populated, then the statistics stored in StatisticsInfo are used instead and the result is the same as if you had not specified USE MODIFIED STATISTICS.

You can perform “what if” analyses on the data to examine how the Optimizer processes various scenarios.

**WITH INDEX TYPE *number***

**WITH NO INDEX TYPE *number***

Types of secondary and single-table join indexes that are to be considered for analysis by the Teradata Index Wizard.

- If you specify the clause as WITH INDEX TYPE *number*, the Teradata Index Wizard includes the specified set of index types in its analysis.
- If you specify the clause as WITH NO INDEX TYPE *number*, the Teradata Index Wizard excludes the specified set of index types from its analysis.
- If you do not specify this clause, the Teradata Index Wizard includes all valid index types in its analysis by default.

The *number* code is an identifier for the IndexType column values stored in the QCD IndexRecommendations table.

Following are the values you can specify for *number* and the corresponding descriptions.

Option	Description
1	Unique secondary index (USI).
2	Value-ordered secondary index (VOSI).
3	Hash-ordered secondary index (HOSI).
4	Nonunique secondary index (NUSI).
5	Simple join index (JI).
6	Aggregate join index (AJI).

The Teradata Index Wizard includes only *single-table* join indexes in the analyses for codes 5 and 6 and hash indexes are not supported.

See *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142 for more information about the QCD and the Teradata Index Wizard. Also see *Teradata® Index Wizard User Guide*, B035-2506 for information about using the Teradata Index Wizard client utility.

#### **CHECKPOINT *checkpoint\_trigger***

Number of queries after which to take a checkpoint snapshot.

The value for *checkpoint\_trigger* must be a positive integer.

- If *checkpoint\_trigger* is 0, the system ignores the CHECKPOINT clause.
- If *checkpoint\_trigger* is > the total number of queries to be analyzed, the system does not take a checkpoint snapshot.

See [Example: Using a CHECKPOINT](#).

#### **TIME LIMIT = *elapsed\_time***

The maximum elapsed time in minutes allowed for this index analysis to complete.

The default value is no time limit.

The permitted range of specified values is from 1 to 2880 minutes, for a maximum of 48 hours.

You must specify the value for *elapsed\_time* as an integer.

If the index analysis does not complete before reaching the specified time limit, the system stops the task and retains the best recommendations found up to the point when the time limit expired.

The time limit that you specify is only an approximation because the ongoing index analysis task only checks periodically as to whether the specified time limit has been exceeded.

Also see [Example: Setting a TIME LIMIT on an Index Analysis](#).

## ANSI Compliance

INITIATE INDEX ANALYSIS is a Teradata extension to the ANSI SQL:2011 standard.

## Usage Notes

### Invocation

Normally invoked using the Teradata Index Wizard utility.

### Boundary Options

The following table lists the valid boundary option values and their definitions.

Option	Definition
CHANGERATE	<p>Every column appearing in an UPDATE SET clause is automatically assigned a change rating ranging from 0 to 9, representing the frequency at which it is updated. For more information about change ratings, see .</p> <p>The system assigns columns with a very high update frequency a change rating of 9, while columns that are never updated are assigned a change rating of 0.</p> <p>Columns with a change rating greater than this user-specified ChangeRate parameter are not considered during index analysis for potential indexes. In other words, the ChangeRate value is a threshold for determining whether to use columns for an index analysis or not.</p> <p>If the specified ChangeRate parameter is 9, the change rating system is not used to disqualify columns from index analysis.</p> <p>The default is 5.</p> <p>The valid range is 0 to 9, where 0 means the column is not updated and 9 means the column is highly volatile with respect to the specified workload.</p>
COLUMNSPERINDEX	<p>Maximum number of composite secondary index columns to be considered for index recommendation by the Index Wizard.</p> <p>The default is 4.</p> <p>The valid range is 1 to 64.</p>
COLUMNSPERJOININDEX	<p>Maximum number of referenced columns in a single-table join index to be considered for index analysis by the Index Wizard.</p> <p>The default is 8.</p> <p>The valid range is 1 to 16.</p>
INDEXMAINTMODE	<p>Controls how estimated index maintenance costs are used during analysis. The following values are supported.</p> <ul style="list-style-type: none"> <li>0 Maintenance costs are not estimated. The feature is disabled.</li> <li>1</li> </ul>

Option	Definition
	<p>Maintenance costs are estimated only as supplemental information for the final index recommendations. This is the default.</p> <ul style="list-style-type: none"> <li>• 2 Maintenance costs are used to evaluate and choose between candidate indexes and are also included with the final recommendations.</li> </ul>
INDEXESPERTABLE	<p>Maximum number of new indexes, including secondary and join indexes, that can be recommended per table. The default is 16. The system limit is 32. Each value-ordered secondary index reduces the number of consecutive indexes by 2. The valid range is 1 to 32.</p>
SEARCHSPACE	<p>Maximum size of the search space to be allowed for evaluating index candidates. The higher the number you specify, the more candidate indexes the system can evaluate. The number specified for SEARCHSPACE corresponds to an internal variable that is used to control the size of the search space evaluated by the index search algorithm. Keep in mind that the larger the number you specify, the longer the evaluation time. To shorten your evaluation times, change the specification for SearchSpace to a lower number. The default is 256. The valid range is 128 to 512.</p>

## Rules for Performing INITIATE INDEX ANALYSIS

The following set of rules applies to INITIATE INDEX ANALYSIS:

- The analysis accepts any of the following DML statements as valid workload components:

◦ DELETE ◦ INSERT	◦ MERGE ◦ SELECT	◦ UPDATE
----------------------	---------------------	----------

- The specified QCD database must exist in the system. Otherwise, the request aborts and the system returns an error.
- The specified workload must exist in the specified *QCD\_name* database. Otherwise, the request aborts and the system returns an error.
- If you specify a table list, then only that list of tables is considered for index analysis.  
If the workload does not contain any tables that match at least one table in the table list, then an error is reported. Otherwise, only the matching list of tables is considered for index analysis.
- A maximum of 1,024 tables can be analyzed per workload.
- No error is reported if the same table is specified more than once for index analysis.

- The index analysis parameters must not be repeated.  
If they are repeated, or if a value is out of range for a parameter, then the request aborts and the system returns an error.
- If the session is logged off or restarted while INITIATE INDEX ANALYSIS is being performed and no checkpoint is specified, then no index recommendations are saved.  
You must perform the INITIATE INDEX ANALYSIS request again in the restarted session to create the index recommendations.

## Sequence of Events for an Index Analysis

The following process indicates the general phases of an index analysis:

1. The analysis begins when an INITIATE INDEX ANALYSIS request is submitted.
2. The database query analysis tool analyzes the submitted workload for indexing opportunities.
3. The index recommendations are written to the IndexRecommendations, IndexColumns, and JoinIndexColumns tables of the specified QCD database.
4. The recommendations are retrieved for evaluation using a set of SQL SELECT requests defined within a number of macros provided within each QCD.

For example, the following SELECT request is the core of the Recommendations macro.

```
SELECT TableName, DatabaseName, IndexTypeText, SpaceEstimate,
       IndexDDL
FROM   IndexRecommendations
WHERE  WorkloadID = :WorkloadID;
```

You can find a list of the index recommendation query macros in *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

## Retrieving a Completed Index Analysis

After the index analysis completes successfully, you can retrieve the index recommendations from the IndexRecommendations and IndexColumns tables in the appropriate QCD (see *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142 for more information about the QCD).

## Using the WITH INDEX TYPE or WITH NO INDEX TYPE Option

You can use this option to either include or exclude various types of secondary and join indexes from consideration for the analysis to be performed on the QCD data by the Teradata Index Wizard.

You can include for consideration, or exclude from consideration, by the analysis any of the following types of indexes.

- Unique secondary (USI)

- Nonunique secondary (NUSI)
- Hash-ordered secondary
- Value-ordered secondary
- Single-table simple join
- Single-table aggregate join

The default, indicated by not specifying this clause in your INITIATE INDEX ANALYSIS request, is to include *all* valid index types for consideration by the analysis. Note that the list of valid index types for index analysis does *not* include multitable join indexes or hash indexes.

For example, you might join indexes from consideration by the analysis, which you could consider to be undesirable for workloads that invoke utilities such as Teradata Archive/Recovery or MultiLoad, neither of which can be run against tables that have join indexes defined on them. For more information about restrictions and interactions with tables on which join indexes are defined, see “CREATE JOIN INDEX” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - Database Design*, B035-1094.

In this case, you can substitute FastLoad for MultiLoad to batch load rows into an empty staging table and then use either an INSERT ... SELECT request with error logging or a MERGE request with error logging to move the rows into their target table if the scenario permits. For details, see [INSERT/INSERT ... SELECT](#), [MERGE](#), and CREATE ERROR TABLE” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

## Using the CHECKPOINT Option

For long index analyses or cases where the workload has many SQL requests that must be analyzed, you should always specify a checkpoint using the CHECKPOINT option.

The checkpoint trigger value indicates the frequency with which a checkpoint is logged in the AnalysisLog table. A checkpoint is taken after every *n* request occurrences, where *n* is the checkpoint trigger value, and the interim index analysis information is logged.

Note that you can specify both the CHECKPOINT and the TIME LIMIT options for an INITIATE INDEX ANALYSIS request.

For information about how to restart a checkpointed index analysis, see [RESTART INDEX ANALYSIS](#).

## CHECKPOINT and TIME LIMIT Options

The intent of the CHECKPOINT option and the related RESTART INDEX ANALYSIS statement is to establish safeguards for long running INITIATE INDEX ANALYSIS requests for cases where you want to ensure that you do not have to restart the analysis from the beginning if the request unexpectedly aborts. See [RESTART INDEX ANALYSIS](#).

The TIMELIMIT option permits you to place a time limit on the index analysis. Note that you must be willing to accept the best recommendations found up to that point if you specify this option.



Depending on the size and complexity of the defined workload, the operations undertaken by an `INITIATE INDEX ANALYSIS` request can be very resource-intensive. The process of repeatedly calling the Optimizer with different sets of candidate indexes incurs heavy CPU usage in the Parsing Engine. You should not use this statement on a production system.

[INITIATE PARTITION ANALYSIS](#) and `INITIATE INDEX ANALYSIS` are related statements in the sense that both first examine a workload and then make recommendations to improve performance. You can submit one or both of these statements in either order for a given workload.

## Evaluation of the ChangeRate Variable by INITIATE INDEX ANALYSIS

The following table indicates the method `INITIATE INDEX ANALYSIS` uses to evaluate column change rate values for index recommendation. Only those columns with a change rate rank  $\leq$  the specified change rate rank are considered for recommendation as an index or composite index component. If no change rate is specified, the default value is 5.

Change Rate Rank	Description	Percentage of Updates on Column
0	No updates on the column.	0
1	1 update per 1,000,000 selects on the table.	.0001
2	1 update per 100,000 selects on the table.	.001
3	1 update per 10,000 selects on the table.	.01
4	1 update per 1,000 selects on the table.	.1
5	1 update per 100 selects on the table.	1
6	5 updates per 100 selects on the table.	5
7	10 updates per 100 selects on the table.	10
8	25 updates per 100 selects on the table.	25
9	More than 25 updates per 100 selects on the table.	>25

## INITIATE INDEX ANALYSIS Not Supported From Macros

You cannot specify an `INITIATE INDEX ANALYSIS` request from a macro. If you execute a macro that contains an `INITIATE INDEX ANALYSIS` request, the system returns an error.

## Examples

### Example: Index Analysis

Assume that the QCD named *MyQCD* exists on the system. The following INITIATE INDEX ANALYSIS request performs successfully:

```
INITIATE INDEX ANALYSIS ON table_1
FOR MyWorkload
IN MyQCD
AS table_1Index;
```

If *MyQCD* does not exist in the system, the same request fails.

```
INITIATE INDEX ANALYSIS ON table_1
FROM MyWorkload
IN MyQCD
AS table_1Index;

*** Failure 3802 Database 'MyQCD' does not exist.
```

### Example: Index Analysis and Privileges

Assume that you have the INSERT privilege on the *IndexRecommendations* and *IndexColumns* tables or on database *QCD\_name*. The following INITIATE INDEX ANALYSIS request performs successfully.

```
INITIATE INDEX ANALYSIS ON table_1
FOR MyWorkload
IN MyQCD
AS table_1Index;
```

If you do not have INSERT privilege on *MyQCD*, the same request fails.

```
INITIATE INDEX ANALYSIS ON table_1
FOR MyWorkload
IN MyQCD
AS table_1Index;

*** Failure 3523 The user does not have INSERT access to
MyQCD.IndexRecommendations.
```

## Example: Index Analysis and Table List

When you specify a table list, then only that list of tables is considered for index analysis. If the workload does not have any tables matching at least one table in the list, then an error is reported. Otherwise, only the matching list of tables is considered for selecting the indexes.

Assume that *MyWorkload* describes the workload for tables *table\_1*, *table\_2* and *table\_3*. The following INITIATE INDEX ANALYSIS request performs successfully and the recommendations are considered only for *table\_1* and *table\_2* because *table\_3* is not specified in the table list.

```
INITIATE INDEX ANALYSIS ON table_1, table_2
FOR MyWorkload
INTO MyQCD
AS table_1Index;
```

Suppose *MyWorkload* describes the workload for *table\_1* and *table\_2* only. The following INITIATE INDEX ANALYSIS request fails because *table\_3* and *table\_4* are not defined in the workload.

```
INITIATE INDEX ANALYSIS ON table_3, table_4
FOR MyWorkload
IN MyQCD
AS table_1Index;

*** Failure 5638 No match found for specified tables in the workload
'MyWorkload'.
```

## Example: Index Analysis and Duplicate Table Specifications

No error is reported if the same table is specified more than once for index selection. In the following example, *table\_1* is specified twice.

```
INITIATE INDEX ANALYSIS ON table_1, table_2, table_1
FOR MyWorkload
IN MyQCD
AS table_1Index;
```

## Example: Index Analysis and Repeated Parameters

If index analysis parameters are specified, they must not be repeated. If the parameters are repeated, an error is reported.

In the following example, the only index analysis parameter specified, *IndexesPerTable*, is specified once.

```
INITIATE INDEX ANALYSIS ON table_1
FOR MyWorkload
IN MyQCD
AS table_1Index
SET IndexesPerTable = 2;
```

In the following example, the index analysis parameter `IndexesPerTable` is specified twice, so an error is returned.

```
INITIATE INDEX ANALYSIS ON table_1
FOR MyWorkload
IN MyQCD
AS table_1Index
SET IndexesPerTable=2, ChangeRate=4, IndexesPerTable=4;

*** Failure 3706 Syntax error: Multiple "IndexesPerTable" options.
```

Note that the error is not caused by the discrepancy between the specified boundary conditions for the `IndexesPerTable` parameter: the same error would be reported if the boundary conditions matched in both specifications.

## Example: Using a CHECKPOINT

You should specify the `CHECKPOINT` option for long operations and for workloads containing many SQL requests that must be analyzed. The *checkpoint\_trigger* value indicates the frequency with which a checkpoint is taken. After every *checkpoint\_trigger* number of SQL requests, a checkpoint is taken and completed *IndexRecommendation* information is saved in the *AnalysisLog* table in the QCD.

```
INITIATE INDEX ANALYSIS ON table_1
FOR MyWorkload
IN MyQCD
AS table_1Index
SET IndexesPerTable = 2
KEEP INDEX
CHECKPOINT 10;
```

## Example: Setting a TIME LIMIT on an Index Analysis

This example places a 10 minute time limit on the index analysis:

```
INITIATE INDEX ANALYSIS
FOR myworkload
IN myqcd
```

```
AS myfastidxanalysis
TIME LIMIT = 10;
```

### Example: Specifying a CHECKPOINT and a TIME LIMIT

This example takes a checkpoint for every request in the workload and places a 1 minute time limit on the index analysis.

```
INITIATE INDEX ANALYSIS
FOR w11
IN myqcd
AS w11_analysis1
CHECKPOINT 1
TIME LIMIT = 1;
```

### Example: Include NUSI and Simple Join Indexes Only in the Analysis

This example considers only NUSIs and simple single-table join indexes for the index analysis.

```
INITIATE INDEX ANALYSIS ON tab1
FOR MyWorkload
IN MyQCD AS tab1Index
WITH INDEX TYPE 4, 5;
```

### Example: Exclude All Join Indexes From The Analysis

This example excludes all types of join indexes from the index analysis.

```
INITIATE INDEX ANALYSIS ON tab1
FOR MyWorkload
IN MyQCD AS tab1Index
WITH NO INDEX TYPE 5, 6;
```

### Example: Include All Valid Index Types in the Analysis

This example considers all valid index types for the index analysis by default.

```
INITIATE INDEX ANALYSIS ON tab1
FOR MyWorkload
IN MyQCD AS tab1Index;
```

# INITIATE PARTITION ANALYSIS

## Purpose

Analyzes a given workload to make recommendations for partitioning expressions.

INITIATE PARTITION ANALYSIS also considers the potential performance benefits of partitioning one or more tables in a given workload. The resulting recommendations, if any, are stored in the *PartitionRecommendations* table in the specified Query Capture Database.

For more information about QCDs, partitioning expression analysis, and partitioned tables and join indexes, see:

- “CREATE TABLE (Index Definition Clause)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184
- [INITIATE INDEX ANALYSIS](#)
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ - Database Design*, B035-1094
- *Teradata® Index Wizard User Guide*, B035-2506

## Required Privileges

You must have the following privileges on tables in database *QCD\_name*:

- INSERT and SELECT on *PartitionRecommendations*
- INSERT on *RangePartExpr*

## Syntax

```

INITIATE PARTITION ANALYSIS (ON (table_name) | (database_name.table_name) | (user_name.table_name)) FOR workload_name (A)
(A) IN QCD_name AS result_name_tag [TIME LIMIT = elapsed_time] ;
  
```

## Syntax Elements

**ON *table\_name***

**ON *database\_name.table\_name***

**ON *user\_name.table\_name***

Set of comma-separated table names to be considered for partitioning recommendations.

The list can include any mixture of tables, including those that already have partitioning expressions.

If you do not specify a table name set, the system analyzes all tables referenced in the workload that do not currently have partitioning by default.

The maximum number of tables that can be analyzed in a given workload is 1,024.

**FOR *workload\_name***

The name of the workload whose queries are to be analyzed.

**IN *QCD\_name***

The name of the Query Capture Database containing the specified workload and in which the resulting row partitioning recommendations are to be stored.

***result\_name\_tag***

A unique name that you assign to the partitioning expression recommendation set.

The system stores the results of the partition analysis under this name in *QCD\_name*.

**TIME LIMIT = *elapsed\_time***

The maximum elapsed time in minutes that you want this partitioning expression analysis to take.

The default value is no time limit.

The permitted range of specified values is from 1 to 2880 minutes, for a maximum of 48 hours.

If the partitioning expression analysis does not complete before reaching the specified time limit, the system stops the task and retains the best recommendations found up to the point when the time limit expired.

You must specify the value for *elapsed\_time* as an integer value.

Note that the time limit that you specify is only an approximation because the ongoing partitioning expression analysis task only checks to see if the time has been exceeded periodically.

## ANSI Compatibility

INITIATE PARTITION ANALYSIS is a Teradata extension to the ANSI SQL:2011 standard.

## Invocation

Normally invoked using the Teradata Index Wizard utility.

## About INITIATE PARTITION ANALYSIS

INITIATE PARTITION ANALYSIS does not recommend the removal of any currently defined partitioning expressions. It does consider and can recommend changing an existing partitioning expression if you include a row-partitioned table in the specified *table\_name* list.

The set of partitioning expression recommendations, if any, is the one that minimizes the total cost of *all* requests in the specified workload. Recommendations made by INITIATE PARTITION ANALYSIS typically involve a simple RANGE\_N function on an INTEGER or DATE column. Because of the inherent nature of partitioning, it is possible that by following the recommendations INITIATE PARTITION ANALYSIS provides, the performance of one or more *individual* requests can be degraded, even though the *total* workload cost is reduced.

The partitioning expression analysis process stores its final recommendations in the QCD table *PartitionRecommendations*. For the definitions of *PartitionRecommendations* and its columns, see *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142. INITIATE PARTITION

ANALYSIS stores a separate row in this table for each workload request that is affected by the recommendation.

Depending on the size and complexity of the defined workload, the operations undertaken by an INITIATE PARTITION ANALYSIS request can be very resource intensive. This is because the process of repeatedly calling the Optimizer with different sets of candidate partitioning expressions incurs heavy CPU usage in the Parsing Engine. Because of this, you should run your partitioning expression analyses on a test system rather than a production system.

### Partition Analysis and Multilevel Partitioning

INITIATE PARTITION ANALYSIS does not recommend multilevel partitioning or column partitioning. However, for analyses that report more than one recommendation for a partitioning expression, you might consider experimenting with combining some or all of the recommendations to create a multilevel partitioning (with or without column partitioning).

### INITIATE PARTITION ANALYSIS Not Supported From Macros

You cannot specify an INITIATE PARTITION ANALYSIS request from a macro. If you execute a macro that contains an INITIATE PARTITION ANALYSIS request, Teradata Database aborts the request and returns an error.

### Example: Partition Analysis on All Tables

The following example considers partitioning expression recommendations for all tables in a workload; therefore, no ON clause specified:

```
INITIATE PARTITION ANALYSIS
FOR myworkload
IN myqcd
AS myppirecs;
```

Now you can examine the recommended partitioning expressions from this partition analysis using the following SELECT request:

```
SELECT tablename, expressiontext
FROM qcd.partitionrecommendations
WHERE resultnametag = 'myppirecs';
```

### Example: Partition Analysis on Specified Tables

The following example considers partitioning expression recommendations for specific tables within a workload; therefore, an ON clause is specified:

```
INITIATE PARTITION ANALYSIS
ON tab1, tab2
FOR myworkload
```



```
IN myqcd
AS myppitabrecs;
```

### Example: Partition Analysis with a Time Limit

The following example places a 5 minute time limit on the partitioning expression analysis:

```
INITIATE PARTITION ANALYSIS
FOR myworkload
IN myqcd
AS myfastanalysis
TIME LIMIT = 5;
```

## INSERT EXPLAIN

### Purpose

Captures the Optimizer white tree information for a query and writes the information to a user-defined query capture database, optionally as an XML document.

For more information about index analysis, see:

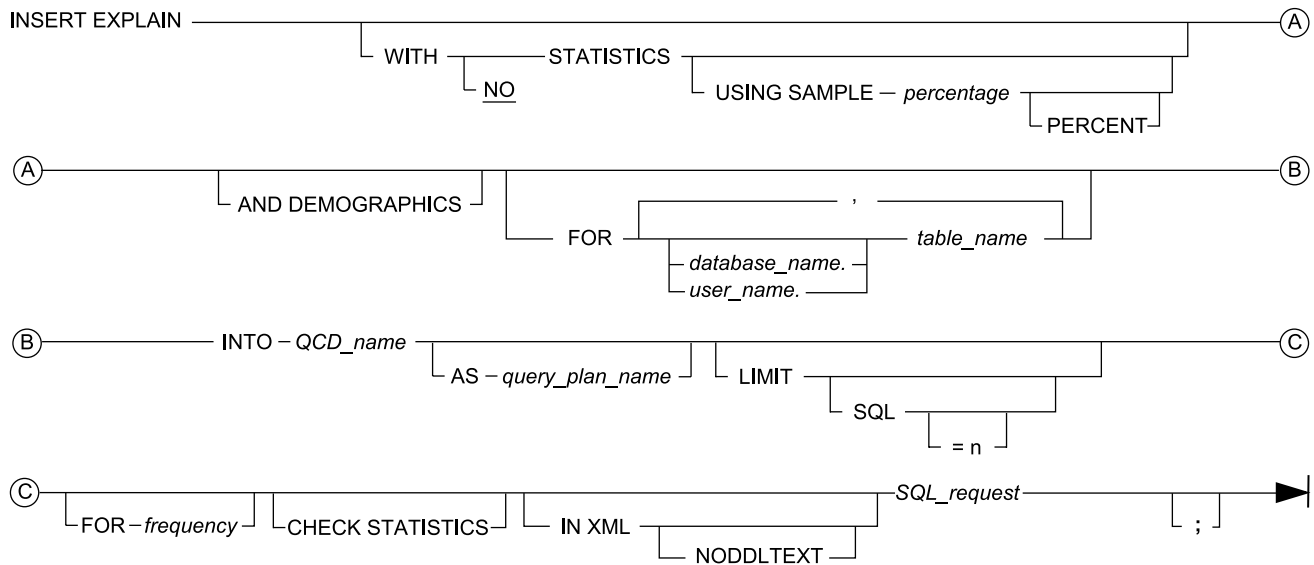
- “BEGIN QUERY LOGGING” and “REPLACE QUERY LOGGING” in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.
- [COLLECT DEMOGRAPHICS](#)
- [COLLECT STATISTICS \(QCD Form\)](#)
- [DUMP EXPLAIN](#)
- [INSERT EXPLAIN](#)
- [INITIATE INDEX ANALYSIS](#)
- [RESTART INDEX ANALYSIS](#)
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ - Database Design*, B035-1094
- *Teradata® Index Wizard User Guide*, B035-2506
- *Teradata® Viewpoint User Guide*, B035-2206, Stats Manager topic
- *Teradata® Visual Explain User Guide*, B035-2504
- *Teradata® System Emulation Tool User Guide*, B035-2492

### Required Privileges

You must have INSERT privileges on the tables in the user-defined query capture database.

You must also have the SELECT privilege on all referenced tables to capture data demographics and column statistics.

## Syntax



## Syntax Elements

### WITH STATISTICS

Capture the statistics for the WHERE clause condition columns specified in *SQL\_statement*.

You cannot specify WITH STATISTICS if you also specify the IN XML option.

Specifying this option is equivalent to performing COLLECT STATISTICS (QCD Form) on the query table columns.

The columns inserted into the QCD are those determined to be index candidates by the Teradata Index Wizard.

### USING SAMPLE *percentage*

### USING SAMPLE *percentage* PERCENT

Percentage of data to be read for collecting statistics.

If you do not specify this clause, the system automatically selects a sample size based on the cardinality of the table.

You cannot specify a USING SAMPLE clause if you also specify an IN XML clause.

The valid range is between 1 and 100, inclusive.

### WITH NO STATISTICS

Do not capture statistics for the WHERE clause condition columns specified in *SQL\_statement*.

### AND DEMOGRAPHICS

Collect table-level demographics as well as statistics.

Specifying this option is equivalent to performing a COLLECT DEMOGRAPHICS request on the query tables. See [COLLECT DEMOGRAPHICS](#).

If you do not specify this option, then the system does not collect table-level demographics.

**FOR *table\_name***

**FOR *database\_name.table\_name***

**FOR *user\_name.table\_name***

Set of tables for which data demographics and statistics details are either to be included or excluded for analysis by the Teradata Index Wizard, depending on whether you specify the WITH STATISTICS option or the WITH NO STATISTICS options, respectively.

Statistics are collected on columns that are index candidates and for all columns and indexes referenced explicitly with values in *SQL\_statement* for the following cases. A candidate index column is defined as a column for which the value or join range access frequencies stored in the Field table of the specified QCD are greater than 0. If you specify:

- WITH STATISTICS, Teradata Database collects statistics for all tables referenced by the query.
- WITH STATISTICS ... FOR *table\_name*, Teradata Database collects statistics only for the tables referenced by the query that are also specified in the FOR *table\_name* clause.
- WITH NO STATISTICS ... FOR *table\_name*, Teradata Database collects statistics only for the tables referenced by the query that are not listed in the FOR *table\_name* clause.

You cannot specify a FOR *table\_name* clause if you also specify an IN XML clause.

**INTO *QCD\_name***

User-defined query capture database.

A database named *QCD\_name* must exist on the system. Otherwise, the request aborts and returns an error.

Using the Control Center feature of the Visual Explain tool is the easiest way to create your QCD databases.

**AS *query\_plan\_name***

Optional user-defined name under which the query plan information is to be stored. For information on object naming, see *Teradata Vantage™ SQL Fundamentals*, B035-1141.

If no *query\_plan\_name* is specified, the query plan information is stored with a null name. Because each query plan is stored with a unique non-null Query ID, various query plans within a given database can be distinguished. see *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142. Query IDs are not unique across databases.

*query\_plan\_name* is not constrained to be unique.

You can store *query\_plan\_name* as *query\_plan\_name* if you enclose the pad character-separated words in APOSTROPHE characters as "query plan name".

**LIMIT****LIMIT SQL****LIMIT SQL = *n***

Limit the size of the query, DDL, and view text captured for the QCD tables Query, Relation, and ViewText, respectively.

If you do not specify this clause, then the system captures complete SQL text.

The value for *n* indicates the upper limit on the amount of text to capture.

If you specify either LIMIT by itself or LIMIT SQL without a value, then the clause is equivalent to LIMIT SQL = 0, and no text is captured.

**FOR frequency**

Number of times *SQL\_statement* is typically performed within its identified workload.

This value is used to weight the respective benefits of each column analyzed for inclusion in the index recommendation computed by the Teradata Index Wizard utility.

Any positive integer up to 4 bytes is valid.

You cannot specify a FOR *frequency* clause if you also specify an IN XML clause.

If this clause is not specified, then the value for *frequency* defaults to 1.

**CHECK STATISTICS**

Capture COLLECT STATISTICS recommendations for *SQL\_statement* in the StatsRecs QCD table. See *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142.

You cannot specify CHECK STATISTICS if you also specify the IN XML option.

**IN XML**

Capture the output as an XML document.

This document is stored in the QCD table named XMLQCD.

You cannot specify an IN XML clause if you also specify any of the following options:

- USING SAMPLE
- FOR *table\_name*
- FOR *frequency*

If you specify the IN XML option, you can also specify the NODDLTEXT option.

If you specify the IN XML option, you cannot specify any of the following options with it:

- CHECK STATISTICS
- WITH STATISTICS

**NODDLTEXT**

Do not capture the DDL text in the XML document stored in qcd.XMLQCD.

**SQL\_request**

DML request whose Optimizer white tree information is to be captured and written to the QCD.

*SQL\_request* is limited to the following statements.

- DELETE
- EXEC (Macro Form)
- INSERT
- MERGE
- SELECT
- UPDATE

## ANSI Compliance

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

## Usage Notes

### Invocation

Normally invoked using client-based database query analysis tools.

### Rules for Using INSERT EXPLAIN

The following rules apply to the use of INSERT EXPLAIN:

- When you specify a table list, but none of the tables referenced by *SQL\_statement* are in the specified table list, an error is returned.
- When you specify a table list, but not all of the tables referenced by *SQL\_statement* are in the specified table list, no error is returned, but the nonmatching tables are ignored.

### INSERT EXPLAIN Actions Without An Existing QCD

Ensure that there is an existing *QCD\_name* database before running INSERT EXPLAIN for the first time. Use the Control Center feature of the Visual Explain tool to create your QCD databases. For more information, see *Teradata® Visual Explain User Guide*, B035-2504.

If a *QCD\_name* database or any of its tables does not exist, or if you do not have the appropriate privileges on one or more *QCD\_name* database tables when you perform the INSERT EXPLAIN request, then the system displays an appropriate error message and terminates performance of the request.

### Creating Index Analysis Workloads Using the WITH STATISTICS and FOR frequency Options

When you specify the WITH STATISTICS option with INSERT EXPLAIN, sample table cardinality estimates and column statistics are captured in the TableStatistics tables of your QCD databases. The confidence the Optimizer has in its index recommendations is expressed in the Level column of the QCD table StatsRecs. For details about the *StatsRec* table, see “Query Capture Facility” in *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142. This information is critical to the index recommendation analyses performed by the Teradata Index Wizard.

The sampled statistics captured using the WITH STATISTICS clause are identical to those captured by performing a COLLECT STATISTICS (QCD form) request and a COLLECT request. For more information,

see “COLLECT STATISTICS (Optimizer Form)” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

The FOR *frequency* clause provides an estimate of the average number of times the SQL request being analyzed is typically performed during the submitted workload. This information is used by the Teradata Index Wizard utility to determine the benefit of including various columns as indexes.

## Collecting New Statistics When You Specify INSERT EXPLAIN WITH STATISTICS

INSERT EXPLAIN WITH STATISTICS collects fresh, sampled statistics only if there are no statistics for the candidate index column set in DBC.TVFields for *table\_name*. Otherwise, the existing statistics, whether based on a sample or not, are copied to the QCD and are not recollected. This enables the index analysis process to make full use of nonsampled statistics if they are available.

### Actions Performed by INSERT EXPLAIN

INSERT EXPLAIN performs the following actions in the order indicated:

1. Runs an EXPLAIN on the SQL DML request specified by *SQL\_statement* to generate an Optimizer white tree.
2. Captures the Optimizer white tree output of that EXPLAIN.
3. Writes the output to the appropriate tables in the user-specified query capture database.

Option	Description
USING SAMPLE	Collects statistics on the specified sample size from the population rather than allowing the system to select a sample size based on the cardinality of the table.
CHECK STATISTICS	Generates a set of recommended statistics to collect and stores the recommendations in the StatsRec QCD table.

You cannot specify either of these options if you also specify the IN XML option.

If you are collecting statistics for use by the Teradata Index Wizard utility, then INSERT EXPLAIN performs the following actions in the order indicated:

1. Runs an EXPLAIN (see [EXPLAIN Request Modifier](#)) on the SQL DML request specified by *SQL\_statement* to generate an Optimizer white tree.
2. Captures the Optimizer white tree output for that EXPLAIN.
3. Captures the table cardinality and column statistics for the tables referenced by *SQL\_request*, excluding any tables explicitly excluded by the FOR *table\_name* clause. The columns are those identified as potential index candidates.
4. Captures the number of times *SQL\_request* is performed in the identified workload as specified by the FOR *frequency* clause.
5. Writes the output to the appropriate tables in the user-specified query capture database.

## INSERT EXPLAIN Is Not Valid Within a Multistatement Request

You cannot submit an INSERT EXPLAIN request as part of a multistatement request, though you can submit an INSERT EXPLAIN request for a multistatement request. If you attempt to submit a multistatement request that contains an INSERT EXPLAIN request, the multistatement request returns an error.

While the INSERT EXPLAIN request in the following example appears to be a valid statement in a multistatement request, request actually captures the query plan for a multistatement request that follows it, and is not itself part of that multistatement request. As a result, the request is treated like any other INSERT EXPLAIN and completes successfully.

```
INSERT EXPLAIN INTO qcd SELECT * FROM d1.t1
;SELECT * FROM d1.t1;
*** Insert completed. One row added.
*** Total elapsed time was 1 second.
```

## Effects of Data Parcel Peeking on INSERT EXPLAIN Outcomes

When a data parcel is submitted with an INSERT EXPLAIN request, the plan might be generated with peeking at USING and CURRENT\_DATE or DATE values, or both. If any of these values are peeked, then the query plan shows them.

If no data parcel is submitted with an INSERT EXPLAIN request, the resulting plan is generated without peeking at USING, CURRENT\_DATE, or DATE values, so it is a generic plan by definition. Note that the Visual Explain, Teradata System Emulation Tool, and Teradata Index Wizard tool do not accept USING data as input while capturing query plans using INSERT EXPLAIN requests unless those requests are submitted using BTEQ or Teradata SQL Assistant.

The Teradata Index Wizard internally generates plans for workload queries for the purpose of estimating workload costs, which are used to determine optimal index recommendations. When queries in the workloads specify USING request modifiers, the plan is generated without peeking at USING, CURRENT\_DATE, or DATE values. Because of these factors, Request Cache peeking has no impact on the resulting index. Given that workload analyses should be independent of USING values, this behavior is correct.

## Capturing the INSERT EXPLAIN Output as an XML Document

The IN XML option for INSERT EXPLAIN provides a functionally equivalent representation of a query plan created by an INSERT EXPLAIN request that does not produce XML output.

INSERT EXPLAIN operations that require several minutes to be processed when you do not specify the IN XML option take only a few seconds to complete when you capture the information as an XML file. This is because an INSERT EXPLAIN ... IN XML request performs only one insert operation by storing the entire output of the request as one or more 31,000 byte slices of a single XML file. When INSERT EXPLAIN inserts XML data into the QCD table XMLQCD, it also updates the QCD table SeqNumber, but the update of SeqNumber is a trivial operation.

When you specify the IN XML option, Teradata Database does not update any other QCD tables beyond XMLQCD and SeqNumber. This is unlike a standard INSERT EXPLAIN request where inserts must be made into multiple QCD tables.

In a single QCD table, XMLQCD has a marked performance advantage over a standard INSERT EXPLAIN QCD data collection operation. For the definition of the XMLQCD table, see *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142

A similar option for the BEGIN QUERY LOGGING and REPLACE QUERY LOGGING statements optionally creates a compact XML version of QCD information that can be stored in a single DBQL table, DBC.DBQLXMLTbl, making the features complementary to one another.

XMLPLAN query logging is not an alternative method for capturing the information captured by INSERT EXPLAIN requests.

- In addition to differences in the content of the documents produced, these two methods have an important difference regarding the execution of the query.

BEGIN QUERY LOGGING ... XMLPLAN	INSERT EXPLAIN description
Logs query plans for executed queries.	captures query plans <i>without</i> executing the query.

XMLPLAN logging is ideal when you want to record query plans for your executing workloads and using INSERT EXPLAIN requests to capture query plans for those workloads is too slow for your needs.

If you are only tuning a query and do not want to execute it, XMLPLAN logging is not as useful as capturing the query plan for a request using INSERT EXPLAIN requests.

In this case, executing an INSERT EXPLAIN INTO QCD\_ *name* IN XML request or an EXPLAIN IN XML SQL\_ *request* is a more viable alternative. See [Capturing EXPLAIN Text Output in XML Format](#).

- Runtime information from the traditional DBQL tables is also captured for a logged plan.
- XMLPLAN logging is more of an extension to query logging than an extension to the Query Capture Facility.

## Examples

### Example: INSERT EXPLAIN

The following examples, with some exceptions, use the same SELECT request to show how the different syntax options of INSERT EXPLAIN produce different results.

Each INSERT EXPLAIN request inserts the data for the query plan produced by the Optimizer for the SELECT request that it modifies directly into the specified QCD database (see “Query Capture Facility” in *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184 for documentation of the QCD tables).



**Example: Query Plan Name Stated, QCD Name Is *TLE\_Queries***

The output of this INSERT EXPLAIN request is referenced under the query plan name *EmployeeSmithQuery* in the *TLE\_Queries* database. The system collects neither statistics nor demographics for this request.

```
INSERT EXPLAIN
INTO TLE_Queries AS EmployeeSmithQuery
SELECT emp_id, emp_address
FROM employee
WHERE emp_name = 'Smith';
```

**Example: No Query Plan Stated, So Name in *TLE\_Queries* QCD Is Null**

The output of this INSERT EXPLAIN request has a null query plan name in the *TLE\_Queries* database because there is no query plan naming clause. The system collects neither statistics nor demographics for this request.

```
INSERT EXPLAIN
INTO TLE_Queries
SELECT emp_id, emp_address
FROM employee
WHERE emp_name = 'Smith';
```

**Example: Query Plan Name Stated, QCD Name Is *QCD***

The output of this INSERT EXPLAIN request is referenced under the query plan name *Employee Smith Query* in the *QCD* database. The system collects neither statistics nor demographics for this request.

```
INSERT EXPLAIN
INTO QCD AS "Employee Smith Query"
SELECT emp_id, emp_address
FROM employee
WHERE emp_name = 'Smith';
```

**Example: Workload Execution Frequency Clause Specified**

The output of this INSERT EXPLAIN request is referenced under the query plan name *Wizard Test* in the *Wizard\_QCD* database. Statistics are collected for *table\_1* and column statistics are collected for *column\_2* and *column\_3* in *table\_1*. The frequency of performance of the specified SQL SELECT request is 10 times per workload. No demographics are collected.

```
INSERT EXPLAIN WITH STATISTICS FOR table_1
INTO Wizard_QCD AS "Wizard Test" FOR 10
SELECT *
FROM table_1
WHERE table_1.column_2 = 10
```

```
AND table_1.column_3 = 20
FOR 10;
```

### Example: Statistics on Single Table With Two Tables Selected

The output of this INSERT EXPLAIN request is saved under the query plan name *WizardTest2* in the *Wizard\_QCD* database. Statistics are collected for the *employee* table and column statistics are collected for the *emp\_lastname* column. No statistics are captured for the *department* table because the request specifies they are to be collected only for the *employee* table in the *FOR table\_name* clause.

```
INSERT EXPLAIN WITH STATISTICS FOR employee
INTO Wizard_QCD AS WizardTest2
SELECT employee.first_name, department.dept_num
FROM employee, department
WHERE employee.emp_lastname = 'Smith'
AND department.dept_name IN ('Engineering', 'Support');
```

### Example: SQL Query Does Not Reference The Tables Named In The Table List

This INSERT EXPLAIN request returns an error because none of the tables specified in the table list are referenced in the specified query.

```
INSERT EXPLAIN WITH STATISTICS FOR employee1
INTO Wizard_QCD AS WizardTest2
SELECT employee.first_name, department.dept_num
FROM employee, department
WHERE employee.emp_lastname = 'Smith'
AND department.dept_name IN ('Engineering', 'Support');

****Failure 5644: No match found for specified tables in the request.
```

### Example: Specifying Statistical Sampling

This example uses an explicit sampling percentage of 80 to capture QCD statistics for the specified SELECT request:

```
INSERT EXPLAIN WITH STATISTICS
USING SAMPLE 80 PERCENT
INTO MyQCD AS query1
SELECT t2.y3, t1.x3
FROM t1, t2
WHERE t1.pi = t2.y2;
```

### Example: Generating Single-Column Statistics

Single-column statistics recommendations are represented by a single row in the *StatsRecs* table. In this example there are two separate recommendations, one for the primary index on column *a1* and one for the nonindexed column *c1*.

After you run this request, you can then apply the generated recommendations for collecting statistics by running the two *COLLECT STATISTICS* requests that the system stores in the *StatsDDL* column of the *StatsRec* table in the *QCD* database.

The definition for the only table referenced by the SQL request to be analyzed is as follows:

```
CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER,
  c1 INTEGER);
```

The following *INSERT EXPLAIN* request generates the single-column statistics recommendations for the SQL request *SELECT \* FROM t1 WHERE c1=5*.

```
INSERT EXPLAIN INTO QCD
CHECK STATISTICS
SELECT * FROM t1 WHERE c1 = 5;
```

The following *SELECT* request selects all the columns that contain the statistics recommendations for *QueryID* 1, which is the system-assigned unique identifier for the row set in the *QCD StatsRecs* table that contains the recommendations:

```
SELECT *
FROM QCD.StatsRecs
WHERE QueryID = 1
ORDER BY StatsID;
```

The query returns the following report:

```
QueryID          1
StatsID          0
DatabaseName D1
TableName t1
FieldID         1025
FieldName A1
Weight          3
StatsDDL COLLECT STATISTICS D1.t1 COLUMN A1 ;
QueryID          1
StatsID          1
DatabaseName D1
TableName t1
```

```

FieldID      1027
FieldName C1
Weight       3
StatsDDL COLLECT STATISTICS D1.t1 COLUMN C1 ;

```

### Example: Generating Multicolumn Statistics

Multicolumn statistics recommendations are represented by multiple rows in the StatsRecs table. Each column of the recommendation is stored in a separate row, sharing its *StatsID* value with the other columns in the set, each of which is represented by its own row.

Note how the COLLECT STATISTICS request text is stored in the row corresponding to the lowest *FieldID* value. The StatsDDL value for the other rows is null. In this example, there is one recommendation for the multicolumn primary index.

After you run this request, you can then apply the generated recommendations for collecting statistics by running the COLLECT STATISTICS request that the system stores in the *StatsDDL* column of the *StatsRec* table in the QCD database.

The definition for the only table referenced by the SQL request to be analyzed is as follows:

```

CREATE TABLE t2 (
  a2 INTEGER,
  b2 INTEGER,
  c2 INTEGER,
  d2 INTEGER)
PRIMARY INDEX (a2, b2, c2);

```

The following INSERT EXPLAIN request generates the multicolumn statistics recommendations for the SQL request SELECT \* FROM t2.

```

INSERT EXPLAIN INTO QCD
CHECK STATISTICS
SELECT * FROM t2;

```

The following SELECT request selects all the columns that contain statistics recommendations for QueryID 2, which is the system-assigned unique identifier for the row set in the QCD StatsRecs table that contains the recommendations:

```

SELECT *
FROM QCD.StatsRecs
WHERE QueryID = 2
ORDER BY StatsID, FieldID;

```

The query returns the following report:

```

QueryID      2
StatsID      0

```

```

DatabaseName D1
  TableName t2
    FieldID      1025
    FieldName A2
      Weight      3
      StatsDDL COLLECT STATISTICS D1.t2 COLUMN (A2 ,B2 ,C2 ) ;
      QueryID      2
      StatsID      0
DatabaseName D1
  TableName t2
    FieldID      1026
    FieldName B2
      Weight      3
      StatsDDL ?
      QueryID      2
      StatsID      0
DatabaseName D1
  TableName t2
    FieldID      1027
    FieldName C2
      Weight      3
      StatsDDL ?

```

There are three rows in the report: one for each of the columns that make up the primary index for the table. You can tell these are multicolumn statistics because all three rows share the same QueryID value (2) and the same StatsID value (0). Another sign that these are multicolumn statistics is that there is only one column of DDL text, and it is stored in the row with the lowest FieldID value (1025).

### Example: Capturing Output Data as an XML Document

This example produces one row of query capture output and stores it in XML format in the QCD table *myqcd.XMLQCD*.

```

INSERT EXPLAIN INTO myqcd IN XML
SELECT *
FROM DBC.DBCInfoV;

```

The following output shows a fragment of the row inserted into *myqcd.XMLQCD* table.

```

      Id      1
      Kind     Q
      Seq      1
      Length   4501
      Text  <?xml version="1.0" encoding="UTF-8"?><QCF>
<ElapsedSeconds>0.030<ElapsedSeconds><QueryID>8<QueryID>

```

```
<User_Database><UDB_Key>1</UDB_Key><UDB_ID>256</UDB_ID>
<MachineName>localhost</MachineName><UDB_Name>DBC</UDB_Name>
</User_Database><Query><UDB_Key>1</UDB_Key><MachName>localhost
</MachName><NumAMPs>2</NumAMPs><NumPEs>1<NumPEs><NumNodes>1
<NumNodes><ReleaseInfo>13w.00.00.00</ReleaseInfo><VersionInfo>
13w.00.00.00</VersionInfo>...
```

### Example: Capturing Output Data as an XML Document With No Associated DDL Text

This example produces one row of query capture output and stores it in XML format in the QCD table *myqcd.XMLQCD*. Because you specified the NODDLTEXT option, the system does not store the DDL SQL text related to the request.

```
INSERT EXPLAIN INTO myqcd IN XML NODDLTEXT
SELECT *
FROM DBC.DBCInfoV;
```

## RESTART INDEX ANALYSIS

### Purpose

Restarts a previously halted index analysis started by an INITIATE INDEX ANALYSIS or RESTART INDEX ANALYSIS request that specified a checkpoint.

For more information about index analysis, see:

- [COLLECT DEMOGRAPHICS](#)
- [COLLECT STATISTICS \(QCD Form\)](#)
- [INITIATE INDEX ANALYSIS](#)
- [INSERT EXPLAIN](#)
- *Teradata Vantage™ SQL Request and Transaction Processing*, B035-1142
- *Teradata® Index Wizard User Guide*, B035-2506

### Required Privileges

The privileges required to submit a RESTART INDEX ANALYSIS request are the following:

You must have this privilege set ...	On this table set in database <i>QCD_name...</i>
INSERT	<ul style="list-style-type: none"> <li>• <i>IndexColumns</i></li> <li>• <i>IndexRecommendations</i></li> </ul>
<ul style="list-style-type: none"> <li>• INSERT</li> <li>• DELETE</li> </ul>	<i>AnalysisLog</i>

The user who performs a RESTART INDEX ANALYSIS request must be the same user who performed the checkpointed INITIATE INDEX ANALYSIS request.

## Syntax

```

RESTART INDEX ANALYSIS — FOR — workload_name — IN — QCD_name — (A)

(A) — AS — index_name_tag ————— (B)
      |
      | CHECKPOINT — checkpoint_trigger —
      |
      |
(B) —————
      |
      | TIME LIMIT = elapsed_time —
      |
      | ; —▶
  
```

## Syntax Elements

### *workload\_name*

Name of the workload on which index analysis is to be restarted.

### *QCD\_name*

Name of the query capture database in which *workload\_name* is found.

### *index\_name\_tag*

Name of the index analysis to be restarted.

A row with this name must exist in the QCD *AnalysisLog* or *AnalysisStmts* table or you cannot restart the index analysis.

### CHECKPOINT *checkpoint\_trigger*

Number of queries after which a checkpoint snapshot must be taken.

The value for *checkpoint\_trigger* must be a positive integer.

- If the value is 0, Teradata Database ignores the CHECKPOINT clause.
- If the value is greater than the total number of queries to be analyzed, Teradata Database does not take a checkpoint snapshot.

Also see [Example: Using a CHECKPOINT](#).

### TIME LIMIT = *elapsed\_time*

The maximum elapsed time in minutes that this restarted index analysis should take to complete.

The default value is no time limit.

The permitted range of specified values is from 1 to 2880 minutes, for a maximum of 48 hours.

You must specify the value for *elapsed\_time* as an integer.

If the index analysis does not complete before reaching the specified time limit, the system stops the task and retains the best recommendations found up to the point when the time limit expired.

Note that the time limit that you specify is only an approximation because the ongoing index analysis task only checks periodically to see if the specified time limit has been exceeded.

See [Example: Setting a TIME LIMIT on an Index Analysis](#).

## ANSI Compliance

RESTART INDEX ANALYSIS is a Teradata extension to the ANSI SQL:2011 standard.

## Invocation

Normally invoked using the Teradata Index Wizard utility.

## Rules for Performing RESTART INDEX ANALYSIS

The rules for using RESTART INDEX ANALYSIS and INITIATE INDEX ANALYSIS are the same. For more information, see [Rules for Performing INITIATE INDEX ANALYSIS](#).

## RESTART INDEX ANALYSIS Requires a Checkpoint

If you do not specify a CHECKPOINT clause in your INITIATE INDEX ANALYSIS request, then you cannot restart a previously halted index analysis.

If you are not satisfied with the outcome of a previously timed out INITIATE INDEX ANALYSIS request and want to extend the analysis with the intent of achieving better results, you can restart the analysis from the point at which it was stopped by specifying a CHECKPOINT rather than redoing the entire INITIATE INDEX ANALYSIS request from the beginning.

## Actions Performed by RESTART INDEX ANALYSIS

RESTART INDEX ANALYSIS functions like INITIATE INDEX ANALYSIS, except that RESTART INDEX ANALYSIS begins its analysis with the first SQL request in the workload that was not checkpoint logged in the AnalysisLog table.

If the restart aborts for any reason before completion, the next RESTART INDEX ANALYSIS request begins its analysis with the first SQL request in the workload that was not checkpointed in the AnalysisLog table.

Once the restart completes successfully, index recommendations are written to the IndexRecommendations table within the specified QCD.

## RESTART INDEX ANALYSIS Not Supported From Macros

You cannot specify a RESTART INDEX ANALYSIS request from a macro. If you execute a macro that contains a RESTART INDEX ANALYSIS request, Teradata Database aborts the request and returns an error.

## Example: Index Analysis With CHECKPOINT

Assume that the query capture database named *MyQCD* exists on the system. The workload named *MyWorkload* in *MyQCD* consists of 100 SQL requests. The following SQL request causes the index analysis information to be saved after every 10 SQL requests because of its CHECKPOINT clause specification:

```
INITIATE INDEX ANALYSIS ON tab1
FOR MyWorkload
```



```
IN MyQCD AS table_1Index
CHECKPOINT 10;
```

Suppose a database restart occurs while processing the 55<sup>th</sup> SQL request. The system then records the information pertaining to the first 50 SQL requests processed.

The following SQL request causes the analysis to be done from the 51<sup>st</sup> SQL request.

```
RESTART INDEX ANALYSIS
FROM MyWorkload
IN MyQCD AS table1_Index;
```

### Example: Index Analysis Without CHECKPOINT

Again assume that the query capture database *MyQCD* exists on the system and that the workload named *MyWorkload* consists of 100 SQL requests.

The following SQL request does not specify a checkpoint, so no incremental information is written to the *AnalysisLog* table in *MyQCD*.

```
INITIATE INDEX ANALYSIS ON tab1
FOR MyWorkload
IN MyQCD AS table_1Index;
```

Suppose once again that a database restart occurs while processing the 55<sup>th</sup> SQL request.

The following SQL request returns an error because there is no row in *AnalysisLog* that checkpoints the specified index analysis name, *table\_1Index*.

```
RESTART INDEX ANALYSIS      FROM MyWorkload      IN MyQCD AS table_1Index;

*** Failure 5669 No restart information found for specified index      analysis
'table_1Index'.
```

# Notation Conventions

## About Notation Conventions

This section describes the notation conventions used in this document.

Convention	Description
Syntax Diagrams	Describes SQL syntax form, including options.
Square braces in the text	<p>Represent options. The indicated parentheses are required when you specify options. For example:            DECIMAL [(n[,m])] means the decimal data type can be defined optionally:</p> <ul style="list-style-type: none"> <li>• without specifying the precision value n or scale value m</li> <li>• specifying precision (n) only</li> <li>• specifying both values (n,m)</li> </ul> <p>You cannot specify scale without first defining precision.</p> <ul style="list-style-type: none"> <li>• CHARACTER [(n)] means that use of (n) is optional.</li> </ul> <p>The values for n and m are integers in all cases.</p>

## Syntax Diagram Conventions

### Notation Conventions

Item	Definition and Comments
Letter	An uppercase or lowercase alphabetic character ranging from A through Z.
Number	<p>A digit ranging from 0 through 9.</p> <p>Do not use commas when typing a number with more than 3 digits.</p>
Word	<p>Keywords and variables.</p> <ul style="list-style-type: none"> <li>• UPPERCASE LETTERS represent a keyword.            Syntax diagrams show all keywords in uppercase, unless operating system restrictions require them to be in lowercase.</li> <li>• lowercase letters represent a keyword that you must type in lowercase, such as a Linux command.</li> <li>• Mixed Case letters represent exceptions to uppercase and lowercase rules. The exceptions are noted in the syntax explanation.</li> <li>• <i>lowercase italic letters</i> represent a variable such as a column or table name.            Substitute the variable with a proper value.</li> <li>• <b>lowercase bold letters</b> represent an excerpt from the diagram.            The excerpt is defined immediately following the diagram that contains it.</li> </ul>

Item	Definition and Comments
	<ul style="list-style-type: none"> <li><u>UNDERLINED LETTERS</u> represent the default value. This applies to both uppercase and lowercase words.</li> </ul>
Spaces	Use one space between items such as keywords or variables.
Punctuation	Type all punctuation exactly as it appears in the diagram.

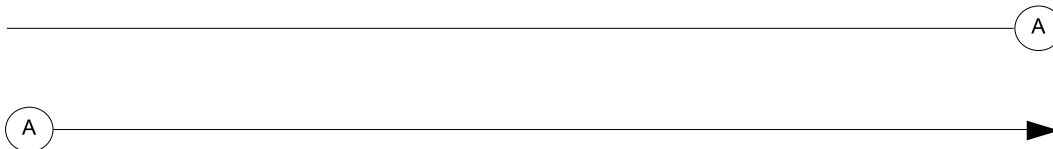
## Paths

The main path along the syntax diagram begins at the left with a keyword, and proceeds, left to right, to the vertical bar, which marks the end of the diagram. Paths that do not have an arrow or a vertical bar only show portions of the syntax.

The only part of a path that reads from right to left is a loop.

## Continuation Links

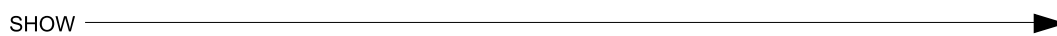
Paths that are too long for one line use continuation links. Continuation links are circled letters indicating the beginning and end of a link:



When you see a circled letter in a syntax diagram, go to the corresponding circled letter and continue reading.

## Required Entries

Required entries appear on the main path:



If you can choose from more than one entry, the choices appear vertically, in a stack. The first entry appears on the main path:



## Optional Entries

You may choose to include or disregard optional entries. Optional entries appear below the main path:



If you can optionally choose from more than one entry, all the choices appear below the main path:



Some commands and statements treat one of the optional choices as a default value. This value is UNDERLINED. It is presumed to be selected if you type the command or statement without specifying one of the options.

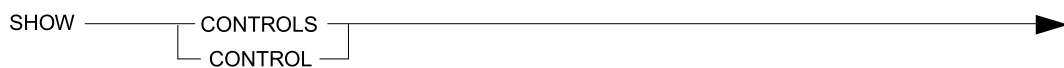
## Strings

String literals appear in apostrophes:



## Abbreviations

If a keyword or a reserved word has a valid abbreviation, the unabbreviated form always appears on the main path. The shortest valid abbreviation appears beneath.



In the above syntax, the following formats are valid:

SHOW CONTROLS  
SHOW CONTROL

## Loops

A loop is an entry or a group of entries that you can repeat one or more times. Syntax diagrams show loops as a return path above the main path, over the item or items that you can repeat:



Read loops from right to left.

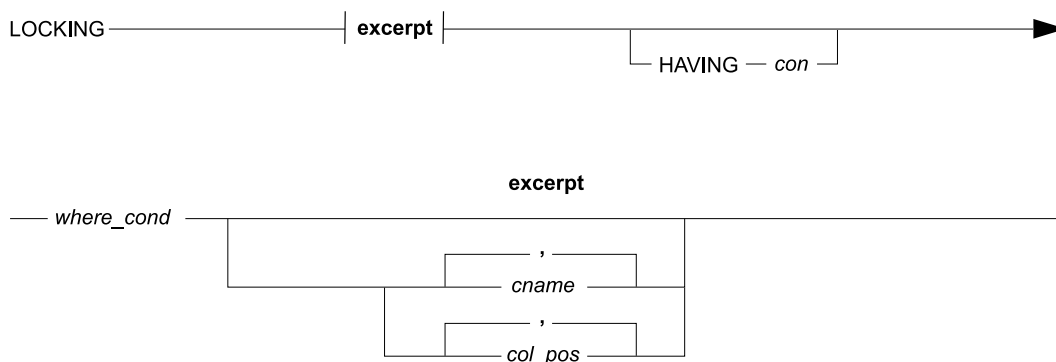
The following conventions apply to loops:

Item	Description	Example
maximum number of entries allowed	The number appears in a circle on the return path.	In the example, you may type <i>cname</i> a maximum of four times.
minimum number of entries allowed	The number appears in a square on the return path.	In the example, you must type at least three groups of column names.
separator character required between entries	The character appears on the return path. If the diagram does not show a separator character, use one blank space.	In the example, the separator character is a comma.
delimiter character required around entries	The beginning and end characters appear outside the return path. Generally, a space is not needed between delimiter characters and entries.	In the example, the delimiter characters are the left and right parentheses.

## Excerpts

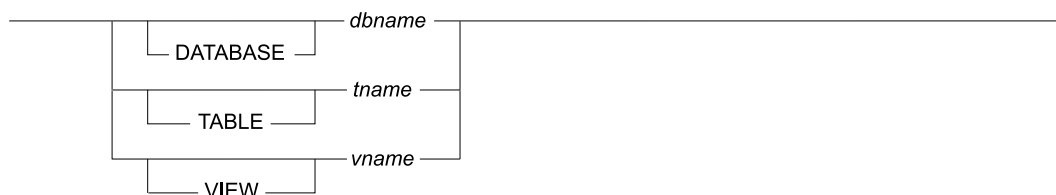
Sometimes a piece of a syntax phrase is too large to fit into the diagram. Such a phrase is indicated by a break in the path, marked by (|) terminators on each side of the break. The name for the excerpted piece appears between the terminators in boldface type.

The boldface excerpt name and the excerpted phrase appears immediately after the main diagram. The excerpted phrase starts and ends with a plain horizontal line:



## Multiple Legitimate Phrases

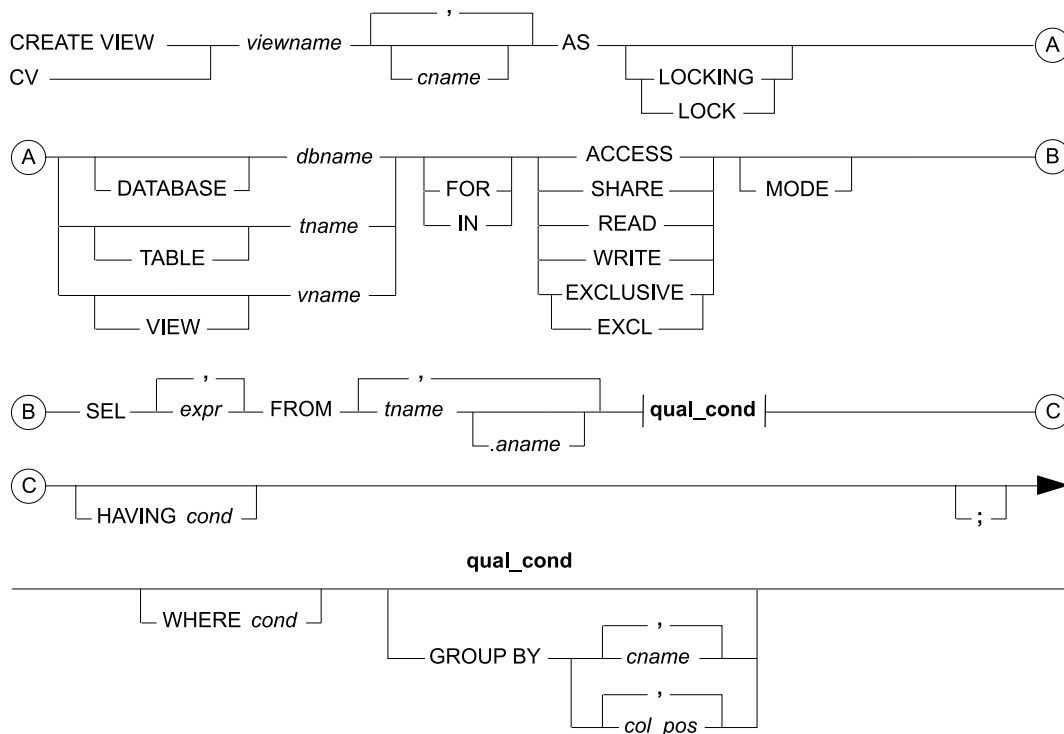
In a syntax diagram, it is possible for any number of phrases to be legitimate:



In this example, any of the following phrases are legitimate:

*dbname*DATABASE *dbname**tname*TABLE *tname**vname*VIEW *vname*

## Sample Syntax Diagram



## 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™ NewSQL 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.
<	KanjiEBCDIC	Shift Out [SO] (0x0E). Indicates transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEBCDIC	Shift In [SI] (0x0F). Indicates transition from multibyte to single byte KanjiEBCDIC.
T	Any	Any multibyte character. The encoding depends on the current character set. For KanjiEUC, code set 3 characters are always preceded by <code>ss3</code> .
!	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by <code>ss2</code> , forming an individual multibyte character.
<u>Δ</u>	Any	Represents the graphic pad character.
Δ	Any	Represents a single or multibyte pad character, depending on context.
ss 2	KanjiEUC	Represents the EUC code set 2 introducer (0x8E).
ss 3	KanjiEUC	Represents the EUC code set 3 introducer (0x8F).

For example, string “TEST”, where each letter is intended to be a fullwidth character, is written as **TEST**. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set:

LMN<TEST>QRS

is represented as:

D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2

## Pad Characters

The following table lists the pad characters for the various character data types.

Server Character Set	Pad Character Name	Pad Character Value
LATIN	SPACE	0x20
UNICODE	SPACE	U+0020
GRAPHIC	IDEOGRAPHIC SPACE	U+3000
KANJISJIS	ASCII SPACE	0x20
KANJI1	ASCII SPACE	0x20



# Performance Considerations

## Logging Row Counts for DML Statements

You can log the row counts of DML statements in DBQLogTbl for single and multistatement requests when you specify BEGIN QUERY LOGGING. See "BEGIN QUERY LOGGING" in *Teradata Vantage™ SQL Data Definition Language Syntax and Examples*, B035-1144.

Row counts for insert, update, and delete operations are logged in the StmtDMLRowCount column of the DBC.QryLogV view and DBC.QryLogV\_SZ security zone view, including stored procedures, external stored procedures, Teradata Parallel Transporter, MultiLoad, and FastLoad operations. The counts are stored in JSON format. You can use JSON functions to extract data from the views. See *Teradata Vantage™ JSON Data Type*, B035-1150.

### Example: Viewing Row Counts for DML Operations

Here is the table definition for the example.

```
CREATE TABLE t1 (a1 INTEGER, b1 INTEGER, c1 INTEGER);
```

You use this statement to begin logging queries.

```
BEGIN QUERY LOGGING ON ALL;
```

The following statements perform insert, update, and delete operations:

```
INSERT INTO t1 VALUES (1,2,3);
```

```
UPDATE t1 SET C1 = 10;
```

```
DELETE FROM t1;
```

This statement ends query logging.

```
END QUERY LOGGING ON ALL;
```

Now, you can query the StmtDMLRowCount column of the view QryLogV in the DBC database to display a count of the rows that were changed.

```
SELECT StmtDMLRowCount FROM DBC.QryLogV;
```

```
StmtDMLRowCount
```

```
-----
```

```
{"Insert":1}
```

```
{"Update":1}
```

```
{"Delete":1}
```

## Optimizing INSERT ... SELECT Requests

### Empty Table INSERT ... SELECT Requests and Performance

An INSERT ... SELECT optimizes performance when the target table is empty. If the target table has no data, INSERT ... SELECT operates on an efficient block-by-block basis that bypasses journaling.

Normally, when inserting a row into a table, the system must make a corresponding entry into the transaction journal (TJ) to roll back the inserted row in case the transaction aborts.

- For inserts into populated tables, if the transaction aborts, the system deletes all inserts from the table one row at a time by scanning the TJ for the transaction-related RowIDs.
- For inserts into empty tables, if the transaction aborts, the system can easily return the table to its original state by deleting all rows.

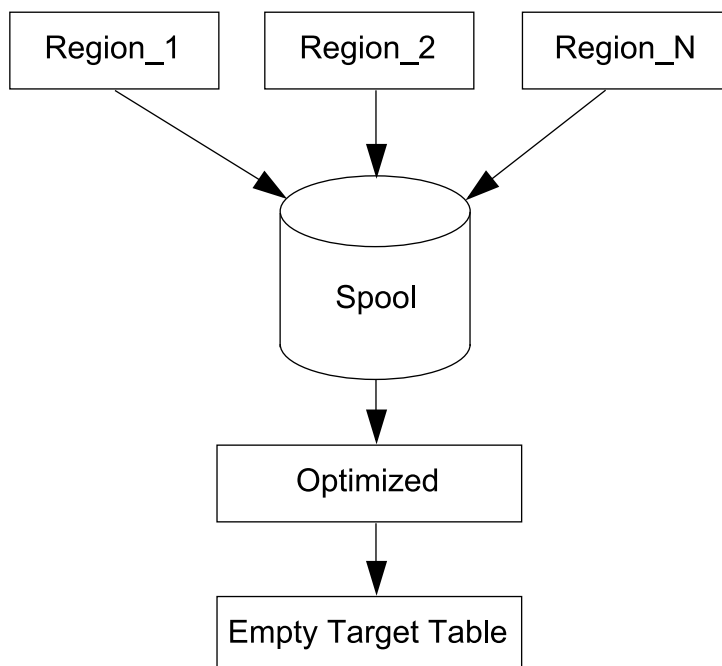
The advantages of using optimized INSERT ... SELECTs are:

- Block-at-a-time processing
- Faster insert logic, to eliminate block merge complexity
- Instantaneous rollback for aborted INSERT ... SELECTs

### Example: INSERT ... SELECT

Using multiple Regional Sales History tables, build a single summary table by combining summaries from the different regions. Then insert these summaries into a single table via a multistatement INSERT ... SELECT statement.

All multistatement INSERT ... SELECT statements output to the same spool table. The output is sorted and inserted into an empty table.



Form a multistatement request by semicolon placement in BTEQ as shown below, or by placing statements in a single macro.

If you execute each of the statements as separate requests, only the first statement is inserted into an empty table.

```
INSERT into Summary_Table
SELECT store, region,sum(sales),count(sale_item)
FROM Region_1
GROUP BY 1,2
;INSERT into Summary_Table
SELECT region2, sum (sales), count(sale_item)
FROM Region_2
GROUP BY 1,2
. . .
;INSERT into Summary_Table
SELECT region3, sum(sales), count(sale_item)
FROM Region_N
GROUP BY 1,2;
```

### **INSERT . . . SELECT Into an Empty SET Table**

INSERT . . . SELECT into an empty SET table from a source known not to have duplicate rows avoids duplicate checking of the target table during insertion. This occurs even during direct insertion from another SET table.

This should offer significant performance improvement in cases where there is a NUPI that is relatively nonunique or has few values that are very nonunique.

### **INSERT . . . SELECT with FastLoad**

Use the optimized INSERT . . . SELECT to manipulate FastLoad data:

1. FastLoad into a staging table.
2. INSERT . . . SELECT into the final table, manipulating the data as required.

FastLoad and INSERT . . . SELECT are faster than using an INMOD to manage data on the host. The host is a single bottleneck as opposed to parallel AMPs that populate temporary tables for reports or intermediate results.

Multiple source tables may populate the same target table. If the target table is empty before a request begins, all INSERT . . . SELECT statements in that request run in the optimized mode.

The staging table can be a No Primary Index (NoPI) table, which has rows that are not hash distributed, as the staging table. FastLoad runs faster inserting into a NoPI table because there is no sort and no row redistribution involved; rows are simply appended to the table evenly across all AMPs.

Use of NoPI reduces skew in intermediate tables without a primary index and is useful for staging tables. However, because the rows of a NoPI table are not hashed-based, an INSERT/SELECT from a NoPI table

to a PI table may be slower than INSERT/SELECT where the tables share the same PI definition. Consider NoPI tables for use with BI tools and applications that generate many intermediate tables

An INSERT ... SELECT from a NoPI table to a PI table can be slower than an INSERT ... SELECT from a PI table to a PI table with the same PI.

### **INSERT . . . SELECT with Join Index**

The fastest way of processing inserts into a table with a join index is:

1. Use FastLoad to load the rows into a staging table with no indexes or join indexes defined.
2. Do an INSERT ... SELECT from the staging table into the target table with the join index.

If the target table has multiple join indexes defined, the Optimizer may choose to use reusable spool during join index maintenance, if applicable.

## **Bulk SQL Error Logging**

Teradata Database supports bulk SQL error handling for MERGE and INSERT . . . SELECT statements. This permits bulk SQL inserts and updates to be done without the target table restrictions that apply to Teradata Database load utilities.

Load utilities are restricted from unique indexes, join or hash indexes, referential constraints, triggers, and LOBs on the target table.

USI and referential integrity (RI) violations cause the request to abort and rollback after all these violations and all other supported error conditions are logged.

For information on creating and dropping the error table, see *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

## **Using the TOP $n$ Row Option**

As an option to the SELECT statement, the TOP  $n$  option automatically restricts the output of queries to a certain number of rows. This option provides a fast way to get a small sample of the data from a table without having to scan the entire table. For example, a user may want to examine the data in an Orders table by browsing through only 10 rows from that table.

The value of  $n$  can be passed into the operator by means of a macro, stored procedure, or USING request modifier parameter.

### **Performance Optimizations**

TOP  $n$  option is optimized for handling TOP  $n$  and “any N” requests. Optimizations include:

- Adding an AMP runtime optimization for TOP  $n$  PERCENT operations.
- Extending “any  $n$ ” optimization to INSERT ... SELECT and CREATE TABLE ... AS requests, views, and derived tables for all values of  $n$ .
- Adding an optimization that avoids redistributing the rows for the hash partitioning case when the grouping columns of a window function contain the PI columns of the source relation.

- Adding a RankLimit optimization for a TOP *n* operation that does not specify the WITH TIES option.
- Adding runtime optimizations for TOP *n* in a request that specifies an ORDER BY specification.

### TOP *n* Option Performance Considerations

For best performance, use the TOP *n* option instead of the QUALIFY clause with RANK or ROW\_NUMBER.

- In best cases, the TOP *n* option provides better performance.
- In worse cases, the TOP *n* option provides equivalent performance.

If a SELECT statement using the TOP *n* option does not also specify an ORDER BY clause, the performance of the SELECT statement is better with BTEQ than with FastExport.

## Using Recursive Queries

A recursive query is a way to query hierarchies of data, such as an organizational structure, bill-of-materials, and document hierarchy.

Recursion is typically characterized by three steps:

- Initialization
- Recursion, or repeated iteration of the logic through the hierarchy
- Termination

Similarly, a recursive query has three execution phases:

- Initial result set
- Iteration based on the existing result set
- Final query to return the final result set

### Ways to Specify a Recursive Query

You can specify a recursive query by:

- Preceding a query with the WITH RECURSIVE clause.
- Creating a view using the RECURSIVE clause in a CREATE VIEW statement.

For a complete description of the recursive query feature, with examples that illustrate how it is used and its restrictions, see *Teradata Vantage™ SQL Fundamentals*, B035-1141.

For information on WITH RECURSIVE clause, see [WITH Modifier](#).

For information on recursive views, see *Teradata Vantage™ SQL Data Definition Language Detailed Topics*, B035-1184.

### Recursive Query Performance Considerations

Following are general guidelines regarding the performance impact of recursive queries:

- Using a recursive query shows a significant performance improvement over using temporary tables with a stored procedures. In most cases, there is a highly significant improvement.

- Using the WITH RECURSIVE clause has basically the same or equivalent performance as using the RECURSIVE VIEW.
- In using a recursive query, it is important to put depth limits on the recursion to prevent infinite recursion when there are cycles in the underlying data.

## Sampling Methods

### Random Sampling

Teradata Database supports extracting a random sample from a database table using the SAMPLE clause and specifying one of the following:

- The number rows
- A fraction of the total number of rows
- A set of fractions as the sample

This sampling method assumes that rows are sampled without replacement and that they are not reconsidered when another sample of the population is taken. This method results in mutually exclusive samples when you request multiple samples. In addition, the random sampling method assumes proportional allocation of rows across the AMPs in the system.

### Random Stratified Sampling

In addition to random sampling option, Teradata Database supports stratified sampling.

Random Stratified Sampling, also called proportional or quota random sampling, involves dividing the population into homogeneous subgroups and taking a random sample in each subgroup. Stratified sampling represents both the overall population and key subgroups of the population. The fraction specification for stratified sampling refers to the fraction of the total number of rows in the stratum.

The following apply to stratified sampling.

You can specify...	You cannot specify...
stratified sampling in derived tables, views, and macros	stratified sampling with set operations or subqueries
either a fraction or an integer as the sample size for every stratum	fraction and integer combinations
up to 16 mutually exclusive samples for each stratum	

## Distincts and Multiple Aggregate Distincts

Teradata Database supports the use of:

- One DISTINCT expression when performing an aggregation.
- Multiple aggregate distincts, which allow multiple DISTINCT expressions for aggregates.

For example:

```
SELECT g, SUM(DISTINCT a), SUM(DISTINCT b)
FROM T
GROUP BY g
COUNT(DISTINCT c) > 5;
```

The feature simplifies SQL generation.

## Merge Joins and Performance

### Merge Joins and Nested Join

In a large join operation, a merge join requires less I/O and CPU time than a nested join. A merge join usually reads each block of the inner table only once, unless a large number of hash collisions occur.

A nested join performs a block read on the inner table for each outer row being evaluated. If the number of rows selected from the outer table is large, this can cause each block of the inner table to be read multiple times.

### Merge Join with Covering NUSI

When large outer tables are being joined, a merge join of a table with a covering index of another table can realize a significant performance improvement.

The Optimizer considers a merge join of a base table with a covering NUSI, which gives the Optimizer an additional join method and costing estimate to choose from.

### Logging Problematic Queries

You can log problematic queries in several ways. See [Logging Problematic Queries](#).

## Additional Information

### Teradata Links

Link	Description
<a href="https://docs.teradata.com/">https://docs.teradata.com/</a>	Teradata documentation (HTML)
<a href="http://www.info.teradata.com">http://www.info.teradata.com</a>	Teradata documentation (PDF)
<a href="https://access.teradata.com">https://access.teradata.com</a>	One stop source for Teradata community, product information, and software downloads. Log in for customer access to: <ul style="list-style-type: none"><li>• Support</li><li>• Software updates</li><li>• Knowledge articles</li><li>• Orange Books</li></ul>
<a href="http://www.teradata.com/products-and-services/TEN">http://www.teradata.com/products-and-services/TEN</a>	Teradata education network
<a href="https://community.teradata.com">https://community.teradata.com</a>	Link to Teradata community (also available from the customer portal)