Teradata Database

# SQL Fundamentals

TERADATA

LABS

# Preface

## Purpose

*SQL Fundamentals* describes basic SQL syntax, including Teradata extensions to ANSI/ISO SQL, SQL data definition, control, and manipulation, data handling, and basic query processing.

Use this book with the other books in the SQL book set.

## Audience

System administrators, database administrators, security administrators, application programmers, Teradata field engineers, end users, and other technical personnel responsible for designing, maintaining, and using Teradata Database will find this book useful.

Experienced SQL users can also see simplified statement, data type, function, and expression descriptions in *SQL Quick Reference*.

## Supported Software Releases and Operating Systems

This book supports Teradata® Database 15.10.

Teradata Database 15.10 is supported on:

- SUSE Linux Enterprise Server 10 SP3
- SUSE Linux Enterprise Server 11 SP1

Teradata Database client applications support other operating systems.

## Prerequisites

If you are not familiar with Teradata Database, you will find it useful to read *Introduction to Teradata* before reading this book.

You should be familiar with basic relational database management technology. This book is not an SQL primer.

# Changes to This Book

| Release | Description |
| --- | --- |
| Teradata Database 15.10 June 2015 | Updated the section Defining Indexes for a Table to include information about Column partitioning with a primary index (PI) or primary AMP index (PA). |
| | Added Query Band as a profile option. |
| | Removed references to 30-character limits for object names. Teradata Database supports object names up to 128 characters long in most cases. |

# Additional Information

| URL | Description |
| --- | --- |
| www.info.teradata.com/ | Use the Teradata Information Products Publishing Library site to:<br><br>• View or download a manual:<br><br>  1  Under **Online Publications**, select **General Search**.<br>  2  Enter your search criteria and click **Search**.<br><br>• Download a documentation CD-ROM:<br><br>  1  Under **Online Publications**, select **General Search**.<br>  2  In the **Title** or **Keyword** field, enter *CD-ROM*, and click **Search**. |
| www.teradata.com | The Teradata home page provides links to numerous sources of information about Teradata. Links include:<br><br>• Executive reports, white papers, case studies of customer experiences with Teradata, and thought leadership<br>• Technical information, solutions, and expert advice<br>• Press releases, mentions and media resources |
| www.teradata.com/t/TEN/ | Teradata Customer Education delivers training that builds skills and capabilities for our customers, enabling them to maximize their Teradata investment. |
| https://tays.teradata.com/ | Use Teradata @ Your Service to access Orange Books, technical alerts, and knowledge repositories, view and join forums, and download software patches. |
| developer.teradata.com/ | Teradata Developer Exchange provides articles on using Teradata products, technical discussion forums, and code downloads. |

To maintain the quality of our products and services, we would like your comments on the accuracy, clarity, organization, and value of this document. Please email teradata-books@lists.teradata.com.

# Product Safety Information

This document may contain information addressing product safety practices related to data or property damage, identified by the word Notice. A notice indicates a situation which, if not avoided, could result in damage to property, such as equipment or data, but not related to personal injury.

**Example**:

Notice:     Improper use of the Reconfiguration utility can result in data loss.

# Teradata Database Optional Features

This book may include descriptions of the following optional Teradata Database features and products:

- In-Memory Optimization
- Secure Zones
- Teradata Columnar
- Teradata Row Level Security
- Teradata Temporal
- Teradata Virtual Storage (VS)

You may not use these features without the appropriate licenses. The fact that these features may be included in product media or downloads, or described in documentation that you receive, does not authorize you to use them without the appropriate licenses.

Contact your Teradata sales representative to purchase and enable optional features.

# Table of Contents

## Chapter 2: Basic SQL Syntax ........................................75

## Chapter 3: SQL Data Definition, Control, and Manipulation .109

## Chapter 4: SQL Data Handling ......................................127

# Database Objects

This chapter describes the database objects that you can use to store, manage, and access data in Teradata Database, and provides guidelines to help determine the best object to use for a specific purpose.

## Databases and Users

A database is a container with an allotment of space in which users can create and maintain database objects, including tables, views, triggers, indexes, stored procedures, user-defined functions, and macros.

A user is similar to a database, except that a user has a password and can log on to the system, whereas a database cannot.

The primary Teradata Database user is DBC, which owns all system disk space and contains the Data Dictionary tables and other system objects and information.

### Defining Databases and Users

Before you can create a database or user, you must have sufficient privileges granted to you.

To create a database, use the CREATE DATABASE statement. You can specify the name of the database, the amount of storage to allocate, and other attributes.

To create a user, use the CREATE USER statement. The statement authorizes a new user identification (user name) for the database and specifies a password for user authentication. Because the system creates a database for each user, the CREATE USER statement is very similar to the CREATE DATABASE statement.

### Difference Between Users and Databases

Formally speaking, the difference between a user and a database is that a user has a password and a database does not. Users can also have default attributes such as time zone, date form, character set, role, and profile, while databases cannot.

You might infer from this that databases are passive objects, while users are active objects. That is only true in the sense that databases cannot execute SQL statements. However, a query, macro, or stored procedure can execute using the privileges of the database.

# Tables

A table is referred to in set theory terminology as a relation, from which the expression relational database is derived.

Every relational table consists of one row of column headings (more commonly referred to as column names) and zero or more unique rows of data values.

Formally speaking, each row represents what set theory calls a tuple. Each column represents what set theory calls an attribute.

The number of rows (or tuples) in a table is referred to as its cardinality and the number of columns (or attributes) is referred to as its degree or arity.

## Defining Tables

Use the CREATE TABLE statement to define base tables. This SQL statement specifies a table name, one or more column names, and the attributes of each column. The CREATE TABLE statement can also specify data block size, percent free space, and other physical attributes of the table.

The CREATE/MODIFY USER and CREATE/MODIFY DATABASE statements provide options for creating permanent journal tables.

## Defining Indexes for a Table

An index is a physical mechanism used to store and access the rows of a table. When you define a table, you can define a primary index or primary AMP index and one or more secondary indexes.

If you define a table and you do not specify a PRIMARY INDEX clause, PRIMARY AMP [INDEX], NO PRIMARY INDEX clause, PRIMARY KEY constraint, UNIQUE constraint, or PARTITION BY, the default behavior is for the system to create the table using the first column as the nonunique primary index. (If you prefer that the system creates a table without a primary or a primary AMP index, use DBS Control to change the value of the PrimaryIndexDefault General field. For details, see *Utilities*.)

If the PARTITION BY clause is specified without specifying a PRIMARY INDEX, PRIMARY AMP [INDEX], or NO PRIMARY INDEX clause, the default is NO PRIMARY INDEX regardless of the setting of this field or whether a PRIMARY KEY or UNIQUE clause is specified.

For details on indexes, see "Indexes" on page 27. For details on NoPI tables, see "No Primary Index (NoPI) Tables" on page 22.

## Duplicate Rows in Tables

Though both set theory and common sense prohibit duplicate rows in relational tables, the ANSI standard defines SQL based not on sets, but on bags, or multisets.

A table defined not to permit duplicate rows is called a SET table because its properties are based on set theory, where set is defined as an unordered group of unique elements with no duplicates.

A table defined to permit duplicate rows is called a MULTISET table because its properties are based on a multiset, or bag, model, where bag and multiset are defined as an unordered group of elements that may be duplicates.

| For more information on … | See … |
|---|---|
| rules for duplicate rows in a table | CREATE TABLE in *SQL Data Definition Language*. |
| the result of an INSERT operation that would create a duplicate row | INSERT in *SQL Data Manipulation Language*. |
| the result of an INSERT using a SELECT subquery that would create a duplicate row | |

## Temporary Tables

Temporary tables are useful for temporary storage of data. Teradata Database supports these types of temporary tables.

| Type | Usage |
|---|---|
| Global temporary | A global temporary table has a persistent table definition that is stored in the Data Dictionary. Any number of sessions can materialize and populate their own local copies that are retained until session logoff. |
| | Global temporary tables are useful for storing temporary, intermediate results from multiple queries into working tables that are frequently used by applications. |
| | Global temporary tables are identical to ANSI global temporary tables. |
| Volatile | Like global temporary tables, the contents of volatile tables are only retained for the duration of a session. However, volatile tables do not have persistent definitions. To populate a volatile table, a session must first create the definition. |
| Global temporary trace | Global temporary trace tables are useful for debugging external routines (UDFs, UDMs, and external stored procedures). During execution, external routines can write trace output to columns in a global temporary trace table. |
| | Like global temporary tables, global temporary trace tables have persistent definitions, but do not retain rows across sessions. |

Materialized instances of a global temporary table share the following characteristics with volatile tables:

- Private to the session that created them.
- Contents cannot be shared by other sessions.
- Optionally emptied at the end of each transaction using the ON COMMIT PRESERVE/ DELETE rows option in the CREATE TABLE statement.

- Activity optionally logged in the transient journal using the LOG/NO LOG option in the CREATE TABLE statement.
- Dropped automatically when a session ends.

For details about the individual characteristics of global temporary and volatile tables, see "Global Temporary Tables" on page 14 and "Volatile Tables" on page 18.

## Global Temporary Tables

Global temporary tables allow you to define a table template in the database schema, providing large savings for applications that require well known temporary table definitions.

The definition for a global temporary table is persistent and stored in the Data Dictionary. Space usage is charged to login user temporary space. Each user session can materialize as many as 2000 global temporary tables at a time.

### How Global Temporary Tables Work

To create the base definition for a global temporary table, use the CREATE TABLE statement and specify the keywords GLOBAL TEMPORARY to describe the table type. Although space usage for materialized global temporary tables is charged to temporary space, creating the global temporary table definition requires an adequate amount of permanent space.

Once created, the table exists only as a definition. It has no rows and no physical instantiation.

When any application in a session accesses a table with the same name as the defined base table, and the table has not already been materialized in that session, then that table is materialized as a real relation using the stored definition. Because that initial invocation is generally due to an INSERT statement, a temporary table—in the strictest sense—is usually populated immediately upon its materialization.

There are only two occasions when an empty global temporary table is materialized:

- A CREATE INDEX statement is issued on the table.
- A COLLECT STATISTICS statement is issued on the table.

The following table summarizes this information.

| WHEN this statement is issued on a global temporary table that has not yet been materialized … | THEN a local instance of the global temporary table is materialized and it is … |
|---|---|
| INSERT | populated with data upon its materialization. |
| CREATE INDEX … ON TEMPORARY … <br><br> COLLECT STATISTICS … ON TEMPORARY … | not populated with data upon its materialization. |

Note:  Issuing a SELECT, UPDATE, or DELETE on a global temporary table that is not materialized produces the same result as issuing a SELECT, UPDATE, or DELETE on an empty global temporary table that is materialized.

## Example

Suppose there are four sessions, Session 1, Session 2, Session 3, and Session 4 and two users, User_1 and User_2. Consider the scenario in the following two tables.

| Step | Session | Action | Result |
|------|---------|--------|--------|
| 1 | 1 | The DBA creates a global temporary table definition in the database scheme named globdb.gt1 according to the following CREATE TABLE statement:<br><br>`CREATE GLOBAL TEMPORARY`<br>`TABLE globdb.gt1,`<br>`LOG`<br>`(f1 INT NOT NULL PRIMARY`<br>`KEY,`<br>`f2 DATE,`<br>`f3 FLOAT)`<br>`ON COMMIT PRESERVE ROWS;` | The global temporary table definition is created and stored in the database schema. |
| 2 | 1 | User_1 logs on an SQL session and references globdb.gt1 using the following INSERT statement:<br><br>`INSERT globdb.gt1 (1,`<br>`980101, 11.1);` | Session 1 creates a local instance of the global temporary table definition globdb.gt1. This is also referred to as a materialized temporary table.<br><br>Immediately upon materialization, the table is populated with a single row having the following values.<br><br>f1=1<br><br>f2=980101<br><br>f3=11.1<br><br>This means that the contents of this local instance of the global temporary table definition is not empty when it is created.<br><br>From this point on, any INSERT/DELETE/UPDATE statement that references globdb.gt1 in Session 1 maps to this local instance of the table. |
| 3 | 2 | User_2 logs on an SQL session and issues the following SELECT statement.<br><br>`SELECT * FROM globdb.gt1;` | No rows are returned because Session 2 has not yet materialized a local instance of globdb.gt1. |

| Step | Session | Action | Result |
|------|---------|--------|--------|
| 4 | 2 | User_2 issues the following INSERT statement:<br><br>`INSERT globdb.gt1 (2, 980202, 22.2);` | Session 2 creates a local instance of the global temporary table definition globdb.gt1.<br><br>The table is populated, immediately upon materialization, with a single row having the following values.<br><br>f1=2<br><br>f2=980202<br><br>f3=22.2<br><br>From this point on, any INSERT/DELETE/UPDATE statement that references globdb.gt1 in Session 2 maps to this local instance of the table. |
| 5 | 2 | User_2 logs on again and issues the following SELECT statement:<br><br>`SELECT * FROM globdb.gt1;` | A single row containing the data (2, 980202, 22.2) is returned to the application. |
| 6 | 1 | User_1 logs off from Session 1. | The local instance of globdb.gt1 for Session 1 is dropped. |
| 7 | 2 | User_2 logs off from Session 2. | The local instance of globdb.gt1 for Session 2 is dropped. |

User_1 and User_2 continue their work, logging onto two additional sessions as described in the following table.

| Step | Session | Action | Result |
|------|---------|--------|--------|
| 1 | 3 | User_1 logs on another SQL session 3 and issues the following SELECT statement:<br><br>`SELECT * FROM globdb.gt1;` | No rows are returned because Session 3 has not yet materialized a local instance of globdb.gt1. |
| 2 | 3 | User_1 issues the following INSERT statement:<br><br>`INSERT globdb.gt1 (3, 980303, 33.3);` | Session 3 creates a local instance of the global temporary table definition globdb.gt1.<br><br>The table is populated, immediately upon materialization, with a single row having the following values.<br><br>f1=3<br><br>f2=980303<br><br>f3=33.3<br><br>From this point on, any INSERT/DELETE/UPDATE statement that references globdb.gt1 in Session 3 maps to this local instance of the table. |

| Step | Session | Action | Result |
|------|---------|--------|--------|
| 3 | 3 | User_1 again issues the following SELECT statement:<br><br>`SELECT * FROM globdb.gt1;` | A single row containing the data (3, 980303, 33.3) is returned to the application. |
| 4 | 4 | User_2 logs on Session 4 and issues the following CREATE INDEX statement:<br><br>`CREATE INDEX (f2) ON TEMPORARY globdb.gt1;` | An empty local global temporary table named globdb.gt1 is created for Session 4.<br><br>This is one of only two cases in which a local instance of a global temporary table is materialized without data.<br><br>The other would be a COLLECT STATISTICS statement—in this case, the following statement:<br><br>`COLLECT STATISTICS ON TEMPORARY globdb.gt1;` |
| 5 | 4 | User_2 issues the following SELECT statement:<br><br>`SELECT * FROM globdb.gt1;` | No rows are returned because the local instance of globdb.gt1 for Session 4 is empty. |
| 6 | 4 | User_2 issues the following SHOW TABLE statement:<br><br>`SHOW TABLE globdb.gt1;` | `CREATE SET GLOBAL TEMPORARY`<br>`TABLE globdb.gt1, FALLBACK,`<br>`LOG`<br>`(`<br>`f1 INTEGER NOT NULL,`<br>`f2 DATE FORMAT 'YYYY-MM-DD',`<br>`f3 FLOAT)`<br>`UNIQUE PRIMARY INDEX (f1)`<br>`ON COMMIT PRESERVE ROWS;` |
| 7 | 4 | User_2 issues the following SHOW TEMPORARY TABLE statement:<br><br>`SHOW TEMPORARY TABLE globdb.gt1;` | `CREATE SET GLOBAL TEMPORARY`<br>`TABLE globdb.gt1, FALLBACK,`<br>`LOG`<br>`(`<br>`f1 INTEGER NOT NULL,`<br>`f2 DATE FORMAT 'YYYY-MM-DD',`<br>`f3 FLOAT)`<br>`UNIQUE PRIMARY INDEX (f1)`<br>`INDEX (f2)`<br>`ON COMMIT PRESERVE ROWS;`<br><br>Note that this report indicates the new index f2 that has been created for the local instance of the temporary table. |

With the exception of a few options (see "CREATE TABLE" in *SQL Data Definition Language* for an explanation of the features not available for global temporary base tables), materialized temporary tables have the same properties as permanent tables.

After a global temporary table definition is materialized in a session, all further references to the table are made to the materialized table. No additional copies of the base definition are materialized for the session. This global temporary table is defined for exclusive use by the session whose application materialized it.

Materialized global temporary tables differ from permanent tables in the following ways:

- They are always empty when first materialized.
- Their contents cannot be shared by another session.
- The contents can optionally be emptied at the end of each transaction.
- The materialized table is dropped automatically at the end of each session.

### Limitations

You cannot use the following CREATE TABLE options for global temporary tables:

- WITH DATA
- Permanent journaling
- Referential integrity constraints

  This means that a temporary table cannot be the referencing or referenced table in a referential integrity constraint.

References to global temporary tables are not permitted in FastLoad, MultiLoad, or FastExport.

The Table Rebuild utility and the Archive/Recovery utility (with the exception of online archiving) operate on base global temporary tables only. Online archiving does not operate on temporary tables.

### Non-ANSI Extensions

Transient journaling options on the global temporary table definition are permitted using the CREATE TABLE statement.

You can modify the transient journaling and ON COMMIT options for base global temporary tables using the ALTER TABLE statement.

### Privileges Required

To materialize a global temporary table, you must have the appropriate privilege on the base global temporary table or on the containing database or user as required by the statement that materializes the table.

No access logging is performed on materialized global temporary tables, so no access log entries are generated.

## Volatile Tables

Neither the definition nor the contents of a volatile table persist across a system restart. You must use CREATE TABLE with the VOLATILE keyword to create a new volatile table each time you start a session in which it is needed.

What this means is that you can create volatile tables as you need them. Being able to create a table quickly provides you with the ability to build scratch tables whenever you need them. Any volatile tables you create are dropped automatically as soon as your session logs off.

Volatile tables are always created in the login user space, regardless of the current default database setting. That is, the database name for the table is the login user name. Space usage is

charged to login user spool space. Each user session can materialize as many as 1000 volatile tables at a time.

### Limitations

The following CREATE TABLE options are not permitted for volatile tables:

*   Permanent journaling
*   Referential integrity constraints

    This means that a volatile table cannot be the referencing or referenced table in a referential integrity constraint.
*   Check constraints
*   Compressed columns
*   DEFAULT clause
*   TITLE clause
*   Named indexes

References to volatile tables are not permitted in FastLoad or MultiLoad.

For more information, see "CREATE TABLE" in *SQL Data Definition Language.*

### Non-ANSI Extensions

Volatile tables are not defined in ANSI.

### Privileges Required

To create a volatile table, you do not need any privileges.

No access logging is performed on volatile tables, so no access log entries are generated.

### Volatile Table Maintenance Among Multiple Sessions

Volatile tables are private to a session. This means that you can log on multiple sessions and create volatile tables with the same name in each session.

However, at the time you create a volatile table, the name must be unique among all global and permanent temporary table names in the database that has the name of the login user.

For example, suppose you log on two sessions, Session 1 and Session 2. Assume the default database name is your login user name. Consider the following scenario.

| Stage | Session 1 | Session 2 | Result |
| --- | --- | --- | --- |
| 1 | Create a volatile table named VT1. | Create a volatile table named VT1. | Each session creates its own copy of volatile table VT1 using your login user name as the database. |
| 2 | Create a permanent table with an unqualified table name of VT2. | — | Session 1 creates a permanent table named VT2 using your login user name as the database. |

| Stage | Session 1 | Session 2 | Result |
|---|---|---|---|
| 3 | — | Create a volatile table named VT2. | Session 2 receives a CREATE TABLE error, because there is already a permanent table with that name. |
| 4 | Create a volatile table named VT3. | — | Session 1 creates a volatile table named VT3 using your login user name as the database. |
| 5 | — | Create a permanent table with an unqualified table name of VT3. | Session 2 creates a permanent table named VT3 using your login user name as the database.<br><br>Because a volatile table is known only to the session that creates it, a permanent table with the same name as the volatile table VT3 in Session 1 can be created as a permanent table in Session 2. |
| 6 | Insert into VT3. | — | Session 1 references volatile table VT3.<br><br>Note: Volatile tables take precedence over permanent tables in the same database in a session.<br><br>Because Session 1 has a volatile table VT3, any reference to VT3 in Session 1 is mapped to the volatile table VT3 until it is dropped (see Step 10).<br><br>On the other hand, in Session 2, references to VT3 remain mapped to the permanent table named VT3. |
| 7 | — | Create volatile table VT3. | Session 2 receives a CREATE TABLE error for attempting to create the volatile table VT3 because of the existence of that permanent table. |
| 8 | — | Insert into VT3. | Session 2 references permanent table VT3. |
| 9 | Drop VT3. | — | Session 2 drops volatile table VT3. |
| 10 | Select from VT3. | — | Session 1 references the permanent table VT3. |

## Queue Tables

Teradata Database supports queue tables, which are similar to ordinary base tables, with the additional unique property of behaving like an asynchronous first-in-first-out (FIFO) queue. Queue tables are useful for applications that want to submit queries that wait for data to be inserted into queue tables without polling.

When you create a queue table, you must define a TIMESTAMP column with a default value of CURRENT_TIMESTAMP. The values in the column indicate the time the rows were inserted into the queue table, unless different, user-supplied values are inserted.

You can then use a SELECT AND CONSUME statement, which operates like a FIFO pop:

- Data is returned from the row with the oldest timestamp in the specified queue table.
- The row is deleted from the queue table, guaranteeing that the row is processed only once.

If no rows are available, the transaction enters a delay state until one of the following occurs:

- A row is inserted into the queue table.
- The transaction aborts, either as a result of direct user intervention, such as the ABORT statement, or indirect user intervention, such as a DROP TABLE statement on the queue table.

To perform a peek on a queue table, use a SELECT statement.

For details about creating a queue table, see "CREATE TABLE (Queue Table Form)" in *SQL Data Definition Language*. For details about the SELECT AND CONSUME statement, see *SQL Data Manipulation Language*.

## Error Logging Tables

You can create an error logging table that you associate with a permanent base table when you want Teradata Database to log information about the following:

- Insert errors that occur during an SQL INSERT … SELECT operation on the permanent base table
- Update and insert errors that occur during an SQL MERGE operation on the permanent base table

To enable error logging for an INSERT … SELECT or MERGE statement, specify the LOGGING ERRORS option.

| IF a MERGE operation or INSERT … SELECT operation generates errors that … | AND the request … | THEN the error logging table contains … |
|---|---|---|
| the error logging facilities can handle | completes | • an error row for each error that the operation generated.<br>• a marker row that you can use to determine the number of error rows for the request.<br><br>The presence of the marker row means the request completed successfully. |
| | aborts and rolls back when referential integrity (RI) or unique secondary index (USI) violations are detected during index maintenance | • an error row for each USI or RI violation that was detected.<br>• an error row for each error that was detected prior to index maintenance.<br><br>The absence of a marker row in the error logging table means the request was aborted. |

| IF a MERGE operation or INSERT … SELECT operation generates errors that … | AND the request … | THEN the error logging table contains … |
|---|---|---|
| reach the error limit specified by the LOGGING ERRORS option | aborts and rolls back | an error row for each error that the operation generated, including the error that caused the error limit to be reached. |
| the error logging facilities cannot handle | aborts and rolls back | an error row for each error that the operation generated until the error that the error logging facilities could not handle. |

You can use the information in the error logging table to determine how to recover from the errors, such as which data rows to delete or correct and whether to rerun the request.

For details on how to create an error logging table, see "CREATE ERROR TABLE" in *SQL Data Definition Language*. For details on how to specify error handling for INSERT … SELECT and MERGE statements, see *SQL Data Manipulation Language*.

## No Primary Index (NoPI) Tables

For better performance when bulk loading data using Teradata FastLoad or through SQL sessions (in particular, using the INSERT statement from TPump with the ArraySupport option enabled), you can create a No Primary Index (NoPI) table to use as a staging table to load your data. Without a primary index (PI), the system can store rows on any AMP that is desired, appending the rows to the end of the table.

By avoiding the data redistribution normally associated with loading data into staging tables that have a PI, NoPI tables provide a performance benefit to applications that load data into a staging table, transform or standardize the data, and then store the converted data into another staging table.

Applications can also benefit by using NoPI tables in the following ways:

• As a log file
• As a sandbox table to store data until an appropriate indexing method is determined

A query that accesses the data in a NoPI table results in a full-table scan unless you define a secondary index on the NoPI table and use the columns that are indexed in the query.

| For more information on … | See … |
|---|---|
| creating a NoPI table | the NO PRIMARY INDEX clause for the CREATE TABLE statement in *SQL Data Definition Language*. |
| loading data into staging tables from FastLoad | *Teradata FastLoad Reference*. |
| loading data into staging tables from TPump | *Teradata Parallel Data Pump Reference*. |

| For more information on … | See … |
| --- | --- |
| primary indexes | "Primary Indexes" on page 33. |
| secondary indexes | "Secondary Indexes" on page 36. |

## Temporal Tables

Temporal tables store and maintain information with respect to time.

For more information about temporal tables, see *Temporal Table Support* and *ANSI Temporal Table Support*.

# Columns

A column is a structural component of a table and has a name and a declared type. Each row in a table has exactly one value for each column. Each value in a row is a value in the declared type of the column. The declared type includes nulls and values of the declared type.

## Defining Columns

The column definition clause of the CREATE TABLE statement defines the table column elements.

A name and a data type must be specified for each column defined for a table.

Here is an example that creates a table called employee with three columns:

```
CREATE TABLE employee
   (deptno INTEGER
   ,name CHARACTER(23)
   ,hiredate DATE);
```

Each column can be further defined with one or more optional attribute definitions. The following attributes are also elements of the SQL column definition clause:

- Data type attribute declaration, such as NOT NULL, FORMAT, TITLE, and CHARACTER SET
- COMPRESS column storage attributes clause
- DEFAULT and WITH DEFAULT default value control clauses
- PRIMARY KEY, UNIQUE, REFERENCES, and CHECK column constraint attributes clauses

Here is an example that defines attributes for the columns in the employee table:

```
CREATE TABLE employee
   (deptno INTEGER NOT NULL
   ,name CHARACTER(23) CHARACTER SET LATIN
   ,hiredate DATE DEFAULT CURRENT_DATE);
```

## System-Derived and System-Generated Columns

In addition to the table columns that you define, tables contain columns that Teradata Database generates or derives dynamically.

| Column | Description |
|---|---|
| Identity | A column that was specified with the GENERATED ALWAYS AS IDENTITY or GENERATED BY DEFAULT AS IDENTITY option in the table definition. |
| Object Identifier (OID) | For a table that has LOB columns, OID columns store pointers to subtables that store the actual LOB data. |
| PARTITION | For a table that is defined with a partitioned primary index (PPI), the PARTITION column provides the partition number of the combined partitioning expression associated with a row, where the combined partitioning expression is derived from the partitioning expressions defined for each level of the PPI. This is zero for a table that does not have a PPI. <br><br> For more information on PPIs, see "Partitioned and Nonpartitioned Primary Indexes" on page 31. |
| PARTITION#L1 through PARTITION#L62 | For tables that are defined with a multilevel PPI, these columns provide the partition number associated with the corresponding level. These are zero for a table that does not have a PPI and zero if the level is greater than the number of partitions. |
| ROWID | Contains the row identifier value that uniquely identifies the row. <br><br> For more information on row identifiers, see "Row Hash and RowID" on page 28. |

Restrictions apply to using the system-derived and system-generated columns in SQL statements. For example, you can use the keywords PARTITION and PARTITION#L1 through PARTITION#L15 in a query where a table column can be referenced, but you can only use the keyword ROWID in a CREATE JOIN INDEX statement.

### Related Topics

| For more information on … | See … |
|---|---|
| data types | "Data Types" on page 24. |
| CREATE TABLE and the column definition clause | *SQL Data Definition Language.* |
| system-derived and system-generated columns | *Database Design.* |

# Data Types

Every data value belongs to an SQL data type. For example, when you define a column in a CREATE TABLE statement, you must specify the data type of the column. Teradata Database

supports the following categories of data types. For a complete list of supported data types and detailed information about each data type, see *SQL Data Types and Literals.*

| Data Type Category | Description | Data Type Examples |
|---|---|---|
| ARRAY/ VARRAY | An ARRAY data type is used for storing and accessing multidimensional data. The ARRAY data type can store many values of the same specific data type in a sequential or matrix-like format. | • One-dimensional (1-D) ARRAY<br>• Multidimensional (n-D) ARRAY |
| Byte | Byte data types store raw data as logical bit streams. These data types are stored in the client system format and are not translated by Teradata Database. The data is transmitted directly from the memory of the client system. | • BYTE<br>• VARBYTE<br>• BLOB (Binary Large Object) |
| Character | Character data types represent characters that belong to a given character set. These data types represent character data. | • CHAR<br>• VARCHAR<br>• CLOB (Character Large Object) |
| DateTime | DateTime data types represent date, time, and timestamp values. | • DATE<br>• TIME<br>• TIMESTAMP<br>• TIME WITH TIME ZONE<br>• TIMESTAMP WITH TIME ZONE |
| Geospatial | Geospatial data types represent geographic information and provides a way for applications that manage, analyze, and display geographic information to interface with Teradata Database. | • ST_Geometry<br>• MBR |
| Interval | Interval data types represent a span of time. For example, an interval value can represent a time span that includes a number of years, months, days, hours, minutes, or seconds. | • INTERVAL YEAR<br>• INTERVAL YEAR TO MONTH<br>• INTERVAL MONTH<br>• INTERVAL DAY<br>• INTERVAL DAY TO HOUR<br>• INTERVAL DAY TO MINUTE<br>• INTERVAL DAY TO SECOND<br>• INTERVAL HOUR<br>• INTERVAL HOUR TO MINUTE<br>• INTERVAL HOUR TO SECOND<br>• INTERVAL MINUTE<br>• INTERVAL MINUTE TO SECOND<br>• INTERVAL SECOND |

| Data Type Category | Description | Data Type Examples |
|---|---|---|
| JSON | The JSON data type represents data that is in JSON (JavaScript Object Notation) format. | JSON |
| Numeric | Numeric data types represent a numeric value that is an exact numeric number (integer or decimal) or an approximate numeric number (floating point). | • BYTEINT<br>• SMALLINT<br>• INTEGER<br>• BIGINT<br>• DECIMAL/NUMERIC<br>• FLOAT/REAL/DOUBLE PRECISION<br>• NUMBER |
| Parameter | Parameter data types are data types that can be used only with input or result parameters in a function, method, stored procedure, or external stored procedure. | • TD_ANYTYPE<br>• VARIANT_TYPE |
| Period | A Period data type represents a time period, where a period is a set of contiguous time granules that extends from a beginning bound up to but not including an ending bound. | • PERIOD(DATE)<br>• PERIOD(TIME)<br>• PERIOD(TIME WITH TIME ZONE)<br>• PERIOD(TIMESTAMP)<br>• PERIOD(TIMESTAMP WITH TIME ZONE) |
| UDT | UDT (User-Defined Type) data types are custom data types that you define to model the structure and behavior of the data used by your applications. | • Distinct<br>• Structured |
| XML | The XML data type represents XML content. The data is stored in a compact binary form that preserves the information set of the XML document, including the hierarchy information and type information derived from XML validation. | XML |

### Related Topics

For detailed information on data types and the related functions, procedures, methods, and operators that operate on these types, see the following books:

- *SQL Data Types and Literals*
- *SQL Geospatial Types*
- *Teradata JSON*
- *Teradata XML*
- *SQL Functions, Operators, Expressions, and Predicates*

# Keys

A primary key is a column, or combination of columns, in a table that uniquely identifies each row in the table. The values defining a primary key for a table:

• Must be unique

• Cannot change

• Cannot be null

A foreign key is a column, or combination of columns, in a table that is also the primary key in one or more additional tables in the same database. Foreign keys provide a mechanism to link related tables based on key values.

## Keys and Referential Integrity

Teradata Database uses primary and foreign keys to maintain referential integrity. For additional information, see "Referential Integrity" on page 46.

## Effect on Row Distribution

Because Teradata Database uses a unique primary or secondary index to enforce a primary key, the primary key can affect how Teradata Database distributes and retrieves rows. For more information, see "Primary Indexes" on page 33 and "Secondary Indexes" on page 36.

## Primary Keys and Primary Indexes

The following table summarizes the differences between keys and indexes, using the primary key and primary index for purposes of comparison.

| Primary Key | Primary Index |
|---|---|
| Important element of logical data model. | Not used in logical data model. |
| Used to maintain referential integrity. | Used to distribute and retrieve data. |
| Must be unique to identify each row. | Can be unique or nonunique. |
| Values cannot change. | Values can change. |
| Cannot be null. | Can be null. |
| Does not imply access path. | Defines the most common access path. |
| Not required for physical table definition. | Required for physical table definition. |

# Indexes

An index is a mechanism that the SQL query optimizer can use to make table access more efficient. Indexes enhance data access by providing a more-or-less direct path to stored data to

avoid performing full table scans to locate the small number of rows you typically want to retrieve or update.

The Teradata Database parallel architecture makes indexing an aid to better performance, not a crutch necessary to ensure adequate performance. Full table scans are not something to be feared in the Teradata Database environment. This means that the sorts of unplanned, ad hoc queries that characterize the data warehouse process, and that often are not supported by indexes, perform very effectively for Teradata Database using full table scans.

## Classic Indexes and Teradata Database Indexes

The classic index for a relational database is itself a file made up of rows having these parts:

- A (possibly unique) data field in the referenced table.
- A pointer to the location of that row in the base table (if the index is unique) or a pointer to all possible locations of rows with that data field value (if the index is nonunique).

Because Teradata Database is a massively parallel architecture, it requires a more efficient means of distributing and retrieving its data. One such method is hashing. All Teradata Database indexes are based on row hash values rather than raw table column values, even though secondary, hash, and join indexes can be stored in order of their values to make them more useful for satisfying range conditions.

## Selectivity of Indexes

An index that retrieves many rows is said to have weak selectivity.

An index that retrieves few rows is said to be strongly selective.

The more strongly selective an index is, the more useful it is. In some cases, it is possible to link together several weakly selective nonunique secondary indexes by bit mapping them. The result is effectively a strongly selective index and a dramatic reduction in the number of table rows that must be accessed.

For more information on linking weakly selective secondary indexes into a strongly selective unit using bit mapping, see .

## Row Hash and RowID

Teradata Database table rows are self-indexing with respect to their primary index and so require no additional storage space. When a row is inserted into a table, Teradata Database stores the 32-bit row hash value of the primary index with it.

Because row hash values are not necessarily unique, Teradata Database also generates a unique 32-bit numeric value (called the Uniqueness Value) that it appends to the row hash value, forming a unique RowID. This RowID makes each row in a table uniquely identifiable and ensures that hash collisions do not occur.

If a table is defined with a partitioned primary index (PPI), the RowID also includes the combined partition number to which the row is assigned, where the combined partition number is derived from the partition numbers for each level of the PPI. For more information on PPIs, see .

| For tables with no PPI … | For tables with a PPI … |
|---|---|
| The first row having a specific row hash value is always assigned a uniqueness value of 1, which becomes the current high uniqueness value. Each time another row having the same row hash value is inserted, the current high value increments by 1, and that value is assigned to the row. | The first row having a specific combined partition number and row hash value is always assigned a uniqueness value of 1, which becomes the highest current uniqueness value. Each time another row having the same combined partition number and row hash value is inserted, the current high value increments by 1, and that value is assigned to the row. |
| Table rows having the same row hash value are stored on disk sorted in the ascending order of RowID.<br><br>Uniqueness values are not reused except for the special case in which the highest valued row within a row hash is deleted from a table. | Table rows having the same combined partition number and row hash value are stored on disk sorted in the ascending order of RowID.<br><br>Uniqueness values are not reused except for the special case in which the highest valued row within a combined partition number and row hash is deleted from a table. |

A RowID for a row might change, for instance, when a primary index or partitioning column is changed, or when there is complex update of the table.

## Index Hash Mapping

Rows are distributed across the AMPS using a hashing algorithm that computes a row hash value based on the primary index. The row hash is a 32-bit value. Depending on the system setting for the hash bucket size, either the higher-order 16 bits or the higher-order 20 bits of a hash value determine an associated hash bucket.

Normally, the hash bucket size is 20 bits for new systems. If you are upgrading from an older release, the hash bucket size may be 16 bits. If a 20-bit hash bucket size is more appropriate for the size of a system, you can use the DBS Control Utility to change the system setting for the hash bucket size. For details, see "DBS Control utility" in *Utilities*.

| IF the hash bucket size is … | THEN the number of hash buckets is … |
|---|---|
| 16 bits | 65536. |
| 20 bits | 1048576. |

The hash buckets are distributed as evenly as possible among the AMPs on a system.

Teradata Database maintains a hash map—an index of which hash buckets live on which AMPs—that it uses to determine whether rows belong to an AMP based on their row hash values. Row assignment is performed in a manner that ensures as equal a distribution as possible among all the AMPs on a system.

## Advantages of Indexes

The intent of indexes is to lessen the time it takes to retrieve rows from a database. The faster the retrieval, the better.

## Disadvantages of Indexes

- They must be updated every time a row is updated, deleted, or added to a table.

  This is only a consideration for indexes other than the primary index in the Teradata Database environment. The more indexes you have defined for a table, the bigger the potential update downside becomes.

  Because of this, secondary, join, and hash indexes are rarely appropriate for OLTP situations.

- All Teradata Database secondary indexes are stored in subtables, and join and hash indexes are stored in separate tables, exerting a burden on system storage space.

- When FALLBACK is defined for a table, a further storage space burden is created because secondary index subtables are always duplicated whenever FALLBACK is defined for a table. An additional burden on system storage space is exerted when FALLBACK is defined for join indexes or hash indexes or both.

For this reason, it is extremely important to use the EXPLAIN modifier to determine optimum data manipulation statement syntax and index usage before putting statements and indexes to work in a production environment. For more information on EXPLAIN, see *SQL Data Manipulation Language.*

## Teradata Database Index Types

- Primary index

  In general, all Teradata Database tables require a primary index because the system distributes tables on their primary indexes. Primary indexes can be:

  - Unique or nonunique
  - Partitioned or nonpartitioned

- Secondary index

  Secondary indexes can be unique or nonunique.

- Join index (JI)

- Hash index

## Unique Indexes

A unique index, like a primary key, has a unique value for each row in a table.

Teradata Database defines two different unique indexes:

- Unique primary index (UPI)

  UPIs provide optimal data distribution and are typically assigned to the primary key for a table. When a NUPI makes better sense for a table, then the primary key is frequently assigned to be a USI.

- Unique secondary index (USI)

  USIs guarantee that each complete index value is unique, while ensuring that data access based on it is always a two-AMP operation.

## Nonunique Indexes

A nonunique index does not require its values to be unique. There are occasions when a nonunique index is the best choice as the primary index for a table.

NUSIs are also very useful for many decision support situations.

## Partitioned and Nonpartitioned Primary Indexes

Primary indexes can be partitioned or nonpartitioned.

A nonpartitioned primary index (NPPI) is the traditional primary index by which rows are assigned to AMPs.

A partitioned primary index (PPI) allows rows to be partitioned, based on some set of columns, on the AMP to which they are distributed, and ordered by the hash of the primary index columns within the partition.

A PPI can improve query performance through partition elimination. A PPI provides a useful alternative to an NPPI for executing range queries against a table, while still providing efficient access, join, and aggregation strategies on the primary index.

A multilevel PPI allows each partition at a level to be subpartitioned based on a partitioning expression, where the maximum number of levels is 15.

A multilevel PPI provides multiple access paths to the rows in the base table and can improve query performance through partition elimination at each of the various levels or combination of levels.

A PPI can only be defined as unique if all the partitioning columns are included in the set of primary index columns.

## Join Indexes

A join index is an indexing structure containing columns from one or more base tables and is generally used to resolve queries and eliminate the need to access and join the base tables it represents.

Teradata Database join indexes can be defined in the following general ways.

- Simple or aggregate
- Single-table or multitable
- Hash-ordered or value-ordered
- Complete or sparse

For details, see .

## Hash Indexes

Hash indexes are used for the same purposes as are single-table join indexes, and are less complicated to define. However, a join index offers more choices.

For additional information, see "Hash Indexes" on page 44.

## Creating Indexes For a Table

Use the CREATE TABLE statement to define a primary index and one or more secondary indexes. (You can also define secondary indexes using the CREATE INDEX statement.) You can define the primary index (and any secondary index) as unique, depending on whether duplicate values are to be allowed in the indexed column set. A partitioned primary index cannot be defined as unique if one or more partitioning columns are not included in the primary index.

To create hash or join indexes, use the CREATE HASH INDEX and CREATE JOIN INDEX statements, respectively.

## Determining the Usefulness of Indexes

The selection of indexes to support a query is not under user control. You cannot provide the Teradata Database query optimizer with pragmas or hints, nor can you specify resource options or control index locking.

The only references made to indexes in the SQL language concern their definition and not their use. Teradata SQL data manipulation language statements do not provide for any specification of indexes.

There are several implications of this behavior.

- To ensure that the optimizer has access to current information about how to best optimize any query or update made to the database, you must use the COLLECT STATISTICS statement to collect statistics regularly on all indexed columns, all frequently joined columns, and all columns frequently specified in query predicates.

- To ensure that your queries access your data in the most efficient manner possible:

  - Use the EXPLAIN request modifier or the Teradata Visual Explain client utility to try out various candidate queries or updates and to note which indexes are used by the optimizer in their execution (if any) as well as to examine the relative cost of the operation.

  - Use the Teradata Index Wizard client utility to recommend and validate sets of secondary indexes, single-table join indexes, and PPIs for a given query workload.

| For more information on … | See … |
|---|---|
| using the EXPLAIN request modifier | *SQL Data Manipulation Language* |
| using the Teradata Visual Explain client utility | *Teradata Visual Explain User Guide* |

| For more information on … | See … |
|---|---|
| additional performance-related information about how to use the access and join plan reports produced by EXPLAIN to optimize the performance of your databases | *Database Design* |
| using the Teradata Index Wizard client utility | • *Teradata Index Wizard User Guide* <br> • *SQL Request and Transaction Processing* |
| collecting and maintaining accurate database statistics | "COLLECT STATISTICS" in *SQL Data Definition Language* |

# Primary Indexes

The primary index for a table controls the distribution and retrieval of the data for that table across the AMPs. Both distribution and retrieval of the data is controlled using the Teradata Database hashing algorithm (see "Row Hash and RowID" on page 28).

If the primary index is defined as a partitioned primary index (PPI), the data is partitioned, based on some set of columns, on each AMP, and ordered by the hash of the primary index columns within the partition.

Data accessed based on a primary index is always a one-AMP operation because a row and its index are stored on the same AMP. This is true whether the primary index is unique or nonunique, and whether it is partitioned or nonpartitioned.

## Primary Index Assignment

In general, most Teradata Database tables require a primary index. (Some tables in the Data Dictionary do not have primary indexes and a global temporary trace table is not allowed to have a primary index.) To create a primary index, use the CREATE TABLE statement.

If you do not assign a primary index explicitly when you create a table, Teradata Database assigns a primary index, based on the following rules.

| Primary Index | Primary Key | Unique Column Constraint | Teradata Database selects the … |
|---|---|---|---|
| no | YES | no | primary key column set to be a UPI. |
| no | no | YES | first column or columns having a UNIQUE constraint to be a UPI. |
| no | YES | YES | primary key column set to be a UPI. |
| no | no | no | first column defined for the table to be a NUPI. <br><br> If the data type of the first column in the table is a LOB, then the CREATE TABLE operation aborts and the system returns an error message. |

In general, the best practice is to specify a primary index instead of having Teradata Database select a default primary index.

## Uniform Distribution of Data and Optimal Access Considerations

When choosing the primary index for a table, there are two essential factors to keep in mind: uniform distribution of the data and optimal access.

With respect to uniform data distribution, consider the following factors:

- The more distinct the primary index values, the better.
- Rows having the same primary index value are distributed to the same AMP.
- Parallel processing is more efficient when table rows are distributed evenly across the AMPs.

With respect to optimal data access, consider the following factors:

- Choose the primary index on the most frequently used access path.
  For example:
  - If rows are generally accessed by a range query, consider defining a PPI on the table that creates a useful set of partitions.
  - If the table is frequently joined with a specific set of tables, consider defining the primary index on the column set that is typically used as the join condition.
- Primary index operations must provide the full primary index value.
- Primary index retrievals on a single value are always one-AMP operations.

Although it is true that the columns you choose to be the primary index for a table are often the same columns that define the primary key, it is also true that primary indexes often comprise fields that are neither unique nor components of the primary key for the table.

## Unique and Nonunique Primary Index Considerations

In addition to uniform distribution of data and optimal access considerations, other guidelines and performance considerations apply to selecting a unique or a nonunique column set as the primary index for a table.

Other considerations can include:

- Primary and other alternate key column sets
- The value range seen when using predicates in a WHERE clause
- Whether access can involve multiple rows or a spool file or both

For more information on criteria for selecting a primary index, see *Database Design*.

## Partitioning Considerations

The decision to define a single-level or multilevel Partitioned Primary Index (PPI) for a table depends on how its rows are most frequently accessed. PPIs are designed to optimize range queries while also providing efficient primary index join strategies and may be appropriate for

other classes of queries. Performance of such queries is improved by accessing only the rows of the qualified partitions.

A PPI increases query efficiency by avoiding full table scans without the overhead and maintenance costs of secondary indexes.

The most important factors for PPIs are accessibility and maximization of partition elimination. In all cases, it is critical for parallel efficiency to define a primary index that distributes the rows of the table fairly evenly across the AMPs.

For more information on partitioning considerations, including information on column partitioning, called Teradata Columnar, see *Database Design*, *SQL Data Definition Language Detailed Topics*, and *SQL Data Manipulation Language*.

## Restrictions

Restrictions apply to the columns you choose to be the primary index for a table. For partitioned primary indexes, further restrictions apply to the columns you choose to be partitioning columns. Here are some of the restrictions:

- A primary index column or partitioning column cannot be a column that has a CLOB, BLOB, ST_Geometry, MBR, or Period data type.
- Partitioning expressions for single-level PPIs may only use the following general forms:
  - Column (must be INTEGER or a data type that casts to INTEGER)
  - Expressions based on one or more columns, where the expression evaluates to INTEGER or a data type that casts to INTEGER
  - The CASE_N and RANGE_N functions
- Partitioning expressions of a multilevel PPI may only specify CASE_N and RANGE_N functions.
- You cannot compress columns that are members of the primary index column set or are partitioning columns.

Other restrictions apply to the partitioning expression for PPIs. For example:

- Comparison of character data where the server character set is KANJI1 or KANJISJIS is not allowed.
- The expression for the RANGE_N test value must evaluate to BYTEINT, SMALLINT, INTEGER, DATE, CHAR, VARCHAR, GRAPHIC or VARGRAPHIC.
- Nondeterministic partitioning expressions are not allowed, including cases that might not report errors, such as casting TIMESTAMP to DATE, because of the potential for wrong results.

For details on all the restrictions that apply to primary indexes and partitioned primary indexes, see CREATE TABLE in *SQL Data Definition Language*.

## Primary Index Properties

- Defined with the CREATE TABLE data definition statement.
  CREATE INDEX is used only to create secondary indexes.

- Modified with the ALTER TABLE data definition statement.

  Some modifications, such as partitioning and primary index columns, require an empty table.

- Automatically assigned by CREATE TABLE if you do not explicitly define a primary index. However, the best practice is to always specify the primary index, because the default may not be appropriate for the table.

- Can be composed of as many as 64 columns.

- A maximum of one can be defined per table.

- Can be partitioned or nonpartitioned.

  Partitioned primary indexes are not automatically assigned. You must explicitly define a partitioned primary index.

- Can be unique or nonunique.

  Note that a partitioned primary index can only be unique if all the partitioning columns are also included as primary index columns. If the primary index does not include all the partitioning columns, uniqueness on the primary index columns may be enforced with a unique secondary index on the same columns as the primary index.

- Defined as nonunique if the primary index is not defined explicitly as unique or if the primary index is specified for a single column SET table.

- Controls data distribution and retrieval using the Teradata Database hashing algorithm.

- Improves performance when used correctly in the WHERE clause of an SQL data manipulation statement to perform the following actions.

  - Single-AMP retrievals
  - Joins between tables with identical primary indexes, the optimal scenario
  - Partition elimination when the primary index is partitioned

## Related Topics

For more information on using primary indexes to enhance the performance of your databases, see *Database Design.*

# Secondary Indexes

Secondary indexes are never required for Teradata Database tables, but they can often improve system performance.

You create secondary indexes explicitly using the CREATE TABLE and CREATE INDEX statements. Teradata Database can implicitly create unique secondary indexes; for example, when you use a CREATE TABLE statement that specifies a primary index, Teradata Database implicitly creates unique secondary indexes on column sets that you specify using PRIMARY KEY or UNIQUE constraints.

Creating a secondary index causes Teradata Database to build a separate internal subtable to contain the index rows, thus adding another set of rows that requires updating each time a table row is inserted, deleted, or updated.

Nonunique secondary indexes (NUSIs) can be specified as either hash-ordered or value-ordered. Value-ordered NUSIs are limited to a single numeric-valued (including DATE) sort key whose size is four or fewer bytes.

Secondary index subtables are also duplicated whenever a table is defined with FALLBACK.

After the table is created and usage patterns have developed, additional secondary indexes can be defined with the CREATE INDEX statement.

## Unique and Nonunique Secondary Indexes

Teradata Database processes USIs and NUSIs very differently.

Consider the following statements that define a USI and a NUSI.

| Secondary Index | Statement |
|---|---|
| USI | `CREATE UNIQUE INDEX (customer_number)`<br>`ON customer_table;` |
| NUSI | `CREATE INDEX (customer_name)`<br>`ON customer_table;` |

The following table highlights differences in the build process for the preceding statements.

| USI Build Process | NUSI Build Process |
|---|---|
| Each AMP accesses its subset of the base table rows. | Each AMP accesses its subset of the base table rows. |
| Each AMP copies the secondary index value and appends the RowID for the base table row. | Each AMP builds a spool file containing each secondary index value found followed by the RowID for the row it came from. |
| Each AMP creates a Row Hash on the secondary index value and puts all three values onto the BYNET. | For hash-ordered NUSIs, each AMP sorts the RowIDs for each secondary index value into ascending order.<br><br>For value-ordered NUSIs, the rows are sorted by NUSI value order. |

| USI Build Process | NUSI Build Process |
|---|---|
| The appropriate AMP receives the data and creates a row in the index subtable. | For hash-ordered NUSIs, each AMP creates a row hash value for each secondary index value on a local basis and creates a row in its portion of the index subtable. |
| If the AMP receives a row with a duplicate index value, an error is reported. | For value-ordered NUSIs, storage is based on NUSI value rather than the row hash value for the secondary index. |
| | Each row contains one or more RowIDs for the index value. |

Consider the following statements that access a USI and a NUSI.

| Secondary Index | Statement |
|---|---|
| USI | `SELECT * FROM customer_table`<br>`WHERE customer_number=12;` |
| NUSI | `SELECT * FROM customer_table`<br>`WHERE customer_name = 'SMITH';` |

The following table identifies differences for the access process of the preceding statements.

| USI Access Process | NUSI Access Process |
|---|---|
| The supplied index value hashes to the corresponding secondary index row. | A message containing the secondary index value is broadcast to every AMP. |
| The retrieved base table RowID is used to access the specific data row. | For a hash-ordered NUSI, each AMP creates a local row hash and uses it to access its portion of the index subtable to see if a corresponding row exists. |
| | Value-ordered NUSI index subtable values are scanned only for the range of values specified by the query. |
| The process is complete.<br>This is typically a two-AMP operation. | If an index row is found, the AMP uses the RowID or value order list to access the corresponding base table rows. |
| | The process is complete.<br>This is always an all-AMP operation, with the exception of a NUSI that is defined on the same columns as the primary index. |

Note:  The NUSI is not used if the estimated number of rows to be read in the base table is equal to or greater than the estimated number of data blocks in the base table; in this case, a full table scan is done, or, if appropriate, partition scans are done.

## NUSIs and Covering

The Optimizer aggressively pursues NUSIs when they cover a query. Covered columns can be specified anywhere in the query, including the select list, the WHERE clause, aggregate functions, GROUP BY clauses, expressions, and so on. Presence of a WHERE condition on each indexed column is not a prerequisite for using a NUSI to cover a query.

## Value-Ordered NUSIs

Value-ordered NUSIs are very efficient for range conditions, and more so when strongly selective or when combined with covering. Because the NUSI rows are sorted by data value, it is possible to search only a portion of the index subtable for a given range of key values.

Value-ordered NUSIs have the following limitations.

- The sort key is limited to a single numeric or DATE column.
- The sort key column must be four or fewer bytes.

The following query is an example of the sort of SELECT statement for which value-ordered NUSIs were designed.

```
SELECT *
FROM Orders
WHERE o_date BETWEEN DATE '1998-10-01' AND DATE '1998-10-07';
```

## Multiple Secondary Indexes and Composites

Database designers frequently define multiple secondary indexes on a table.

For example, the following statements define two secondary indexes on the EMPLOYEE table:

```
CREATE INDEX (department_number) ON EMPLOYEE;
CREATE INDEX (job_code) ON EMPLOYEE;
```

The WHERE clause in the following query specifies the columns that have the secondary indexes defined on them:

```
SELECT last_name, first_name, salary_amount
FROM employee
WHERE department_number = 500
AND job_code = 2147;
```

Whether the Optimizer chooses to include one, all, or none of the secondary indexes in its query plan depends entirely on their individual and composite selectivity.

For more information on multiple and composite secondary index access, and other aspects of index selection, see *Database Design*.

## NUSI Bit Mapping

Bit mapping is a technique used by the Optimizer to effectively link several weakly selective indexes in a way that creates a result that drastically reduces the number of base rows that must be accessed to retrieve the desired data. The process determines common rowIDs among multiple NUSI values by means of the logical intersection operation.

Bit mapping is significantly faster than the three-part process of copying, sorting, and comparing rowID lists. Additionally, the technique dramatically reduces the number of base table I/Os required to retrieve the requested rows.

| For more information on … | See … |
| --- | --- |
| when Teradata Database performs NUSI bit mapping | *Database Design* |
| how NUSI bit maps are computed | |
| using the EXPLAIN modifier to determine if bit mapping is being used for your indexes | • *Database Design*<br>• *SQL Data Manipulation Language* |

## Secondary Index Properties

- Can enhance the speed of data retrieval.

  Because of this, secondary indexes are most useful in decision support applications.
- Do not affect data distribution.
- Can be a maximum of 32 defined per table.
- Can be composed of as many as 64 columns.
- For a value-ordered NUSI, only a single numeric or DATE column of four or fewer bytes may be specified for the sort key.
- For a hash-ordered covering index, only a single column may be specified for the hash ordering.
- Can be created or dropped dynamically as data usage changes or if they are found not to be useful for optimizing data retrieval performance.
- Require additional disk space to store subtables.
- Require additional I/Os on inserts and deletes.

  Because of this, secondary indexes might not be as useful in OLTP applications.
- Should not be defined on columns whose values change frequently.
- Should not include columns that do not enhance selectivity.
- Should not use composite secondary indexes when multiple single column indexes and bit mapping might be used instead.
- Composite secondary index is useful if it reduces the number of rows that must be accessed.
- The Optimizer does not use composite secondary indexes unless there are explicit values for each column in the index.
- Most efficient for selecting a small number of rows.
- Can be unique or nonunique.
- NUSIs can be hash-ordered or value-ordered, and can optionally include covering columns.
- Cannot be partitioned, but can be defined on a table with a partitioned primary index.

## USI and NUSI Properties

| USI | NUSI |
|-----|------|
| • Guarantee that each complete index value is unique.<br>• Any access using the index is a two-AMP operation. | • Useful for locating rows having a specific value in the index.<br>• Can be hash-ordered or value-ordered.<br>Value-ordered NUSIs are particularly useful for enhancing the performance of range queries.<br>• Can include covering columns.<br>• Any access using the index is an all-AMP operation. |

## Related Topics

See "SQL Data Definition Language Statement Syntax" of *SQL Data Definition Language* under "CREATE TABLE" and "CREATE INDEX" for more information.

# Join Indexes

Join indexes are not indexes in the usual sense of the word. They are file structures designed to permit queries (join queries in the case of multitable join indexes) to be resolved by accessing the index instead of having to access and join their underlying base tables.

You can use join indexes to:

• Define a prejoin table on frequently joined columns (with optional aggregation) without denormalizing the database.

• Create a full or partial replication of a base table with a primary index on a foreign key column table to facilitate joins of very large tables by hashing their rows to the same AMP as the large table.

• Define a summary table without denormalizing the database.

You can define a join index on one or several tables.

Depending on how the index is defined, join indexes can also be useful for queries where the index structure contains only some of the columns referenced in the statement. This situation is referred to as a partial cover of the query.

Unlike traditional indexes, join indexes do not implicitly store pointers to their associated base table rows. Instead, they are generally used as a quick access method that eliminates the need to access and join the base tables they represent. They substitute for rather than point to base table rows. The only exception to this is the case where an index partially covers a query. If the index is defined using either the ROWID keyword or the UPI or USI of its base table as one of its columns, then it can be used to join with the base table to cover the query.

## Defining Join Indexes

To create a join index, use the CREATE JOIN INDEX statement.

For example, suppose that a common task is to look up customer orders by customer number and date. You might create a join index like the following, linking the customer table, the order table, and the order detail table:

```
CREATE JOIN INDEX cust_ord2
AS SELECT cust.customerid,cust.loc,ord.ordid,item,qty,odate
FROM cust, ord, orditm
WHERE cust.customerid = ord.customerid
AND ord.ordid = orditm.ordid;
```

## Multitable Join Indexes

A multitable join index stores and maintains the joined rows of two or more tables and, optionally, aggregates selected columns.

Multitable join indexes are for join queries that are performed frequently enough to justify defining a prejoin on the joined columns.

A multitable join index is useful for queries where the index structure contains all the columns referenced by one or more joins, thereby allowing the index to cover that part of the query, making it possible to retrieve the requested data from the index rather than accessing its underlying base tables. For obvious reasons, an index with this property is often referred to as a covering index.

## Single-Table Join Indexes

Single-table join indexes are very useful for resolving joins on large tables without having to redistribute the joined rows across the AMPs.

Single-table join indexes facilitate joins by hashing a frequently joined subset of base table columns to the same AMP as the table rows to which they are frequently joined. This enhanced geography eliminates BYNET traffic as well as often providing a smaller sized row to be read and joined.

## Aggregate Join Indexes

When query performance is of utmost importance, aggregate join indexes offer an extremely efficient, cost-effective method of resolving queries that frequently specify the same aggregate operations on the same column or columns. When aggregate join indexes are available, the system does not have to repeat aggregate calculations for every query.

You can define an aggregate join index on two or more tables, or on a single table. A single-table aggregate join index includes a summary table with:

- A subset of columns from a base table
- Additional columns for the aggregate summaries of the base table columns

## Sparse Join Indexes

You can create join indexes that limit the number of rows in the index to only those that are accessed when, for example, a frequently run query references only a small, well known subset

of the rows of a large base table. By using a constant expression to filter the rows included in the join index, you can create what is known as a sparse index.

Any join index, whether simple or aggregate, multitable or single-table, can be sparse.

To create a sparse index, use the WHERE clause in the CREATE JOIN INDEX statement.

## Effects of Join Indexes

- Load Utilities

  MultiLoad and FastLoad utilities cannot be used to load or unload data into base tables that have a join index defined on them because join indexes are not maintained during the execution of these utilities. If an error occurs because of a join index, take these steps:

  - Ensure that any queries that use the join index are not running.
  - Drop the join index. (The system defers completion of this step until there are no more queries running that use the join index.)
  - Load the data into the base table.
  - Recreate the join index.

  The TPump utility, which performs standard SQL row inserts and updates, can be used to load or unload data into base tables with join indexes because it properly maintains join indexes during execution. However, in some cases, performance may improve by dropping join indexes on the table prior to the load and recreating them after the load.

- ARC (Archive/Recovery Utility)

  Archive and recovery cannot be used on a join index itself. Archiving is permitted on a base table or database that has an associated join index defined. Before a restore of such a base table or database, you must drop the existing join index definition. Before using any such index again in the execution of queries, you must recreate the join index definition.

- Permanent Journal Recovery

  Using a permanent journal to recover a base table (that is, ROLLBACK or ROLLFORWARD) with an associated join index defined is permitted. The join index is not automatically rebuilt during the recovery process. Instead, it is marked as nonvalid and it must be dropped and recreated before it can be used again in the execution of queries.

## Join Indexes and Base Tables

In most respects, a join index is similar to a base table. For example, you can do the following things to a join index:

- Create nonunique secondary indexes on its columns.
- Create a unique primary index on its columns, provided it is a non-compressed and nonvalue-ordered single-table join index.
- Execute COLLECT STATISTICS, DROP STATISTICS, HELP, and SHOW statements.
- Partition its primary index, if it is a noncompressed join index.

Unlike base tables, you cannot do the following things with join indexes:

- Query or update join index rows explicitly.

- Store and maintain arbitrary query results such as expressions.

  Note:  You can maintain aggregates or sparse indexes if you define the join index to do so.

## Related Topics

| For more information on … | See … |
|---|---|
| creating join indexes | "CREATE JOIN INDEX" in *SQL Data Definition Language.* |
| dropping join indexes | "DROP JOIN INDEX" in *SQL Data Definition Language.* |
| displaying the attributes of the columns defined by a join index | "HELP JOIN INDEX" in *SQL Data Definition Language.* |
| using join indexes to enhance the performance of your databases | • *Database Design.* <br> • *SQL Data Definition Language.* |
| • database design considerations for join indexes <br> • improving join index performance | *Database Design.* |

# Hash Indexes

Hash indexes are used for the same purposes as single-table join indexes. The following table lists the principal differences between hash indexes and single-table join indexes.

| Hash Index | Single-Table Join Index |
|---|---|
| Column list cannot contain aggregate or ordered analytical functions. | Column list can contain aggregate functions. |
| Cannot have a unique primary index. | A non-compressed and nonvalue-ordered single-table join index can have a unique primary index. |
| Cannot have a secondary index. | Can have a secondary index. |
| Supports transparently added, system-defined columns that point to the underlying base table rows. | Does not implicitly add underlying base table row pointers. <br><br> Pointers to underlying base table rows can be created explicitly by defining one element of the column list using the ROWID keyword or the UPI or USI of the base table. |
| Cannot be defined on a NoPI table. | Can be defined on a NoPI table. |

Hash indexes are useful for creating a full or partial replication of a base table with a primary index on a foreign key column to facilitate joins of very large tables by hashing them to the same AMP.

You can define a hash index on one table only. The functionality of hash indexes is a subset to that of single-table join indexes.

## Hash and Single-Table Join Indexes

The reasons for using hash indexes are similar to those for using single-table join indexes. Not only can hash indexes optionally be specified to be distributed in such a way that their rows are AMP-local with their associated base table rows, they also implicitly provide an alternate direct access path to those base table rows. This facility makes hash indexes somewhat similar to secondary indexes in function. Hash indexes are also useful for covering queries so that the base table need not be accessed at all.

The following list summarizes the similarities of hash and single-table join indexes:

- Primary function of both is to improve query performance.
- Both are maintained automatically by the system when the relevant columns of their base table are updated by a DELETE, INSERT, UPDATE, or MERGE statement.
- Both can be the object of any of the following SQL statements:
    - COLLECT STATISTICS
    - DROP STATISTICS
    - HELP INDEX
    - SHOW
- Both receive their space allocation from permanent space and are stored in distinct tables.
- The storage organization for both supports a compressed format to reduce storage space, but for a hash index, Teradata Database makes this decision.
- Both can be FALLBACK protected.
- Neither can be queried or directly updated.
- Neither can store an arbitrary query result.
- Both share the same restrictions for use with the MultiLoad, FastLoad, and Archive/ Recovery utilities.
- A hash index implicitly defines a direct access path to base table rows. A join index may be explicitly specified to define a direct access path to base table rows.

## Effects of Hash Indexes

Hash indexes affect Teradata Database functions and features the same way join indexes affect Teradata Database functions and features. For details, see "Effects of Join Indexes" on page 43.

## Queries Using a Hash Index

In most respects, a hash index is similar to a base table. For example, you can perform COLLECT STATISTICS, DROP STATISTICS, HELP, and SHOW statements on a hash index.

Unlike base tables, you cannot do the following things with hash indexes:

- Query or update hash index rows explicitly.
- Store and maintain arbitrary query results such as expressions.

- Create explicit unique indexes on its columns.
- Partition the primary index of the hash index.

## Related Topics

| For information on … | See … |
|---|---|
| using CREATE HASH INDEX to create a hash index | *SQL Data Definition Language.* |
| using DROP HASH INDEX to drop a hash index | |
| using HELP HASH INDEX to display the data types of the columns defined by a hash index | |
| database design considerations for hash indexes | *Database Design.* |

Consult the following books for more detailed information on using hash indexes to enhance the performance of your databases:

- *Database Design*
- *SQL Data Definition Language*

# Referential Integrity

Referential integrity (RI) is defined as all the following notions.

- The concept of relationships between tables, based on the definition of a primary key (or UNIQUE alternate key) and a foreign key.
- A mechanism that provides for specification of columns within a referencing table that are foreign keys for columns in some other referenced table.

  Referenced columns must be defined as one of the following.

  - Primary key columns
  - Unique columns

- A reliable mechanism for preventing accidental database corruption when performing inserts, updates, and deletes.

Referential integrity requires that a row having a value that is not null for a referencing column cannot exist in a table if an equal value does not exist in a referenced column.

## Referential Integrity Enforcement

Teradata Database supports two forms of declarative SQL for enforcing referential integrity:

- A standard method that enforces RI on a row-by-row basis
- A batch method that enforces RI on a statement basis

Both methods offer the same measure of integrity enforcement, but perform it in different ways.

A third form, sometimes informally referred to as soft RI, is related to these because it provides a declarative definition for a referential relationship, but it does not enforce that relationship. Enforcement of the declared referential relationship is left to the user by any appropriate method.

## Referencing (Child) Table

The referencing table is referred to as the child table, and the specified child table columns are the referencing columns.

Note:  Referencing columns must have the same numbers and types of columns, data types, and sensitivity as the referenced table keys. Column-level constraints are not compared and, for standard referential integrity, compression is not allowed on either referenced or referencing columns.

## Referenced (Parent) Table

A child table must have a parent, and the referenced table is referred to as the parent table.

The parent key columns in the parent table are the referenced columns.

For standard and batch RI, the referenced columns must be one of the following unique indexes:

- A unique primary index (UPI), defined as NOT NULL
- A unique secondary index (USI), defined as NOT NULL

Soft RI does not require any index on the referenced columns.

## Terminology

| Term | Definition |
| --- | --- |
| Child Table | A table where the referential constraints are defined.<br>Child table and referencing table are synonyms. |
| Parent Table | The table referenced by a child table.<br>Parent table and referenced table are synonyms. |
| Primary Key | A unique identifier for a row of a table. |
| UNIQUE Alternate Key | |
| Foreign Key | A column set in the child table that is also the primary key (or a UNIQUE alternate key) in the parent table.<br>Foreign keys can consist of as many as 64 different columns. |

| Term | Definition |
|---|---|
| Referential Constraint | A constraint defined on a column set or a table to ensure referential integrity. <br><br> For example, consider the following table definition: <br><br> `CREATE TABLE A` <br> `(A1 CHAR(10) REFERENCES B (B1), /* 1 */` <br> ` A2 INTEGER` <br> `FOREIGN KEY (A1,A2) REFERENCES C /* 2 */` <br> `PRIMARY INDEX (A1));` <br><br> This CREATE TABLE statement specifies the following referential integrity constraints. <br><br> • Constraint 1 is defined at the column level. Implicit foreign key A1 references the parent key B1 in table B. <br><br> • Constraint 2 is defined at the table level. <br><br> Explicit composite foreign key (A1, A2) implicitly references the UPI (or a USI) of parent table C, which must be two columns, the first typed CHAR(10) and the second typed INTEGER. <br><br> Both parent table columns must also be defined as NOT NULL. |

## Why Referential Integrity Is Important

Consider the employee and payroll tables for any business.

With referential integrity constraints, the two tables work together as one. When one table gets updated, the other table also gets updated.

The following case depicts a useful referential integrity scenario.

Looking for a better career, Mr. Clark Johnson leaves his company. Clark Johnson is deleted from the employee table.

The payroll table, however, does not get updated because the payroll clerk simply forgets to do so. Consequently, Mr. Clark Johnson keeps getting paid.

With good database design, referential integrity relationship would have been defined on these tables. They would have been linked and, depending on the defined constraints, the deletion of Clark Johnson from the employee table could not be performed unless it was accompanied by the deletion of Clark Johnson from the payroll table.

Besides data integrity and data consistency, referential integrity also has the benefits listed in the following table.

| Benefit | Description |
|---|---|
| Increases development productivity | It is not necessary to code SQL statements to enforce referential constraints. <br><br> Teradata Database automatically enforces referential integrity. |

| Benefit | Description |
|---------|-------------|
| Requires fewer programs to be written | All update activities are programmed to ensure that referential constraints are not violated. |
| | Teradata Database enforces referential integrity in all environments. |
| | No additional programs are required. |
| Improves performance | Teradata Database chooses the most efficient method to enforce the referential constraints. |
| | Teradata Database can optimize queries based on the fact that there is referential integrity. |

## Rules for Assigning Columns as FOREIGN KEYS

The FOREIGN KEY columns in the referencing table must be identical in definition with the keys in the referenced table. Corresponding columns must have the same data type and case sensitivity.

- For standard referential integrity, the COMPRESS option is not permitted on either the referenced or referencing column(s).
- Column level constraints are not compared.
- A one-column FOREIGN KEY cannot reference a single column in a multicolumn primary or unique key—the foreign and primary/unique key must contain the same number of columns.

## Circular References

References can be defined as circular in that Table A can reference Table B, which can reference Table A. In this case, at least one set of FOREIGN KEYS must be defined on nullable columns.

If the FOREIGN KEYS in Table A are on columns defined as nullable, then rows could be inserted into Table A with nulls for the FOREIGN KEY columns. Once the appropriate rows exist in Table B, the nulls of the FOREIGN KEY columns in Table A could then be updated to contain values that are not null that match the Table B values.

## References to the Table Itself

FOREIGN KEY references can also be to the same table that contains the FOREIGN KEY.

The referenced columns must be different columns than the FOREIGN KEY, and both the referenced and referencing columns must subscribe to the referential integrity rules.

## CREATE and ALTER TABLE Syntax

Referential integrity affects the syntax and semantics of CREATE TABLE and ALTER TABLE. For more details, see "ALTER TABLE" and "CREATE TABLE" in *SQL Data Definition Language.*

## Maintaining Foreign Keys

Definition of a FOREIGN KEY requires that Teradata Database maintain the integrity defined between the referenced and referencing table.

Teradata Database maintains the integrity of foreign keys as explained in the following table.

| For this data manipulation activity ... | The system verifies that ... |
|---|---|
| A row is inserted into a referencing table and foreign key columns are defined to be NOT NULL. | a row exists in the referenced table with the same values as those in the foreign key columns. If such a row does not exist, then an error is returned. If the foreign key contains multiple columns, and if any one column value of the foreign key is null, then none of the foreign key values are validated. |
| The values in foreign key columns are altered to be NOT NULL. | a row exists in the referenced table that contains values equal to the altered values of all of the foreign key columns. If such a row does not exist, then an error is returned. |
| A row is deleted from a referenced table. | no rows exist in referencing tables with foreign key values equal to those of the row to be deleted. If such rows exist, then an error is returned. |
| Before a referenced column in a referenced table is updated. | no rows exist in a referencing table with foreign key values equal to those of the referenced columns. If such rows exist, then an error is returned. |
| Before the structure of columns defined as foreign keys or referenced by foreign keys is altered. | the change would not violate the rules for definition of a foreign key constraint. An ALTER TABLE or DROP INDEX statement attempting to change such a columns structure returns an error. |
| A table referenced by another is dropped. | the referencing table has dropped its foreign key reference to the referenced table. |

| For this data manipulation activity … | The system verifies that … |
|---|---|
| An ALTER TABLE statement adds a foreign key reference to a table.<br><br>The same processes occur whether the reference is defined for standard or for soft referential integrity. | all of the values in the foreign key columns are validated against columns in the referenced table.<br><br>When the system parses ALTER TABLE, it defines an error table that:<br><br>• Has the same columns and primary index as the target table of the ALTER TABLE statement.<br><br>• Has a name that is the same as the target table name suffixed with the reference index number.<br><br>  A reference index number is assigned to each foreign key constraint for a table.<br><br>  To determine the number, use one of the following system views.<br><br>    • RI_Child_TablesV<br>    • RI_Distinct_ChildrenV<br>    • RI_Distinct_ParentsV<br>    • RI_Parent_TablesV<br><br>• Is created under the same user or database as the table being altered.<br><br>  If a table already exists with the same name as that generated for the error table then an error is returned to the ALTER TABLE statement.<br><br>Rows in the referencing table that contain values in the foreign key columns that cannot be found in any row of the referenced table are copied into the error table (the base data of the target table is not modified).<br><br>It is your responsibility to:<br><br>• Correct data values in the referenced or referencing tables so that full referential integrity exists between the two tables.<br><br>  Use the rows in the error table to define which corrections to make.<br><br>• Maintain the error table. |

## Referential Integrity and the ARC Utility

The Archive/Recovery (ARC) utility archives and restores individual tables. It also copies tables from one database to another.

When a table is restored or copied into a database, the Data Dictionary definition of that table is also restored. The Data Dictionary definitions of both the referenced (parent) and referencing (child) table contain the complete definition of a reference.

By restoring a single table, it is possible to create an inconsistent reference definition in Teradata Database. When either a parent or child table is restored, the reference is marked as inconsistent in the Data Dictionary definitions. The ARC utility can validate these references after the restore is done.

While a table is marked as inconsistent, no updates, inserts, or deletes are permitted. The table is fully usable only when the inconsistencies are resolved (see below). This restriction is true for both hard and soft (Referential Constraint) referential integrity constraints.

It is possible that the user either intends to or must revert to a definition of a table which results in an inconsistent reference on that table. The Archive and Restore operations are the most common cause of such inconsistencies.

To remove inconsistent references from a child table that is archived and restored, follow these steps:

1   After archiving the child table, drop the parent table.

2   Restore the child table.

    When the child table is restored, the parent table no longer exists. The normal ALTER TABLE DROP FOREIGN KEY statement does not work, because the parent table references cannot be resolved.

3   Use the DROP INCONSISTENT REFERENCES option to remove these inconsistent references from a table.

    The syntax is:

    ```
    ALTER TABLE database_name.table_name DROP INCONSISTENT REFERENCES
    ```
    You must have DROP privileges on the target table of the statement to perform this option, which removes all inconsistent internal indexes used to establish references.

For more information, see *Teradata Archive/Recovery Utility Reference.*

## Referential Integrity and the FastLoad and MultiLoad Utilities

Foreign key references are not supported for any table that is the target table for a FastLoad or MultiLoad.

For further details, see:

*   *Database Design*
*   *Teradata FastLoad Reference*
*   *Teradata MultiLoad Reference*

# Views

## Views and Tables

A view can be compared to a window through which you can see selected portions of a database. Views retrieve portions of one or more tables or other views.

Views look like tables to a user, but they are virtual, not physical, tables. They display data in columns and rows and, in general, can be used as if they were physical tables. However, only the column definitions for a view are stored: views are not physical tables.

A view does not contain data: it is a virtual table whose definition is stored in the Data Dictionary. The view is not materialized until it is referenced by a statement. Some operations that are permitted for the manipulation of tables are not valid for views, and other operations are restricted, depending on the view definition.

## Defining a View

The CREATE VIEW statement defines a view. The statement names the view and its columns, defines a SELECT on one or more columns from one or more underlying tables and/or views, and can include conditional expressions and aggregate operators to limit the row retrieval.

## Using Views

The primary reason to use views is to simplify end user access to Teradata Database. Views provide a constant vantage point from which to examine and manipulate the database. Their perspective is altered neither by adding or nor by dropping columns from its component base tables unless those columns are part of the view definition.

From an administrative perspective, views are useful for providing an easily maintained level of security and authorization. For example, users in a Human Resources department can access tables containing sensitive payroll information without being able to see salary and bonus columns. Views also provide administrators with an ability to control read and update privileges on the database with little effort.

## Restrictions

Some operations that are permitted on base tables are not permitted on views—sometimes for obvious reasons and sometimes not.

The following set of rules outlines the restrictions on how views can be created and used.

- You cannot create an index on a view.
- A view definition cannot contain an ORDER BY clause.
- Any derived columns in a view must explicitly specify view column names, for example by using an AS clause or by providing a column list immediately after the view name.

- You cannot update tables from a view under the following circumstances:
  - The view is defined as a join view (defined on more than one table)
  - The view contains derived columns.
  - The view definition contains a DISTINCT clause.
  - The view definition contains a GROUP BY clause.
  - The view defines the same column more than once.

## Archiving Views

Views are archived and restored as part of a database archive and restoration. Individual views can be archived or restored using the ARCHIVE or RESTORE statements of the ARC utility.

# Triggers

Triggers are active database objects associated with a subject table. A trigger essentially consists of a stored SQL statement or a block of SQL statements.

Triggers execute when an INSERT, UPDATE, DELETE, or MERGE modifies a specified column or columns in the subject table.

Typically, a stored trigger performs an UPDATE, INSERT, DELETE, MERGE, or other SQL operation on one or more tables, which may possibly include the subject table.

Triggers in Teradata Database conform to the ANSI/ISO SQL:2008 standard, and also provide some additional features.

Triggers have two types of granularity:

- Row triggers fire once for each row of the subject table that is changed by the triggering event and that satisfies any qualifying condition included in the row trigger definition.
- Statement triggers fire once upon the execution of the triggering statement.

You can create, alter, and drop triggers.

| IF you want to … | THEN use … |
|---|---|
| define a trigger | CREATE TRIGGER. |
| <ul><li>enable a trigger</li><li>disable a trigger</li><li>change the creation timestamp for a trigger</li></ul> | ALTER TRIGGER.<br><br>Disabling a trigger stops the trigger from functioning, but leaves the trigger definition in place as an object. This allows utility operations on a table that are not permitted on tables with enabled triggers.<br><br>Enabling a trigger restores its active state. |
| remove a trigger from the system permanently | DROP TRIGGER. |

For details on creating, dropping, and altering triggers, see *SQL Data Definition Language.*

## Process Flow

Note that this is a logical flow, not a physical re-enactment of how Teradata Database processes a trigger.

1  The triggering event occurs on the subject table.

2  A determination is made as to whether triggers defined on the subject table are to become active upon a triggering event.

3  Qualified triggers are examined to determine the trigger action time, whether they are defined to fire before or after the triggering event.

4  When multiple triggers qualify, then they fire normally in the ANSI-specified order of creation timestamp.

   To override the creation timestamp and specify a different execution order of triggers, you can use the ORDER clause, a Teradata extension.

   Even if triggers are created without the ORDER clause, you can redefine the order of execution by changing the trigger creation timestamp using the ALTER TRIGGER statement.

5  The triggered SQL statements (triggered action) execute.

   If the trigger definition uses a REFERENCING clause to specify that old, new, or both old and new data for the triggered action is to be collected under a correlation name (an alias), then that information is stored in transition tables or transition rows:

   • OLD [ROW] values under *old_transition_variable_name*, NEW [ROW] values under *new_transition_variable_name*, or both.

   • OLD TABLE set of rows under *old_transition_table_name*, NEW TABLE set of rows under *new_transition_table_name*, or both.

   • OLD_NEW_TABLE set of rows under *old_new_table_name*, with old values as *old_transition_variable_name* and new values as *new_transition_variable_name*.

6  The trigger passes control to the next trigger, if defined, in a cascaded sequence. The sequence can include recursive triggers.

   Otherwise, control passes to the next statement in the application.

7  If any of the actions involved in the triggering event or the triggered actions abort, then all of the actions are aborted.

## Restrictions

Most Teradata load utilities cannot access a table that has an active trigger.

An application that uses triggers can use ALTER TRIGGER to disable the trigger and enable the load. The application must be sure that loading a table with disabled triggers does not result in a mismatch in a user defined relationship with a table referenced in the triggered action.

Other restrictions on triggers include:

• BEFORE statement triggers are not allowed.

- BEFORE triggers cannot have data-changing statements as triggered action (triggered SQL statements).
- BEFORE triggers cannot access OLD TABLE, NEW TABLE, or OLD_NEW_TABLE.
- Triggers and hash indexes are mutually exclusive. You cannot define triggers on a table on which a hash index is already defined.
- A positioned (updatable cursor) UPDATE or DELETE is not allowed to fire a trigger. An attempt to do so generates an error.
- You cannot define triggers on an error logging table.

## Archiving Triggers

Triggers are archived and restored as part of a database archive and restoration. Individual triggers can be archived or restored using the ARCHIVE or RESTORE statements of the ARC utility.

## Related Topics

| For detailed information on … | See … |
|---|---|
| <ul><li>guidelines for creating triggers</li><li>conditions that cause triggers to fire</li><li>trigger action that occurs when a trigger fires</li><li>the trigger action time</li><li>when to use row triggers and when to use statement triggers</li></ul> | CREATE TRIGGER in *SQL Data Definition Language.* |
| <ul><li>temporarily disabling triggers</li><li>enabling triggers</li><li>changing the creation timestamp of a trigger</li></ul> | ALTER TRIGGER in *SQL Data Definition Language.* |
| permanently removing triggers from the system | DROP TRIGGER in *SQL Data Definition Language.* |

# Macros

A macro consists of one or more statements that can be executed by performing a single statement. Each time the macro is performed, one or more rows of data can be returned.

A frequently used SQL statement or series of statements can be incorporated into a macro and defined using the SQL CREATE MACRO statement. See "CREATE MACRO" in *SQL Data Definition Language.*

The statements in the macro are performed using the EXECUTE statement. See "EXECUTE (Macro Form)" in *SQL Data Manipulation Language.*

A macro can include an EXECUTE statement that executes another macro.

Performing a macro is similar to performing a multistatement request (see "Multistatement Requests" on page 130).

## Single-User and Multiuser Macros

You can create a macro for your own use, or grant execution authorization to others.

For example, your macro might enable a user in another department to perform operations on the data in Teradata Database. When executing the macro, a user need not be aware of the database being accessed, the tables affected, or even the results.

## Contents of a Macro

With the exception of CREATE AUTHORIZATION and REPLACE AUTHORIZATION, a data definition statement is allowed in macro if it is the only SQL statement in that macro.

A data definition statement is not resolved until the macro is executed, at which time unqualified database object references are fully resolved using the default database of the user submitting the EXECUTE statement. If this is not the desired result, you must fully qualify all object references in a data definition statement in the macro body.

A macro can contain parameters that are substituted with data values each time the macro is executed. It also can include a USING modifier, which allows the parameters to be filled with data from an external source such as a disk file. A COLON character prefixes references to a parameter name in the macro. Parameters cannot be used for data object names.

## Executing a Macro

Regardless of the number of statements in a macro, Teradata Database treats it as a single request.

When you execute a macro, either all its statements are processed successfully or none are processed. If a macro fails, it is aborted, any updates are backed out, and the database is returned to its original state.

## Ways to Perform SQL Macros in Embedded SQL

| IF the macro … | THEN use … |
|---|---|
| is a single statement, and that statement returns no data | • the EXEC statement to specify static execution of the macro<br>  -or-<br>• the PREPARE and EXECUTE statements to specify dynamic execution.<br>Use DESCRIBE to verify that the single statement of the macro is not a data returning statement. |
| • consists of multiple statements<br>• returns data | a cursor, either static or dynamic.<br>The type of cursor used depends on the specific macro and on the needs of the application. |

## Static SQL Macro Execution in Embedded SQL

Static SQL macro execution is associated with a macro cursor using the macro form of the DECLARE CURSOR statement.

When you perform a static macro, you must use the EXEC form to distinguish it from the dynamic SQL statement EXECUTE.

## Dynamic SQL Macro Execution in Embedded SQL

Define dynamic macro execution using the PREPARE statement with the statement string containing an EXEC *macro_name* statement rather than a single-statement request.

The dynamic request is then associated with a dynamic cursor. See "DECLARE CURSOR (Macro Form)" in *SQL Stored Procedures and Embedded SQL* for more information on the use of macros.

## Dropping, Replacing, Renaming, and Retrieving Information About a Macro

| IF you want to … | THEN use the following statement … |
|---|---|
| drop a macro | DROP MACRO |
| redefine an existing macro | REPLACE MACRO |
| rename a macro | RENAME MACRO |
| get the attributes for a macro | HELP MACRO |
| get the data definition statement most recently used to create, replace, or modify a macro | SHOW MACRO |

For more information, see *SQL Data Definition Language*.

## Archiving Macros

Macros are archived and restored as part of a database archive and restoration. Individual macros can be archived or restored using the ARCHIVE or RESTORE statements of the ARC utility.

For details, see *Teradata Archive/Recovery Utility Reference*.

# Stored Procedures

Stored procedures are called Persistent Stored Modules in the ANSI/ISO SQL:2008 standard. They are written in SQL and consist of a set of control and condition handling statements that make SQL a computationally complete programming language.

These features provide a server-based procedural interface to Teradata Database for application programmers.

Teradata stored procedure facilities are a subset of and conform to the ANSI/ISO SQL:2008 standards for semantics.

## Elements of Stored Procedures

The set of statements constituting the main tasks of the stored procedure is called the stored procedure body, which can consist of a single statement or a compound statement, or block.

A single statement stored procedure body can contain one control statement, such as LOOP or WHILE, or one SQL DDL, DML, or DCL statement, including dynamic SQL. Some statements are not allowed, including:

- Any declaration (local variable, cursor, condition, or condition handler) statement
- A cursor statement (OPEN, FETCH, or CLOSE)

A compound statement stored procedure body consists of a BEGIN-END statement enclosing a set of declarations and statements, including:

- Local variable declarations
- Cursor declarations
- Condition declarations
- Condition handler declaration statements
- Control statements
- SQL DML, DDL, and DCL statements supported by stored procedures, including dynamic SQL
- Multistatement requests (including dynamic multistatement requests) delimited by the keywords BEGIN REQUEST and END REQUEST

Compound statements can also be nested.

For information about control statements, parameters, local variables, and labels, see *SQL Stored Procedures and Embedded SQL*.

## Creating Stored Procedures

A stored procedure can be created from:

- BTEQ utility using the COMPILE command
- CLIv2 applications, ODBC, JDBC, and Teradata SQL Assistant (formerly called Queryman) using the SQL CREATE PROCEDURE or REPLACE PROCEDURE statement.

The procedures are stored in the user database space as objects and are executed on the server.

For the syntax of data definition statements related to stored procedures, including CREATE PROCEDURE and REPLACE PROCEDURE, see *SQL Data Definition Language*.

Note: The stored procedure definitions in the next examples are designed only to demonstrate the usage of the feature. They are not recommended for use.

### Example

Assume you want to define a stored procedure NewProc to add new employees to the Employee table and retrieve the name of the department to which the employee belongs.

You can also report an error, in case the row that you are trying to insert already exists, and handle that error condition.

The CREATE PROCEDURE statement looks like this:

```
CREATE PROCEDURE NewProc (IN name CHAR(12),
                          IN number INTEGER,
                          IN dept INTEGER,
                          OUT dname CHAR(10))
BEGIN
   INSERT INTO Employee (EmpName, EmpNo, DeptNo )
      VALUES (name, number, dept);
   SELECT DeptName
      INTO dname FROM Department
         WHERE DeptNo = dept;
END;
```

This stored procedure defines parameters that must be filled in each time it is called.

## Modifying Stored Procedures

To modify a stored procedure definition, use the REPLACE PROCEDURE statement.

### Example

Assume you want to change the previous example to insert salary information to the Employee table for new employees.

The REPLACE PROCEDURE statement looks like this:

```
REPLACE PROCEDURE NewProc (IN name CHAR(12),
                           IN number INTEGER,
                           IN dept INTEGER,
                           IN salary DECIMAL(10,2),
                           OUT dname CHAR(10))

BEGIN
   INSERT INTO Employee (EmpName, EmpNo, DeptNo, Salary_Amount)
      VALUES (name, number, dept, salary);
   SELECT DeptName
      INTO dname FROM Department
         WHERE DeptNo = dept;
END;
```

## Executing Stored Procedures

If you have sufficient privileges, you can execute a stored procedure from any supporting client utility or interface using the SQL CALL statement. You can also execute a stored procedure from an external stored procedure written in C, C++, or Java.

You have to specify arguments for all the parameters contained in the stored procedure.

Here is an example of a CALL statement to execute the procedure created in "Example":

```
CALL NewProc ('Jonathan', 1066, 34, dname);
```

For details on using CALL to execute stored procedures, see "CALL" in *SQL Data Manipulation Language*.

For details on executing stored procedures from an external stored procedure, see *SQL External Routine Programming*.

## Output From Stored Procedures

Stored procedures can return output values in the INOUT or OUT arguments of the CALL statement. They can also return result sets, the results of SELECT statements that are executed when the stored procedure opens result set cursors. To return result sets, the CREATE PROCEDURE or REPLACE PROCEDURE statement for the stored procedure must specify the DYNAMIC RESULT SET clause.

For details on how to write a stored procedure that returns result sets, see *SQL Stored Procedures and Embedded SQL*.

## Recompiling Stored Procedures

The ALTER PROCEDURE statement enables recompilation of stored procedures without having to execute SHOW PROCEDURE and REPLACE PROCEDURE statements.

This statement provides the following benefits:

- Stored procedures created in earlier releases of Teradata Database can be recompiled to derive the benefits of new features and performance improvements.
- Recompilation is also useful for cross-platform archive and restoration of stored procedures.
- ALTER PROCEDURE allows changes in the following compile-time attributes of a stored procedure:
  - SPL option
  - Warnings option

## Deleting, Renaming, and Retrieving Information About a Stored Procedure

| IF you want to … | THEN use the following statement … |
|---|---|
| delete a stored procedure from a database | DROP PROCEDURE |
| rename a stored procedure | RENAME PROCEDURE |
| get information about the parameters specified in a stored procedure and their attributes | HELP PROCEDURE |
| get the data definition statement most recently used to create, replace, or modify a stored procedure | SHOW PROCEDURE |

For more information, see *SQL Data Definition Language.*

## Archiving Procedures

Stored procedures are archived and restored as part of a database archive and restoration. Individual stored procedures can be archived or restored using the ARCHIVE or RESTORE statements of the ARC utility.

## Related Topics

| For details on … | See … |
|---|---|
| stored procedure control and condition handling statements | *SQL Stored Procedures and Embedded SQL.* |
| invoking stored procedures | the CALL statement in *SQL Data Manipulation Language.* |
| creating, replacing, dropping, or renaming stored procedures | *SQL Data Definition Language.* |
| controlling and tracking privileges for stored procedures | • *SQL Data Control Language.*<br>• *Database Administration.* |

# External Stored Procedures

External stored procedures are written in the C, C++, or Java programming language, installed on the database, and then executed like stored procedures.

## Usage

Here is a synopsis of the steps you take to develop, compile, install, and use external stored procedures:

1   Write, test, and debug the C, C++, or Java code for the procedure.

2   If you are using Java, place the class or classes for the external stored procedure in an archive file (JAR or ZIP) and call the SQLJ.INSTALL_JAR external stored procedure to register the archive file with the database.

3   Use CREATE PROCEDURE or REPLACE PROCEDURE for external stored procedures to create a database object for the external stored procedure.

4   Use GRANT to grant privileges to users who are authorized to use the external stored procedure.

5   Invoke the procedure using the CALL statement.

## Executing SQL From External Stored Procedures

To execute SQL, a C or C++ external stored procedure can use CLIv2 and a Java external stored procedure can use JDBC.

A C or C++ external stored procedure can also call the FNC_CallSP library function to call a stored procedure that contains SQL.

## Differences Between Stored Procedures and External Stored Procedures

Using external stored procedures is very similar to using stored procedures, except for the following:

•   Invoking an external stored procedure from a client application does not affect the nesting limit for stored procedures.

•   To install an external stored procedure on a database, you must have the CREATE EXTERNAL PROCEDURE privilege on the database.

•   The CREATE PROCEDURE statement for external stored procedures is different from the CREATE PROCEDURE statement for stored procedures. In addition to syntax differences, you do not have to use the COMPILE command in BTEQ.

## Related Topics

| For details on … | See … |
|---|---|
| external stored procedure programming | *SQL External Routine Programming.* |

| For details on … | See … |
|---|---|
| invoking external stored procedures | the CALL statement in *SQL Data Manipulation Language.* |
| installing external stored procedures on the server | the CREATE/REPLACE PROCEDURE statement in *SQL Data Definition Language.* |
| archiving and restoring external stored procedures | *Teradata Archive/Recovery Utility Reference.* |

# User-Defined Functions

SQL provides a set of useful functions, but they might not satisfy all of the particular requirements you have to process your data.

Teradata Database supports two types of user-defined functions (UDFs) that allow you to extend SQL by writing your own functions:

- SQL UDFs
- External UDFs

## SQL UDFs

SQL UDFs allow you to encapsulate regular SQL expressions in functions and then use them like standard SQL functions.

Rather than coding commonly used SQL expressions repeatedly in queries, you can objectize the SQL expressions through SQL UDFs.

Moving complex SQL expressions from queries to SQL UDFs makes the queries more readable and can reduce the client/server network traffic.

## External UDFs

External UDFs allow you to write your own functions in the C, C++, or Java programming language, install them on the database, and then use them like standard SQL functions.

You can also install external UDF objects or packages from third-party vendors.

Teradata Database supports three types of external UDFs.

| UDF Type | Description |
|---|---|
| Aggregate | Aggregate functions produce summary results. They differ from scalar functions in that they take grouped sets of relational data, make a pass over each group, and return one result for the group. Some examples of standard SQL aggregate functions are AVG, SUM, MAX, and MIN. |
| Scalar | Scalar functions take input parameters and return a single value result. Examples of standard SQL scalar functions are CHARACTER_LENGTH, POSITION, and TRIM. |

| UDF Type | Description |
|----------|-------------|
| Table | A table function is invoked in the FROM clause of a SELECT statement and returns a table to the statement. |

## Usage

To create and use an SQL UDF, follow these steps:

1   Use CREATE FUNCTION or REPLACE FUNCTION to define the UDF.

2   Use GRANT to grant privileges to users who are authorized to use the UDF.

3   Call the function.

Here is a synopsis of the steps you take to develop, compile, install, and use an external UDF:

1   Write, test, and debug the C, C++, or Java code for the UDF.

2   If you are using Java, place the class or classes for the UDF in an archive file (JAR or ZIP) and call the SQLJ.INSTALL_JAR external stored procedure to register the archive file with the database.

3   Use CREATE FUNCTION or REPLACE FUNCTION to create a database object for the UDF.

4   Use GRANT to grant privileges to users who are authorized to use the UDF.

5   Call the function.

## Related Topics

| For more information on … | See … |
|---------------------------|-------|
| writing, testing, and debugging source code for an external UDF | *SQL External Routine Programming.* |
| data definition statements related to UDFs, including CREATE FUNCTION and REPLACE FUNCTION | *SQL Data Definition Language.* |
| invoking a table function in the FROM clause of a SELECT statement | *SQL Data Manipulation Language.* |
| archiving and restoring UDFs | *Teradata Archive/Recovery Utility Reference.* |

# Profiles

Profiles define values for the following system parameters:

•   Default database

•   Spool space

•   Temporary space

- Default account and alternate accounts
- Password security attributes
- Optimizer cost profile
- Query band

An administrator can define a profile and assign it to a group of users who share the same settings. For information on Optimizer cost profiles, see *SQL Request and Transaction Processing*

## Advantages of Using Profiles

- Simplify system administration.

  Administrators can create a profile that contains system parameters and assign the profile to a group of users. To change a parameter, the administrator updates the profile instead of each individual user.

- Control password security.

  A profile can define password attributes such as the number of:

  - Days before a password expires
  - Days before a password can be used again
  - Minutes to lock out a user after a certain number of failed logon attempts

  Administrators can assign the profile to an individual user or to a group of users.

## Usage

The following steps describe how to use profiles to manage a common set of parameters for a group of users.

1   Define a user profile.

    A CREATE PROFILE statement defines a profile, and lets you set:

    - Account identifiers to charge for the space used and a default account identifier
    - Default database
    - Space to allocate for spool files and temporary tables
    - Optimizer cost profile to activate at the session and request scope level
    - Number of incorrect logon attempts to allow before locking a user and the number of minutes before unlocking a locked user
    - A query band to enforce query band name/value pairs
    - Number of days before a password expires and the number of days before a password can be used again
    - Minimum and maximum number of characters in a password string
    - The characters allowed in a password string, including whether to:
      - Allow digits and special characters
      - Require at least one numeric character
      - Require at least one alpha character

- Require at least one special character
- Restrict the password string from containing the user name
- Require a mixture of uppercase and lowercase characters
- Restrict certain words from being a significant part of a password string

2   Assign the profile to users.

Use the CREATE USER or MODIFY USER statement to assign a profile to a user. Profile settings override the values set for the user.

3   If necessary, change any of the system parameters for a profile.

Use the MODIFY PROFILE statement to change a profile.

## Related Topics

| For information on … | See … |
| --- | --- |
| the syntax and usage of profiles | *SQL Data Definition Language.* |
| passwords and security | *Security Administration.* |
| optimizer cost profiles | *SQL Request and Transaction Processing.* |

# Roles

Roles define privileges on database objects. A user who is assigned a role can access all the objects that the role has privileges to.

Roles simplify management of user privileges. A database administrator can create different roles for different job functions and responsibilities, grant specific privileges on database objects to the roles, and then grant membership to the roles to users.

## Advantages of Using Roles

- Simplify privilege administration.

  A database administrator can grant privileges on database objects to a role and have the privileges automatically applied to all users assigned to that role.

  When a user's function within an organization changes, changing the user's role is far easier than deleting old privileges and granting new privileges to go along with the new function.

- Reduce Data Dictionary disk space.

  Maintaining privileges on a role level rather than on an individual level makes the size of the DBC.AccessRights table much smaller. Instead of inserting one row per user per privilege on a database object, Teradata Database inserts one row per role per privilege in DBC.AccessRights, and one row per role member in DBC.RoleGrants.

## Usage

The following steps describe how to manage user privileges using roles.

1   Define a role.

    A CREATE ROLE statement defines a role. A newly created role does not have any associated privileges.

2   Add privileges to the role.

    Use the GRANT statement to grant privileges to roles on databases, tables, views, macros, columns, triggers, stored procedures, join indexes, hash indexes, and UDFs.

3   Grant the role to users or other roles.

    Use the GRANT statement to grant a role to users or other roles.

4   Assign default roles to users.

    Use the DEFAULT ROLE option of the CREATE USER or MODIFY USER statement to specify the default role for a user, where:

| DEFAULT ROLE = … | Specifies … |
| --- | --- |
| *role_name* | the name of one role to assign as the default role for a user. |
| NONE | that the user does not have a default role. |
| NULL | |
| ALL | the default role to be all roles that are directly or indirectly granted to the user. |

At logon time, the default role of the user becomes the current role for the session.

Privilege validation uses the active roles for a user, which include the current role and all nested roles.

5   If necessary, change the current role for a session.

    Use the SET ROLE statement to change the current role for a session.

Managing role-based privileges requires sufficient privileges. For example, the CREATE ROLE statement is only authorized to users who have the CREATE ROLE system privilege.

## Related Topics

For information on the syntax and usage of roles, see *SQL Data Definition Language.*

# User-Defined Types

SQL provides a set of predefined data types, such as INTEGER and VARCHAR, that you can use to store the data that your application uses, but they might not satisfy all of the requirements you have to model your data.

User-defined types (UDTs) allow you to extend SQL by creating your own data types and then using them like predefined data types.

## UDT Types

Teradata Database supports distinct and structured UDTs.

| UDT Type | Description | Example |
|---|---|---|
| Distinct | A UDT that is based on a single predefined data type, such as INTEGER or VARCHAR. | A distinct UDT named euro that is based on a DECIMAL(8,2) data type can store monetary data. |
| Structured | A UDT that is a collection of one or more fields called attributes, each of which is defined as a predefined data type or other UDT (which allows nesting). | A structured UDT named circle can consist of x-coordinate, y-coordinate, and radius attributes. |

Distinct and structured UDTs can define methods that operate on the UDT. For example, a distinct UDT named euro can define a method that converts the value to a US dollar amount. Similarly, a structured UDT named circle can define a method that computes the area of the circle using the radius attribute.

Teradata Database also supports a form of structured UDT called dynamic UDT. Instead of using a CREATE TYPE statement to define the UDT, like you use to define a distinct or structured type, you use the NEW VARIANT_TYPE expression to construct an instance of a dynamic UDT and define the attributes of the UDT at run time.

Unlike distinct and structured UDTs, which can appear almost anywhere that you can specify predefined types, you can only specify a dynamic UDT as the data type of (up to eight) input parameters to external UDFs. The benefit of dynamic UDTs is that they significantly increase the number of input arguments that you can pass in to external UDFs.

## Using a Distinct UDT

Here is a synopsis of the steps you take to develop and use a distinct UDT:

1   Use the CREATE TYPE statement to create a distinct UDT that is based on a predefined data type, such as INTEGER or VARCHAR.

    Teradata Database automatically generates functionality for the UDT that allows you to import and export the UDT between the client and server, use the UDT in a table, perform comparison operations between two UDTs, and perform data type conversions between the UDT and the predefined data type on which the definition is based.

2    If the UDT defines methods, write, test, and debug the C or C++ code for the methods, and then use CREATE METHOD or REPLACE METHOD to identify the location of the source code and install it on the server.

     The methods are compiled, linked to the dynamic linked library (DLL or SO) associated with the SYSUDTLIB database, and distributed to all Teradata Database nodes in the system.

3    Use GRANT to grant privileges to users who are authorized to use the UDT.

4    Use the UDT as the data type of a column in a table definition.

## Using a Structured UDT

Here is a synopsis of the steps you take to develop and use a structured UDT (that is *not* a dynamic UDT):

1    Use the CREATE TYPE statement to create a structured UDT and specify attributes, constructor methods, and instance methods.

     Teradata Database automatically generates the following functionality:

- A default constructor function that you can use to construct a new instance of the structured UDT and initialize the attributes to NULL
- Observer methods for each attribute that you can use to get the attribute values
- Mutator methods for each attribute that you can use to set the attribute values

2    Follow these steps to implement, install, and register cast functionality for the UDT (Teradata Database does not automatically generate cast functionality for structured UDTs):

a    Write, test, and debug C or C++ code that implements cast functionality that allows you to perform data type conversions between the UDT and other data types, including other UDTs.

b    Identify the location of the source code and install it on the server:

| IF you write the source code as a … | THEN use one of the following statements … |
|---|---|
| method | CREATE METHOD or REPLACE METHOD |
| function | CREATE FUNCTION or REPLACE FUNCTION |

     The source code is compiled, linked to the dynamic linked library (DLL or SO) associated with the SYSUDTLIB database, and distributed to all Teradata Database nodes in the system.

c    Use the CREATE CAST or REPLACE CAST statement to register the method or function as a cast routine for the UDT.

d    Repeat Steps a through c for all methods or functions that provide cast functionality.

3   Follow these steps to implement, install, and register ordering functionality for the UDT (Teradata Database does not automatically generate ordering functionality for structured UDTs):

a   Write, test, and debug C or C++ code that implements ordering functionality that allows you to perform comparison operations between two UDTs.

b   Identify the location of the source code and install it on the server:

| IF you write the source code as a ... | THEN use one of the following statements ... |
|---|---|
| method | CREATE METHOD or REPLACE METHOD |
| function | CREATE FUNCTION or REPLACE FUNCTION |

The source code is compiled, linked to the dynamic linked library (DLL or SO) associated with the SYSUDTLIB database, and distributed to all Teradata Database nodes in the system.

c   Use the CREATE ORDERING or REPLACE ORDERING statement to register the method or function as an ordering routine for the UDT.

4   Follow these steps to implement, install, and register transform functionality for the UDT (Teradata Database does not automatically generate transform functionality for structured UDTs):

a   Write, test, and debug C or C++ code that implements transform functionality that allows you to import and export the UDT between the client and server.

b   Identify the location of the source code and install it on the server:

| IF the source code implements transform functionality for ... | THEN ... | |
|---|---|---|
| importing the UDT to the server | you must write the source code as a C or C++ UDF and use CREATE FUNCTION or REPLACE FUNCTION to identify the location of the source code and install it on the server. | |
| exporting the UDT from the server | **IF you write the source code as a ...** | **THEN use one of the following statements to identify the location of the source code and install it on the server ...** |
| | method | CREATE METHOD or REPLACE METHOD |
| | function | CREATE FUNCTION or REPLACE FUNCTION |

The source code is compiled, linked to the dynamic linked library (DLL or SO) associated with the SYSUDTLIB database, and distributed to all Teradata Database nodes in the system.

c   Repeat Steps a through b.

| IF you took Steps a through b to implement and install this transform functionality … | THEN repeat Steps a through b to implement and install this transform functionality … |
| --- | --- |
| importing the UDT to the server | exporting the UDT from the server |
| exporting the UDT from the server | importing the UDT to the server |

d   Use the CREATE TRANSFORM or REPLACE TRANSFORM statement to register the transform routines for the UDT.

5   If the UDT defines constructor methods or instance methods, write, test, and debug the C or C++ code for the methods, and then use CREATE METHOD or REPLACE METHOD to identify the location of the source code and install it on the server.

The methods are compiled, linked to the dynamic linked library (DLL or SO) associated with the SYSUDTLIB database, and distributed to all Teradata Database nodes in the system.

6   Use GRANT to grant privileges to users who are authorized to use the UDT.

7   Use the UDT as the data type of a column in a table definition.

## Using a Dynamic UDT

Follow these steps to use a dynamic UDT as the data type of an input parameter to an external UDF:

1   In the CREATE FUNCTION or REPLACE FUNCTION statement for the UDF, specify the data type of up to eight input parameters as VARIANT_TYPE.

2   Write, test, and debug the C or C++ source code for the UDF.

3   Call the UDF, using the NEW VARIANT_TYPE expression to construct instances of dynamic UDT arguments and define up to 128 attributes for each UDT.

## UDT Indexing

A user can declare a primary or secondary index on a UDT column when issuing a CREATE TABLE, CREATE INDEX, CREATE JOIN INDEX or CREATE HASH INDEX statement.

Once the index has been created, the user can issue DML statements containing UDT predicate and/or UDT join term expressions.

## Related Topics

| For more information on … | See … |
|---|---|
| • CREATE TYPE<br>• CREATE METHOD and REPLACE METHOD<br>• CREATE FUNCTION and REPLACE FUNCTION<br>• CREATE CAST and REPLACE CAST<br>• CREATE ORDERING and REPLACE ORDERING<br>• CREATE TRANSFORM and REPLACE TRANSFORM | *SQL Data Definition Language.* |
| UDT expressions, including NEW and NEW VARIANT_TYPE | *SQL Functions, Operators, Expressions, and Predicates.* |
| writing, testing, and debugging source code for a constructor method or instance method | *SQL External Routine Programming.* |
| writing, testing, and debugging source code for a UDF that uses UDT types | |
| archiving and restoring UDTs and methods | *Teradata Archive/Recovery Utility Reference.* |

This chapter explains basic SQL syntax and the lexicon for Teradata SQL, a single, unified, nonprocedural language that provides the capabilities for querying the Teradata Database.

# SQL Statement Structure

## Syntax

The following diagram indicates the basic structure of an SQL statement.



FF07D232

where:

| This syntax element … | Specifies … |
|---|---|
| *statement_keyword* | the name of the statement. |
| *expressions* | literals, name references, or operations using names and literals. |
| *functions* | the name of a function and its arguments, if any. |
| *keywords* | special values introducing clauses or phrases or representing special objects, such as NULL.<br><br>Most keywords are reserved words and cannot be used in names. |
| *clauses* | subordinate statement qualifiers. |
| *phrases* | data attribute phrases. |
| ; | the Teradata SQL statement separator and request terminator.<br><br>The semicolon separates statements in a multi-statement request and terminates a request when it is the last nonblank character on an input line in BTEQ.<br><br>The request terminator is required for a request defined in the body of a macro.<br><br>For the distinction between *statement* and *request*, see "SQL Statements and SQL Requests" on page 189. |

## Typical SQL Statement

A typical SQL statement consists of a statement keyword, one or more column names, a database name, a table name, and one or more optional clauses introduced by keywords. For example, in the following single-statement request, the statement keyword is SELECT:

```
SELECT deptno, name, salary
FROM personnel.employee
WHERE deptno IN(100, 500)
ORDER BY deptno, name ;
```

The select list for this statement is made up of object names:

- Deptno, name, and salary (the column names)
- Personnel (the database name)
- Employee (the table name)

The search condition, or WHERE clause, is introduced by the keyword WHERE.

```
WHERE deptno IN(100, 500)
```

The sort order, or ORDER BY, clause is introduced by the keywords ORDER BY.

```
ORDER BY deptno, name
```

## Related Topics

The pages that follow provide details on the elements that appear in an SQL statement.

| For more information on … | See … |
| --- | --- |
| statement_keyword | "Keywords" on page 76. |
| keywords | |
| object names | "Object Names" on page 78. |
| expressions | "Expressions" on page 93. |
| functions | "Functions" on page 97. |
| separators | "Separators" on page 99. |
| terminators | "Terminators" on page 103. |

# Keywords

Keywords are words that have special meanings in SQL statements.

## Statement Keyword

The statement keyword, the first keyword in an SQL statement, is usually a verb. For example, in the INSERT statement, the first keyword is INSERT.

## Other Keywords

Other keywords appear throughout a statement as qualifiers (for example, DISTINCT, PERMANENT), or as words that introduce clauses (for example, IN, AS, AND, TO, WHERE).

In this book, keywords appear entirely in uppercase letters, though SQL does not discriminate between uppercase and lowercase letters in a keyword.

For example, SQL interprets the following SELECT statements identically:

```
Select Salary from Employee where EmpNo = 10005;
SELECT Salary FROM Employee WHERE EmpNo = 10005;
select Salary FRom Employee WherE EmpNo = 10005;
```

All keywords must be from the ASCII repertoire. Full-width letters are not valid regardless of the character set being used. For a list of Teradata SQL keywords, see Appendix B: "Restricted Words."

# Character Sets

## Character Data

When making a request to the database, an SQL statement can be represented using any client character set that is available on the client and enabled in the database, including user-defined character sets.

The system converts the SQL into UNICODE, and then converts any character literals from UNICODE to the data type of any column into which they are inserted.

If you enter data by way of a USING clause, the system stores the data based on the session character set, either LATIN, UNICODE, or KANJI1. As with any character data, if inserted into a character column, the USING data is implicitly translated to the character set of the receiving column.

If you create a character column, for example in a CREATE TABLE statement, and no character set is explicitly designated for the column, the column is created using the default character set.

## Setting the Default Server Character Set

You can set the default server character set for:

*   A user in a CREATE or MODIFY USER
*   Profile members in a CREATE or MODIFY PROFILE statement

Note:  Upon upgrade to Teradata Database 14.10 and above from a pre-14.10 release, the server default character set is initialized based on the language support mode; LATIN for standard mode or UNICODE for Japanese mode.

## Related Topics

| For more information on … | See … |
|---|---|
| available character sets and enabling character set options | *International Character Set Support* |
| notation used for characters is described in | "Character Shorthand Notation In This Book." |

# Object Names

Database objects (for example, databases, tables, and columns) that you specify in an SQL statement must conform to system object naming rules or the SQL statement is not valid. For a list of what the system considers to be an object name for the purpose of name validation, see Appendix E: "System Validated Object Names."

## General Rules for Constructing Object Names

Object names must be unique within the scope of the object name.

Within the database system:

- No two Database names can have the same name.
- Creating a user always creates a database with the same name as the user, so a user can not be named the same as an existing database.
- No two profiles can share a name.
- No two roles can share a name.
- A Profile can have the same name as a Role, but no Role or Profile can have the same name as a Database

Within a named Database:

- Tables, views, stored procedures, join or hash indexes, triggers, user-defined functions, and macros must have unique names

Within a Table or View:

- No two columns can have the same name.
- No two named constraints can have the same name.
- No two secondary indexes can have the same name.

Within a macro or stored procedure:

- No two parameters can have the same name.

Names are optional for CHECK constraints, REFERENCE constraints, and INDEX objects.

Object names are also subject to the name validation rules enabled on the system.

## Using QUOTATION MARKS Characters with Object Names

Enclosing an object name in QUOTATION MARKS characters (U+0022) enables the use of characters, spaces, symbols, and other special characters that may not otherwise be allowed.

For example, an object name containing a white space character must be enclosed in quotation marks:

```
"object name"
```

When used as part of an object name, rather than when used to delineate an object name, quotation marks must be represented as a sequence of two QUOTATION MARKS characters (U+0022). Each pair of quotation marks is counted as one character when calculating the name size limit.

QUOTATION MARKS characters that delineate object names are not stored in Dictionary tables, and when querying a Dictionary view, results that contain such names are displayed without the double quotation marks. However, if you include the name in a subsequent database request, you must add the double quotation marks, or the request fails.

For details, see "Characters Allowed in Object Names" on page 81.

## Case Sensitivity of Object Names

Object names are not case-dependent. Any mix of uppercase and lowercase can be used when defining or referencing object names in a request. Because of this, you cannot reuse a name that is required to be unique just by changing its case.

For example, the following statements are identical:

```
SELECT Salary FROM Employee WHERE EmpNo = 10005;
SELECT SALARY FROM EMPLOYEE WHERE EMPNO = 10005;
SELECT salary FROM employee WHERE eMpNo = 10005;
```

Note:  The case in which a column name is defined can be important. The column name is the default title of an output column, so the system returns these names in the same case in which they were defined.

For example, assume that the columns in the SalesReps table are defined:

```
CREATE TABLE SalesReps
 ( last_name VARCHAR(20) NOT NULL,
    first_name VARCHAR(12) NOT NULL, ...
```

In response to a query that does not define a TITLE phrase, for example:

```
SELECT Last_Name, First_Name
FROM SalesReps
ORDER BY Last_Name;
```

the column names are returned exactly as defined, that is, *last_name*, then *first_name*.

Note:  You can use the TITLE phrase to specify the case, wording, and placement of an output column heading either in the column definition or in an SQL request.

For more information, see *SQL Data Manipulation Language*.

### Related Information

| For information on... | See... |
|---|---|
| Using the TITLE phrase | • *SQL Data Definition Language Syntax and Examples*<br>• *SQL Data Manipulation Language* |
| • UPPER Function<br>• TRANSLATE and TRANSLATE_CHK functions | *SQL Functions, Operators, Expressions, and Predicates* |

## Restricted Words in Object Names

Be careful not to use restricted words in object names. Restricted words include both reserved and nonreserved words.

•   If you attempt to create a new object with an unquoted name that includes a reserved word, the request fails with an error.

•   The system does not reject object names that use nonreserved words, but in some contexts, the system may interpret such words using the nonreserved word semantics rather than that of an object name.

For a list of reserved and nonreserved restricted words, see Appendix B.

New releases of Teradata Database often add new reserved and nonreserved words. Names that conflict with these new reserved words should be identified and renamed as part of preparation for a Teradata Database software upgrade. Instead you can enclose names containing reserved or nonreserved words in quotation marks, but changing the words is preferred to avoid possible developer errors.

## Object Name Processing and Storage

Object names are stored in the Data Dictionary tables using the UNICODE server character set, and are processed internally as UNICODE strings. For backwards compatibility, object names are available in HELP output and in compatibility views translated to LATIN or KANJI1 based on the Language Support Mode (Standard or Japanese).

## Comparison and Normalization of Object Names

In comparing two names, the following rules apply:

•   Names are case insensitive.

•   Object names are considered identical if they are converted to NFC or NFD and compared as case insensitive. This means that a fullwidth LATIN SMALL LETTER A (U+FF41) is not the same as a LATIN SMALL LETTER A (U+0061), but case pairs are equivalent.

•   The system converts all database object names to UNICODE for storage in the Data Dictionary. Names are normalized so that they convert to UNICODE Normalization Form C (NFC) after conversion to upper case.

**Note**: Teradata Database supports all UNICODE normalization forms; NFC, NFD, NFCK, and NFDK. For information on use of the TRANSLATE function to specify alternate UNICODE normalization forms, for example, UNICODE_TO_UNICODE_NFD, see *SQL Functions, Operators, Expressions, and Predicates*.

## UNICODE and Object Names

All object names are stored in the Data Dictionary in UNICODE. Teradata provides a file that lists the UNICODE representation of all characters that are valid in an object name. The file is available on the Teradata User Documentation CD and at http://www.info.teradata.com.

| File Name | Publication Product ID | Language Mode |
|-----------|------------------------|---------------|
| UOBJNEXT.txt | B035-1200-112K | Any language mode |

**Note**: Some characters in this list may not be usable if the DBSControl NameValidationRule field is configured to enforce a reduced character set.

## Object Name Validation Options

The DBS Control field NameValidationRule is used to define object name restrictions.

Values of 2 and above place additional limits on the character repertoire allowed.

**Note**: The NameValidationRule is enforced by the system at object creation. Characters outside the repertoire defined by the rule are rejected by the system if they appear when creating object names.

## Characters Allowed in Object Names

The following table summarizes the use of characters in object names.

| Parameter | Description |
|-----------|-------------|
| Object name length | A maximum of 128 characters when expressed in UNICODE normalization form D. |
| | NameValidationRule can be used to apply additional character restrictions. |

| Parameter | Description |
|---|---|
| Characters allowed in unquoted object names<br><br>**Note**: Additional characters may be allowed in quoted or Unicode delimited names. | An object name not enclosed in quotation marks must be composed of an identifier-start character followed by a sequence of identifier-start or identifier extend characters, up to the maximum object name length limit.<br><br>**Note**: Characters in object names not enclosed in quotation marks must also be in the session character set.<br><br>Identifier start characters must be contained in the session character set and belong to one of the following Unicode General Category classes:<br><br>• Upper-case letters [Lu]<br>• Lower-case letters [Ll]<br>• Title-case letters [Lt]<br>• Modifier letters [Lm]<br>• Other letters ([Lo]<br>• Letter numbers [Nl]<br><br>...Or be one of the following characters:<br><br>• NUMBER SIGN (U+0023)<br>• DOLLAR SIGN (U+0024)<br>• LOW LINE (U+005F)<br>• INVERTED EXCLAIMATION MARK (U+001A)<br>• OVERLINE (U+203E)<br>• EURO SIGN (U+20AC)<br>• KATAKANA-HIRAGANA VOICED SOUND MARK (U+309B)<br>• KATAKANA-HIRAGANA SEMI-VOICED SOUND MARK (U+309B)<br>• FULLWIDTH NUMBER SIGN (U+FF03)<br>• FULLWIDTH DOLLAR SIGN (U+FF04)<br>• FULLWIDTH LOW LINE (U+FF3F)<br><br>Identifier-extender characters must be in the session character set and belong to one of the following Unicode General Category classes:<br><br>• Non-spacing marks [Mn]<br>• Spacing combing marks [Mc]<br>• Decimal numbers [Nd]<br>• Connector punctuations [Pc]<br>• Formatting codes [Cf]<br><br>**Note**: The MIDDLE DOT character (U+00B7) is also a valid identifier-extender character. |

| Parameter | Description |
|---|---|
| Characters allowed only in object names that are enclosed in quotation marks | A string literal is required for object names that: <br><br>• Have an identifier-extender character as the first character. <br>• Include the white space character, SPACE (U+0020) <br>• Are Teradata keywords <br><br>In addition, object names that contain any character from the following classes must be enclosed in quotation marks, unless the character explicitly appears in the list of allowed characters: <br><br>• Other, Control [Cc] <br>• Other, Not Assigned [Cn] <br><br>Note: No characters in this category appear in UNICODE character repertoire. <br><br>• Other, Private Use [Co] <br>• Other, Surrogate [Cs] <br>• Letter, Cased [LC] <br>• Mark, Enclosing [Me] <br>• Number, Other [No] <br>• Punctuation, Dash [Pd] <br>• Punctuation, Close [Pe] <br>• Punctuation, Final quotation marks [Pf] (may behave like Ps or Pe depending on usage) <br>• Punctuation, Initial quotation marks [Pi] (may behave like Ps or Pe depending on usage) <br>• Punctuation, Other [Po] <br>• Punctuation, Open [Ps] <br>• Symbol, Currency [Sc] <br>• Symbol, Modifier [Sk] <br>• Symbol, Math [Sm] <br>• Symbol, Other [So] <br>• Separator, Line [Zl] <br>• Separator, Paragraph [Zp] <br>• Separator, Space [Zs] <br><br>Note: When used as part of an object name, quotation marks must be represented as a sequence of two QUOTATION MARKS characters (U+0022). Each set of two quotation marks is counted as one character when calculating the name size limit. |
| Disallowed characters | The following characters cannot appear in an object name: <br><br>• NULL (U+0000) <br>• SUBSTITUTE character (U+001A) <br>• REPLACEMENT CHARACTER (U+FFFD) <br>• Compatibility ideographs (U+FA6C, U+FA6F, U+FAD0, FAD1, FAD5, FAD6, and FAD7) <br><br>Note: The setting of the NameValidationRule field may define additional character restrictions. |

| Parameter | Description |
|---|---|
| Other considerations | These additional restrictions apply:<br>• An object name consisting entirely of white spaces is not allowed.<br>• A trailing white space is not considered part of an object name<br>• You can use the NameValidationRule field to restrict object name allowable characters to a subset of those normally allowed. |

# Working with Unicode Delimited Identifiers

The system allows the use of UNICODE delimited identifiers to enable specification of object names and literals having characters that are not compatible with the session character set.

## System Use of Unicode Delimited Identifiers

Teradata Database uses Unicode delimited identifiers for:

• Returning certain information as the result of running a console utility or executing a UDT, UDF, or stored procedure that contains an SQL TEXT element.

• Returning results of an EXPLAIN, HELP or SHOW request for names that include characters not in the repertoire of the session character set, or that are otherwise unprintable.

For example, the table name:  テーブル

In an ASCII session, where the table name does not translate into the session character set, the table portion of the HELP output appears similar to the following:

```
Table Name: ^Z^Z^Z^Z

Table Dictionary Name: ^Z^Z^Z^Z

Table SQL Name: U&"\FF83\FF70\FF8C\FF9E\FF99"

Table UEscape: \
```

where:

| Output | Description |
|---|---|
| Table Dictionary Name: ^Z^Z ^Z^Z | Each ^Z represents an ASCII replacement character, 0x1A, used to replace the 4 untranslatable characters, テーブル.<br>**Note:** The system uses ^ to represent unprintable control characters. |

| Output | Description |
|---|---|
| Table SQL Name: U&"\FF83\FF70\FF8C\FF9E\FF99" | The SQL Name begins with U& to indicate that it is a UNICODE delimited identifier, that is, it contains untranslatable characters. |
| | The sequence \FF83\FF70\FF8C\FF9E\FF99 is the set of UNICODE identifiers for the 4 untranslatable characters, preceded by the default delimiter character, in this case, \. |
| | The string is enclosed in quotation marks so that the delimiter characters it contains can be used in an SQL request. |
| | Note: The system does not return the UEscape clause, which normally closes a UNICODE delimited identifier. If you want to use a UNICODE delimited identifier in an SQL request, you need to add the UEscape clause. See Using an SQL Name or SQL Title Value in an SQL Request. |
| Table UEscape: \ | Indicates the delimiter character used to separate the UNICODE identifiers in the SQL Name. For list of allowable delimiter characters, see "Determining the Delimiter Character" on page 85. |

## Using an SQL Name or SQL Title Value in an SQL Request

You can use the SQL Name or SQL Title of an object returned as the result of a HELP request in an SQL request.

For example, you can use HELP DATABASE to look in the mydb database for the name of a table you want to drop:

```
HELP DATABASE mydb;
… Table SQL Name U&"table_\4E00_name"
   Table UEscape \
```

The SQL Name returned by the HELP DATABASE statement is a UNICODE delimited identifier, which includes an untranslatable character that the system expresses as \4E00.

If you want to use a UNICODE delimited identifier from the SQL Name as part of an SQL request, you must:

1  Add the closing UEscape clause to the table name taken from the SQL Name field.

2  Specify the delimiter character ( \ )

For example:

```
Drop Table U&"table_\4E00_name" UEscape '\';
```

If the SQL Name is not a UNICODE delimited identifier, you can use the name in an SQL request as it appears in the HELP output, without specifying the UEscape phrase and delimiter.

## Determining the Delimiter Character

You can use one of the following delimiter characters in SQL Name and SQL title, depending on availability in the session character set:

•   BACKSLASH (U+005C)

- TILDE (U+007E)
- The YEN SIGN (U+00A5) or WON SIGN (U+20A9), depending on which is present in the session character set at 0x5C.
- NUMBER SIGN (U+0023), which is present in all supported session character sets.

The UEscape field that follows each SQL Name field must be used to identify the delimiter used.

# Hexadecimal Representation of Object Names

All object names are stored in the Data Dictionary in UNICODE. However, in an international environment, some client users may find that their client (session) character set cannot express an object name created on another client. In these cases, clients may find it useful to use a hexadecimal representation of the unavailable UNICODE characters in the object name to more easily access the object.

## Using Delimiters with Hexadecimal Identifiers

When you reference a database object using a hexadecimal representation of the object name, you must express it as a name literal or enclose the name in UNICODE delimiters.

The best practice is to use UNICODE delimited identifiers because the hexadecimal value remains the same across all supported character sets, whereas using hexadecimal name literals allows the name representation to vary depending on the character set.

### Example 1

Consider a table name of テーブル where the translation to the UNICODE server character set is equivalent to the following Unicode delimited identifier:

```
U&"#FF83#FF70#FF8C#FF9E#FF99" UESCAPE '#'
```

From a client where the client character set is KANJIEUC_0U, you can access the table using the following hexadecimal name literal:

```
'80C380B080CC80DE80D9'XN
```

From a client where the client character set is KANJISJIS_0S, you can access the table using the following hexadecimal name literal:

```
'C3B0CCDED9'XN
```

### Example 2

Assume that a hexadecimal name literal that represents the name **TAB1** using full width Latin characters from the KANJIEBCDIC5035_0I character set on a system enabled for Japanese language support:

```
SELECT EmployeeID FROM "0E42E342C142C242F10F"XN;
```

Teradata Database converts the hexadecimal name literal from a KANJI1 string to a UNICODE string to find the object name in the Data Dictionary.

Here is an example of a Unicode delimited identifier that represents the name **TAB1** on a system enabled for Japanese language support:

```
SELECT EmployeeID FROM U&"#FF34#FF21#FF22#FF11" UESCAPE '#';
```

# Hexadecimal Name Literals

Hexadecimal name literals provide a means to create and reference object names by their internal UNICODE representation in the Data Dictionary, for example, because characters are not representable in the session character set.

Note: Support for hexadecimal name literals may be discontinued in a future Teradata Database release. Use Unicode delimited identifiers when you need characters not representable in the session character set.

### ANSI Compliance

Hexadecimal name literals are Teradata extensions to the ANSI/ISO SQL:2008 standard.

### Maximum Length

A hexadecimal name literal can consist of a maximum of 60 hexadecimal digits.

### Usage

A hexadecimal name literal is useful for specifying an object name containing characters that cannot generally be entered directly from a keyboard.

Object names are stored and processed using the UNICODE server character set. Teradata Database converts a hexadecimal name literal to a UNICODE string to find the object name in the Data Dictionary.

With object naming, the following text files list the characters from the UNICODE server character set that are valid in object names and can appear in hexadecimal name literals. These files are available on the Teradata User Documentation CD and on the Web at http://www.info.teradata.com.

| File Name (on CD) | Title (on the Web) |
|---|---|
| UOBJNSTD.txt | UNICODE in Object Names on Standard Language Support Systems |
| UOBJNJAP.txt | UNICODE in Object Names on Japanese Language Support Systems |

### Restrictions

The minimal restrictions for object names are:

- The object name must not consist entirely of white space characters.
- The following characters are not allowed in the name:
  - NULL (U+0000)
  - SUBSTITUTE character (U+001A)
  - REPLACEMENT CHARACTER (U+FFFD)

- The compatibility ideographs U+FA6C, U+FACF, U+FAD0, U+FAD1, U+FAD5, U+FAD6, U+FAD7

### Example

Consider a table with the name <u>TAB1</u> (in full-width Latin characters) on a system enabled with Japanese language support.

Use this query when the session character set is KANJIEBCDIC5035_0I to return all columns from the table:

```
SELECT * FROM '0E42E342C142C242F10F'XN;
```

Use this query when the session character set is KANJIEUC_0U to return all columns:

```
SELECT * FROM '8273826082618250'XN;
```

## Related Topics

| FOR details on … | See … |
|---|---|
| object names and name validation | "General Rules for Constructing Object Names" |
| client character sets and object name restrictions | *International Character Set Support* |

### Finding the Internal Hexadecimal Representation for an Object Name

You can use the CHAR2HEXINT function to find the internal hexadecimal representation for any object name. For example, the table below shows how to find a database name.

| IF you want to find the internal representation of … | THEN use the CHAR2HEXINT function on the … |
|---|---|
| a database name that you can use to form a hexadecimal name literal | DatabaseName column of the DBC.Databases view. |
| a database name that you can use to form a Unicode delimited identifier | DatabaseName column of the DBC.DatabasesV view. |
| a table, macro, or view name that you can use to form a hexadecimal name literal | TableName column of the DBC.Tables view. |
| a table, macro, or view name that you can use to form a Unicode delimited identifier | TableName column of the DBC.TablesV view. |

For details on CHAR2HEXINT, see *SQL Functions, Operators, Expressions, and Predicates*.

### Example 1: Finding Hexadecimal Name Literals in DBC.Tables

To find the internal hexadecimal representation for the table Dbase that in a form that converts to a hexadecimal name literal, select CHAR2HEXINT from DBC.Tables:

```
SELECT CHAR2HEXINT(T.TableName) (FORMAT 'X(60)',
```

```
      TITLE 'Internal Hex Representation'),T.TableName (TITLE 'Name')
FROM DBC.Tables T
WHERE T.TableKind = 'T' AND T.TableName = 'Dbase';
```

The system returns the result, for example:

```
Internal Hex Representation                                        Name
------------------------------------------------------------------ --------------
4462617365202020202020202020202020202020202020202020202020202020 Dbase
```

You can use the result to reconstruct the object in the form of a hexadecimal name literal.

1   Remove the excess space characters (202020...) from the result.

2   Enclose the remaining numbers (those to the left of the space characters) in double quotation marks.

3   Add XN at the end.

Note:  This method, used in previous Teradata Database releases, no longer works for all names, and is deprecated future use. Instead, use the UESCAPE method shown in "Example 2: Finding Unicode Delimited Identifiers in DBC.TablesV" on page 89.

The hexadecimal name literal for the preceding example is:

```
"4462617365"XN
```

To obtain the internal hexadecimal representation of the names of other objects types, modify the WHERE clause. For example, to obtain the internal hexadecimal representation of a view, modify the WHERE clause to TableKind = 'V'.

Similarly, to obtain the internal hexadecimal representation of a macro, modify the WHERE condition to TableKind = 'M'.

For more information, see the TableKind column in *Data Dictionary*.

### Example 2: Finding Unicode Delimited Identifiers in DBC.TablesV

You can also find the internal hexadecimal representation for any database object in DBC.TablesV view. These hexadecimal representations convert to Unicode delimited identifiers. For example, for the table Dbase:

```
SELECT CHAR2HEXINT(T.TableName) (FORMAT 'X(60)',
    TITLE 'Internal Hex Representation'),T.TableName (TITLE 'Name')
FROM DBC.TablesV T
WHERE T.TableKind = 'T' AND T.TableName = 'Dbase';
```

The system returns the result, for example:

```
Internal Hex Representation                                        Name
------------------------------------------------------------------ --------------
00440062006100730065                                              Dbase
```

You can use the result to reconstruct the object name as a Unicode delimited identifier:

1   Copy the hexadecimal representation from the result of the SELECT CHAR2HEXINT request, for example:

```
00440062006100730065
```

2   Add a delimiter character before each set of double zeros (00), for example:

```
x0044x0062x0061x0073x0065
```

**Note:** You can use most printable (7-bit) ASCII characters as delimiters.

3 Convert the string to UNICODE delimiter format, as shown in "Working with Unicode Delimited Identifiers" on page 84, for example:

```
U&"x0044x0062x0061x0073x0065" UESCAPE 'x'
```

For more information on the DBC.TablesV view, see *Data Dictionary*.

# Referencing Object Names in a Request

When you reference an object name in a request, you may need to qualify the name. The topics that follow explain the rules for use of qualified and unqualified object names.

The following example shows unqualified, qualified and a fully qualified name references in a SELECT statement that accesses the Employee table in the personnel database:

```
SELECT Name, DeptNo, JobTitle
FROM Personnel.Employee
WHERE Personnel.Employee.DeptNo = 100 ;
```

## Fully Qualified Object Names

A fully qualified object name includes the names of all parent objects up to the level of the containing database. A common use case is a fully qualified column name, which consists of a database name, table name, and column name.
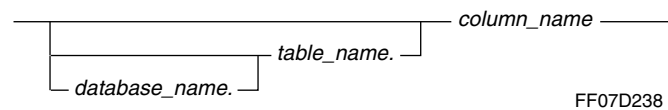


FF07D238

where:

| This Syntax element ... | Specifies ... |
|---|---|
| *database_name* | a qualifying name for the database in which the table and column being referenced is stored. <br><br> Depending on the ambiguity of the reference, *database_name* might or might not be required. <br><br> See "Unqualified Object Names" on page 91. |
| *table_name* | a qualifying name for the table in which the column being referenced is stored. <br><br> Depending on the ambiguity of the reference, *table_name* might or might not be required. <br><br> See "Unqualified Object Names" on page 91. |

| This Syntax element … | Specifies … |
|---|---|
| *column_name* | one of the following:<br>• The name of the column being referenced<br>• The alias of the column being referenced<br>• The keyword PARTITION<br>See "Using a Column Alias" on page 92. |

### Name Resolution Rules and the Need to Fully Qualify a Name

- Name resolution is performed statement by statement.
- An ambiguous unqualified name returns an error to the requestor.
- When an INSERT statement contains a subquery, names are resolved in the subquery first.
- Names in a view are resolved when the view is created.
- Names in a macro data manipulation statement are resolved when the macro is created.
- Names in a macro data definition statement are resolved when the macro is performed using the default database of the user submitting the EXECUTE statement.

  Therefore, you should fully qualify all names in a macro data definition statement, unless you specifically intend for the user's default to be used.

- Names in stored procedure statements are resolved either when the procedure is created or when the procedure is executed, depending on whether the CREATE PROCEDURE statement includes the SQL SECURITY clause and which option the clause specifies.

  Whether unqualified object names acquire the database name of the creator, invoker, or owner of the stored procedure also depends on whether the CREATE PROCEDURE statement includes the SQL SECURITY clause and which option the clause specifies.

## Unqualified Object Names

An unqualified object name is a reference to an object such as a table, column, trigger, macro, or stored procedure that does not include parent objects. For example, the WHERE clause in the following statement uses "DeptNo" as an unqualified column name:

```
SELECT *
FROM Personnel.Employee
WHERE DeptNo = 100 ;
```

### Unqualified Column Names

You can omit database and table name qualifiers when you reference columns as long as the reference is not ambiguous.

For example, the WHERE clause in the following statement:

```
SELECT Name, DeptNo, JobTitle
FROM Personnel.Employee
WHERE Personnel.Employee.DeptNo = 100 ;
```

can be written as:

```
WHERE DeptNo = 100 ;
```

because the database name and table name can be derived from the Personnel.Employee reference in the FROM clause.

## Omitting Database Names

When you omit the database name qualifier, Teradata Database looks in the following databases to find the unqualified object name:

*   The default database (see "The Default Database" on page 104)
*   Other databases, if any, referenced by the SQL statement
*   The login user database for a volatile table, if the unqualified object name is a table name
*   The SYSLIB database, if the unqualified object name is a C or C++ UDF that is not in the default database

The search must find the name in only one of those databases. An ambiguous name error message results if the name exists in more than one of those databases.

For example, if your login user database has no volatile tables named Employee and you have established Personnel as your default database, you can omit the Personnel database name qualifier from the preceding sample query.

## Related Topics

| For more information on, for example… | See … |
| --- | --- |
| default databases | "The Default Database" on page 104. |
| the DATABASE statement | *SQL Data Definition Language.* |
| the CREATE USER statement | |
| the MODIFY USER statement | |
| the CREATE PROCEDURE statement and the SQL SECURITY clause | |

# Using a Column Alias

In addition to referring to a column by name, an SQL query can reference a column by an alias. Column aliases are used for join indexes when two columns have the same name. However, an alias can be used for any column when a pseudonym is more descriptive or easier to use. Using an alias to name an expression allows a query to reference the expression.

You can specify a column alias with or without the keyword AS on the first reference to the column in the query. The following example creates and uses aliases for the first two columns.

```
SELECT departnumber AS d, employeename e, salary
FROM personnel.employee
WHERE d IN(100, 500)
ORDER BY d, e ;
```

Alias names must meet the same requirements as names of other database objects.

The scope of alias names is confined to the query.

### Referencing All Columns in a Table

An asterisk references all columns in a row simultaneously, for example, the following SELECT statement references all columns in the Employee table. A list of those fully qualified column names follows the query.

```
SELECT * FROM Employee;

Personnel.Employee.EmpNo
Personnel.Employee.Name
Personnel.Employee.DeptNo
Personnel.Employee.JobTitle
Personnel.Employee.Salary
Personnel.Employee.YrsExp
Personnel.Employee.DOB
Personnel.Employee.Sex
Personnel.Employee.Race
Personnel.Employee.MStat
Personnel.Employee.EdLev
Personnel.Employee.HCap
```

# Expressions

An expression, which specifies a value, can consist of literals (or constants), name references, or operations using names and literals.

## Scalar Expressions

A scalar expression produces a single number, character string, byte string, date, time, timestamp, or interval.

A value expression has exactly one declared type common to every possible result of evaluation. Implicit type conversion rules apply to expressions.

## Query Expressions

Query expressions operate on table values and produce rows and tables of data. Query expressions can include a FROM clause, which operates on a table reference and returns a single-table value.

Zero-table SELECT statements do not require a FROM clause.

## Literals

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

## Numeric Literals

A numeric literal (also referred to as a constant) is a character string of 1 to 40 characters selected from the following:

- Digits 0 through 9
- Plus sign
- Minus sign
- Decimal point

There are three types of numeric literals: integer, decimal, and floating point.

| Type | Description |
|---|---|
| Integer Literal | An integer literal declares literal strings of integer numbers. Integer literals consist of an optional sign followed by a sequence of up to 10 digits. |
| | A numeric literal that is outside the range of values of an INTEGER is considered a decimal literal. |
| | Hexadecimal integer literals also represent integer values. A hexadecimal integer literal specifies a string of 0 to 62000 hexadecimal digits enclosed with apostrophes followed by the characters XI1 for a BYTEINT, XI2 for a SMALLINT, XI4 for an INTEGER, or XI8 for a BIGINT. |
| Decimal Literal | A decimal literal declares literal strings of decimal numbers. |
| | Decimal literals consist of the following components, reading from left-to-right: an optional sign, an optional sequence of up to 38 digits (mandatory only when no digits appear after the decimal point), an optional decimal point, an optional sequence of digits (mandatory only when no digits appear before the decimal point). The scale and precision of a decimal literal are determined by the total number of digits in the literal and the number of digits to the right of the decimal point, respectively. |
| Floating Point Literal | A floating point literal declares literal strings of floating point numbers. |
| | Floating point literals consist of the following components, reading from left-to-right: an optional sign, an optional sequence of digits (mandatory only when no digits appear after the decimal point) representing the whole number portion of the mantissa, an optional decimal point, an optional sequence of digits (mandatory only when no digits appear before the decimal point) representing the fractional portion of the mantissa, the literal character E, an optional sign, a sequence of digits representing the exponent. |

## DateTime Literals

Date and time literals declare date, time, or timestamp values in a SQL expression, view or macro definition text, or CONSTRAINT definition text.

Date and time literals are introduced by keywords. For example:

```
DATE '1969-12-23'
```

There are three types of DateTime literals: DATE, TIME, and TIMESTAMP.

| Type | Description |
|---|---|
| DATE Literal | A date literal declares a date value in ANSI DATE format. ANSI DATE literal is the preferred format for DATE constants. All DATE operations accept this format. |
| TIME Literal | A time literal declares a time value and an optional time zone offset. |
| TIMESTAMP Literal | A timestamp literal declares a timestamp value and an optional time zone offset. |

### Interval Literals

Interval literals provide a means for declaring spans of time.

Interval literals are introduced and followed by keywords. For example:

```
INTERVAL '200' HOUR
```

There are two mutually exclusive categories of interval literals: Year-Month and Day-Time.

| Category | Type | Description |
|---|---|---|
| Year-Month | • YEAR<br>• YEAR TO MONTH<br>• MONTH | Represent a time span that can include a number of years and months. |
| Day-Time | • DAY<br>• DAY TO HOUR<br>• DAY TO MINUTE<br>• DAY TO SECOND<br>• HOUR<br>• HOUR TO MINUTE<br>• HOUR TO SECOND<br>• MINUTE<br>• MINUTE TO SECOND<br>• SECOND | Represent a time span that can include a number of days, hours, minutes, or seconds. |

### Character Literals

A character literal declares a character value in an expression, view or macro definition text, or CHECK constraint definition text.

| Type | Description |
|---|---|
| Character literal | Character literals consist of 0 to 31000 bytes delimited by a matching pair of apostrophes. A zero-length character literal is represented by two consecutive apostrophes (''). |

| Type | Description |
|------|-------------|
| Hexadecimal character literal | A hexadecimal character literal specifies a string of 0 to 62000 hexadecimal digits enclosed with apostrophes followed by the characters XCF for a CHARACTER data type or XCV for a VARCHAR data type. |
| Unicode character string literal | Unicode character string literals consist of 0 to 31000 Unicode characters and are useful for inserting character strings containing characters that cannot generally be entered directly from a keyboard. |

## Graphic Literals

A graphic literal specifies multibyte characters within the graphic repertoire.

## Period Literals

A period literal specifies a constant value of a Period data type. Period literals are introduced by the PERIOD keyword. For example:

```
PERIOD '(2008-01-01, 2008-02-01)'
```

The element type of a period literal (DATE, TIME, or TIMESTAMP) is derived from the format of the DateTime values specified in the string literal.

## Related Topics

| For more information on, for example… | See … |
|----------------------------------------|-------|
| data type conversions | *SQL Functions, Operators, Expressions, and Predicates.* |
| query expressions | *SQL Data Manipulation Language.* |
| numeric, DateTime, interval, character, graphic, period, object name and hexadecimal literals | *SQL Data Types and Literals.* |
| • aggregate functions<br>• arithmetic expressions<br>• built-in functions<br>• byte functions<br>• calendar functions<br>• CASE expressions<br>• comparison operators and functions<br>• DateTime functions and expressions<br>• interval expressions<br>• logical predicates<br>• ordered analytical functions<br>• set operators<br>• string operators and functions<br>• user-defined functions | *SQL Functions, Operators, Expressions, and Predicates.* |

# Functions

## Scalar Functions

Scalar functions take input parameters and return a single value result. Some examples of standard SQL scalar functions are CHARACTER_LENGTH, POSITION, and SUBSTRING.

## Aggregate Functions

Aggregate functions produce summary results. They differ from scalar functions in that they take grouped sets of relational data, make a pass over each group, and return one result for the group. Some examples of standard SQL aggregate functions are AVG, SUM, MAX, and MIN.

## Built-In Functions

The built-in functions, or special register functions, which have no arguments, return various information about the system and can be used like other literals within SQL expressions. In an SQL query, the appropriate system value is substituted by the Parser after optimization but prior to executing a query using a cachable plan.

Available built-in functions include all of the following:

| | | |
|---|---|---|
| • ACCOUNT | • CURRENT_USER | • ROLE |
| • CURRENT_DATE | • DATABASE | • SESSION |
| • CURRENT_ROLE | • DATE | • TIME |
| • CURRENT_TIME | • PROFILE | • USER |
| • CURRENT_TIMESTAMP | | |

## Related Topics

For the names, parameters, return values, and other details of scalar and aggregate functions, see *SQL Functions, Operators, Expressions, and Predicates*.

For information on clauses and phrases, see *SQL Functions, Operators, Expressions, and Predicates*.

# Operators

SQL operators express logical and arithmetic operations. Operators of the same precedence are evaluated from left to right. See "SQL Operations and Precedence" on page 98 for more detailed information.

Parentheses can be used to control the order of precedence. When parentheses are present, operations are performed from the innermost set of parentheses outward.

The following definitions apply to SQL operators.

| Term | Definition |
|------|-----------|
| numeric | Any literal, data reference, or expression having a numeric value. |
| string | Any character string or string expression. |
| logical | A Boolean expression (resolves to TRUE, FALSE, or unknown). |
| value | Any numeric, character, or byte data item. |
| set | A collection of values returned by a subquery, or a list of values separated by commas and enclosed by parentheses. |

## SQL Operations and Precedence

SQL operations, and the order in which they are performed when no parentheses are present, appear in the following table. Operators of the same precedence are evaluated from left to right.

| Precedence | Result Type | Operation |
|------------|-------------|-----------|
| highest | numeric | + numeric   (unary plus) |
| | | - numeric   (unary minus) |
| intermediate | numeric | numeric ** numeric   (exponentiation) |
| | numeric | numeric * numeric   (multiplication) |
| | | numeric / numeric   (division) |
| | | numeric MOD numeric   (modulo operator) |
| | numeric | numeric + numeric   (addition) |
| | | numeric - numeric   (subtraction) |
| | string | concatenation operator |
| | logical | value EQ value |
| | | value NE value |
| | | value GT value |
| | | value LE value |
| | | value LT value |
| | | value GE value |
| | | value IN set |
| | | value NOT IN set |
| | | value BETWEEN value AND value |
| | | character value LIKE character value |
| | logical | NOT logical |
| | logical | logical AND logical |

| Precedence | Result Type | Operation |
|---|---|---|
| lowest | logical | logical OR logical |

# Separators

## Lexical Separators

A lexical separator is a character string that can be placed between words, literals, and delimiters without changing the meaning of a statement.

Valid lexical separators are any of the following.

- Comments

  For an explanation of comment lexical separators, see .
- Pad characters (several pad characters are treated as a single pad character except in a string literal)
- RETURN characters (X'0D')

## Statement Separators

The SEMICOLON is a Teradata SQL statement separator.

Each statement of a multistatement request must be separated from any subsequent statement with a semicolon.

The following multistatement request illustrates the semicolon as a statement separator.

```
SHOW TABLE Payroll_Test ; INSERT INTO Payroll_Test
(EmpNo, Name, DeptNo) VALUES ('10044', 'Jones M',
'300') ; INSERT INTO ...
```

For statements entered using BTEQ, a request terminates with an input line-ending semicolon unless that line has a comment, beginning with two dashes (- -). Everything to the right of the - - is a comment. In this case, the semicolon must be on the following line.

The SEMICOLON as a statement separator in a multistatement request is a Teradata extension to the ANSI/ISO SQL:2011 standard.

# Delimiters

Delimiters are special characters having meanings that depend on context.

The function of each delimiter appears in the following table.

| Delimiter | Name | Purpose |
|---|---|---|
| (<br>) | LEFT PARENTHESIS<br><br>RIGHT PARENTHESIS | Group expressions and define the limits of various phrases. |
| , | COMMA | Separates and distinguishes column names in the select list, or column names or parameters in an optional clause, or DateTime fields in a DateTime type. |
| : | COLON | Prefixes reference parameters or client system variables.<br><br>Also separates DateTime fields in a DateTime type. |
| . | FULLSTOP | • Separates database names from table, trigger, UDF, UDT, and stored procedure names, such as *personnel.employee*.<br>• Separates table names from a particular column name, such as *employee.deptno*).<br>• In numeric constants, the period is the decimal point.<br>• Separates DateTime fields in a DateTime type.<br>• Separates a method name from a UDT expression in a method invocation. |
| ; | SEMICOLON | • Separates statements in multi-statement requests.<br>• Separates statements in a stored procedure body.<br>• Separates SQL procedure statements in a triggered SQL statement in a trigger definition.<br>• Terminates requests submitted via utilities such as BTEQ.<br>• Terminates embedded SQL statements in C or PL/I applications. |
| ' | APOSTROPHE | • Defines the boundaries of character string constants.<br>• To include an APOSTROPHE character or show possession in a title, double the APOSTROPHE characters.<br>• Also separates DateTime fields in a DateTime type. |
| " | QUOTATION MARKS | Defines the boundaries of nonstandard names. |
| / | SOLIDUS | Separates DateTime fields in a DateTime type. |
| B<br>b | Uppercase B<br>Lowercase b | |
| - | HYPHEN-MINUS | |

## Example

In the following statement submitted through BTEQ, the FULLSTOP separates the database name (Examp and Personnel) from the table name (Profiles and Employee), and, where

reference is qualified to avoid ambiguity, it separates the table name (Profiles, Employee) from the column name (DeptNo).

```
UPDATE Examp.Profiles SET FinGrad = 'A'
WHERE Name = 'Phan A' ; SELECT EdLev, FinGrad,JobTitle,
YrsExp FROM Examp.Profiles, Personnel.Employee
WHERE Profiles.DeptNo = Employee.DeptNo ;
```

The first SEMICOLON separates the UPDATE statement from the SELECT statement. The second SEMICOLON terminates the entire multistatement request.

The semicolon is required in Teradata SQL to separate multiple statements in a request and to terminate a request submitted through BTEQ.

# Comments

You can embed comments within an SQL request anywhere a pad character can occur.

The SQL Parser and the preprocessor recognize the following types of ANSI/ISO SQL:2011-compliant embedded comments:

- Simple
- Bracketed

## Simple Comments

The simple form of a comment is delimited by two consecutive HYPHEN-MINUS (U+002D) characters (--) at the beginning of the comment and the newline character at the end of the comment.

── -- ── *comment_text* ──── *new_line_character* ─────────────────────────

1101E231

The newline character is implementation-specific, but is typed by pressing the Enter (non-3270 terminals) or Return (3270 terminals) key.

Simple SQL comments cannot span multiple lines.

### Examples

The following examples illustrate the use of a simple comment at the end of a line, at the beginning of a line, and at the beginning of a statement:

```
SELECT EmpNo, Name FROM Payroll_Test
ORDER BY Name -- Simple comment at the end of a line
;

SELECT EmpNo, Name FROM Payroll_Test
-- Simple comment at the beginning of a line
ORDER BY Name;

-- Simple comment at the beginning of a statement
SELECT EmpNo, Name FROM Payroll_Test
```

```
ORDER BY Name;
```

## Bracketed Comments

A bracketed comment is a text string of unrestricted length that is delimited by the beginning comment characters SOLIDUS (U+002F) and ASTERISK (U+002A) /* and the end comment characters ASTERISK and SOLIDUS */.

── /* ── *comment_text* ── */ ──────────────────────────

1101E230

Bracketed comments can begin anywhere on an input line and can span multiple lines.

### Examples

The following examples illustrate the use of a bracketed comment at the end of a line, in the middle of a line, at the beginning of a line, and at the beginning of a statement:

```
SELECT EmpNo, Name FROM Payroll_Test /* This bracketed comment starts
                                        at the end of a line
                                        and spans multiple lines. */
ORDER BY Name;

SELECT EmpNo, Name FROM Payroll_Test
/* This bracketed comment starts
   at the beginning of a line
   and spans multiple lines. */
ORDER BY Name;

/* This bracketed comment starts
   at the beginning of a statement
   and spans multiple lines. */
SELECT EmpNo, Name FROM Payroll_Test
ORDER BY Name;

SELECT EmpNo, Name
FROM  /* This comment is in the middle of a line. */ Payroll_Test
ORDER BY Name;

SELECT EmpNo, Name FROM Payroll_Test /* This bracketed
   comment starts at the end of a line, spans multiple
   lines, and ends in the middle of a line. */ ORDER BY Name;

SELECT EmpNo, Name FROM Payroll_Test
/* This bracketed comment starts at the beginning of a line,
   spans multiple lines, and ends in the
   middle of a line. */ ORDER BY Name;
```

## Comments With Multibyte Character Set Strings

You can include multibyte character set strings in both simple and bracketed comments.

When using mixed mode in comments, you must have a properly formed mixed mode string, which means that a Shift-In (SI) must follow its associated Shift-Out (SO).

If an SI does not follow the multibyte string, the results are unpredictable.

When using bracketed comments that span multiple lines, the SI must be on the same line as its associated SO. If the SI and SO are not on the same line, the results are unpredictable.

You must specify the bracketed comment delimiters (/* and */) as single byte characters.

# Terminators

The SEMICOLON is a Teradata SQL request terminator when it is the last nonblank character on an input line in BTEQ unless that line has a comment beginning with two dashes. In this case, the SEMICOLON request terminator must be on the line following the comment line.

A request is considered complete when either the "End of Text" character or the request terminator character is detected.

## ANSI Compliance

The SEMICOLON as a request terminator is a Teradata extension to the ANSI/ISO SQL:2011 standard.

## Example

On the following input line:

```
SELECT *
FROM Employee ;
```

the SEMICOLON terminates the single-statement request "SELECT * FROM Employee".

BTEQ uses SEMICOLONs to terminate multistatement requests.

A request terminator is mandatory for request types that are:

*   In the body of a macro
*   Triggered action statements in a trigger definition
*   Entered using the BTEQ interface
*   Entered using other interfaces that require BTEQ

## Example 1: Macro Request

The following statement illustrates the use of a request terminator in the body of a macro.

```
CREATE MACRO Test_Pay (number (INTEGER),
                       name (VARCHAR(12)),
                       dept (INTEGER) AS
( INSERT INTO Payroll_Test (EmpNo, Name, DeptNo)
  VALUES (:number, :name, :dept) ;
  UPDATE DeptCount
  SET EmpCount = EmpCount + 1 ;
  SELECT *
  FROM DeptCount ; )
```

## Example 2: BTEQ Request

When entered through BTEQ, the entire CREATE MACRO statement must be terminated.

```
CREATE MACRO Test_Pay
(number  (INTEGER),
 name    (VARCHAR(12)),
 dept    (INTEGER) AS
(INSERT INTO Payroll_Test (EmpNo, Name, DeptNo)
 VALUES (:number, :name, :dept) ;
 UPDATE DeptCount
 SET EmpCount = EmpCount + 1 ;
 SELECT *
 FROM DeptCount ; ) ;
```

# The Default Database

The default database is a Teradata extension to SQL that defines a database that Teradata Database uses to look for unqualified names, such as table, view, trigger, or macro names, in SQL statements.

The default database is not the only database that Teradata Database uses to find an unqualified name in an SQL statement. Teradata Database also looks for the name in:

- Other databases, if any, referenced by the SQL statement
- The login user database for a volatile table, if the unqualified object name is a table name
- The SYSLIB database, if the unqualified object name is a C or C++ UDF that is not in the default database

If the unqualified object name exists in more than one of the databases in which Teradata Database looks, the SQL statement produces an ambiguous name error.

## Establishing a Permanent Default Database

You can establish a permanent default database that is invoked each time you log on.

| TO ... | USE one of the following SQL Data Definition statements... |
|---|---|
| define a permanent default database | • CREATE USER, with a DEFAULT DATABASE clause.<br>• CREATE USER, with a PROFILE clause that specifies a profile that defines the default database. |
| change your permanent default database definition | • MODIFY USER, with a DEFAULT DATABASE clause.<br>• MODIFY USER, with a PROFILE clause. |
| add a default database when one had not been established previously | • MODIFY PROFILE, with a DEFAULT DATABASE clause. |

For example, the following statement automatically establishes Personnel as the default database for Marks at the next logon:

```
MODIFY USER marks AS
DEFAULT DATABASE = personnel ;
```

After you assign a default database, Teradata Database uses that database as one of the databases to look for all unqualified object references.

To obtain information from a table, view, trigger, or macro in another database, fully qualify the table reference by specifying the database name, a FULLSTOP character, and the table name.

## Establishing a Default Database for a Session

You can establish a default database for the current session that Teradata Database uses to look for unqualified object names in SQL statements.

| TO ... | USE ... |
|---|---|
| establish a default database for a session | the DATABASE statement. |

For example, after entering the following SQL statement:

```
DATABASE personnel ;
```

you can enter a SELECT statement:

```
SELECT deptno (TITLE 'Org'), name
FROM employee ;
```

which has the same results as:

```
SELECT deptno (TITLE 'Org'), name
FROM personnel.employee;
```

To establish a default database, you must have some privilege on an object in that database. Once defined, the default database remains in effect until the end of a session or until it is replaced by a subsequent DATABASE statement.

## Default Database for a Stored Procedure

Stored procedures can contain SQL statements with unqualified object references. The default database that Teradata Database uses for the unqualified object references depends on whether the CREATE PROCEDURE statement includes the SQL SECURITY clause and, if it does, which option the SQL SECURITY clause specifies.

For details on whether Teradata Database uses the creator, invoker, or owner of the stored procedure as the default database, see CREATE PROCEDURE in *SQL Data Definition Language*.

## Related Topics

| For more information on … | See … |
|---|---|
| the DATABASE statement | *SQL Data Definition Language.* |
| using the CREATE PROFILE statement to define the default database for member users | |
| the CREATE USER statement | |
| the MODIFY USER statement | |
| fully-qualified names | "Referencing Object Names in a Request" on page 90. |
| | "Unqualified Object Names" on page 91. |

# Null Statements

A null statement has no content except for optional pad characters or SQL comments.

## Example 1

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 2

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 3

A SEMICOLON that precedes the first (or only) statement of a request is taken as a null statement.

```
;DROP TABLE temp_payroll;
```

# NULL Keyword as a Literal

A null represents either:

- An empty column
- An unknown value
- An unknowable value

Nulls are neither values nor do they signify values; they represent the absence of value. A null is a place holder indicating that no value is present.

## NULL Keyword

The keyword NULL represents null, and is sometimes available as a special construct similar to, but not identical with, a literal.

## ANSI Compliance

NULL is ANSI/ISO SQL:2011-compliant with extensions.

## Using NULL as a Literal

Use NULL as a literal in the following ways:

- A CAST source operand, for example:

  ```
  SELECT CAST (NULL AS DATE);
  ```
- A CASE result, for example.

  ```
  SELECT CASE WHEN orders = 10 THEN NULL END FROM sales_tbl;
  ```
- An insert item specifying a null is to be placed in a column position on INSERT.
- An update item specifying a null is to be placed in a column position on UPDATE.
- A default column definition specification, for example:

  ```
  CREATE TABLE European_Sales
     (Region INTEGER DEFAULT 99
     ,Sales Euro_Type DEFAULT NULL);
  ```
- An explicit SELECT item, for example:

  ```
  SELECT NULL
  ```
  This is a Teradata extension to ANSI.
- An operand of a function, for example:

  ```
  SELECT TYPE(NULL)
  ```
  This is a Teradata extension to ANSI.

## Data Type of NULL

When you use NULL as an explicit SELECT item or as the operand of a function, its data type is INTEGER. In all other cases NULL has no data type because it has no value.

For example, if you perform SELECT TYPE(NULL), then INTEGER is returned as the data type of NULL.

To avoid type issues, cast NULL to the desired type.

## Related Topics

For information on the behavior of nulls and how to use them in data manipulation statements, see "Manipulating Nulls" on page 141.

# CHAPTER 3 SQL Data Definition, Control, and Manipulation

This chapter describes the functional families of the SQL language.

# SQL Functional Families and Binding Styles

The SQL language can be characterized in several different ways. This chapter is organized around functional groupings of the components of the language with minor emphasis on binding styles.

## Functional Family

SQL provides facilities for defining database objects, for defining user access to those objects, and for manipulating the data stored within them.

The following list describes the principal functional families of the SQL language.

- SQL Data Definition Language (DDL)
- SQL Data Control Language (DCL)
- SQL Data Manipulation Language (DML)
- Query and Workload Analysis Statements
- Help and Database Object Definition Tools

Some classifications of SQL group the data control language statements with the data definition language statements.

## Binding Style

The ANSI/ISO SQL standards do not define the term binding style. The expression refers to a possible method by which an SQL statement can be invoked.

Teradata Database supports the following SQL binding styles:

- Direct, or interactive
- Embedded SQL
- Stored procedure
- SQL Call Level Interface (as ODBC)
- JDBC

The direct binding style is usually not qualified in this book set because it is the default style. Embedded SQL and stored procedure binding styles are always clearly specified, either explicitly or by context.

## Related Topics

You can find more information on binding styles in the SQL book set and in other books.

| For more information on … | See … |
| --- | --- |
| embedded SQL | • *Teradata Preprocessor2 for Embedded SQL Programmer Guide*<br>• *SQL Stored Procedures and Embedded SQL* |
| stored procedures | *SQL Stored Procedures and Embedded SQL* |
| ODBC | *ODBC Driver for Teradata User Guide* |
| JDBC | *Teradata JDBC Driver User Guide* |

# Data Definition Language

The SQL Data Definition Language (DDL) is a subset of the SQL language and consists of all SQL statements that support the definition of database objects.

## Purpose

Data definition language statements perform the following functions:

- Create, drop, rename, alter, modify, and replace database objects
- Comment on database objects
- Collect statistics on a column set or index
- Establish a default database
- Set a different collation sequence, account priority, DateForm, time zone, and database for the session
- Set roles
- Set the query band for a session or transaction
- Begin and end logging
- Enable and disable online archiving for all tables in a database or a specific set of tables

## Rules on Entering DDL Statements

A DDL statement can be entered as:

- A single-statement request.
- The solitary statement, or the last statement, in an explicit transaction (in Teradata mode, one or more requests enclosed by user-supplied BEGIN TRANSACTION and END

TRANSACTION statement, or in ANSI mode, one or more requests ending with the COMMIT keyword).

• The solitary statement in a macro.

DDL statements cannot be entered as part of a multistatement request.

Successful execution of a DDL statement automatically creates and updates entries in the Data Dictionary.

## SQL DDL Statements

For detailed information about the function, syntax, and usage of Teradata SQL Data Definition statements, see *SQL Data Definition Language*.

# Altering Table Structure and Definition

You may need to change the structure or definition of an existing table or temporary table. In many cases, you can use ALTER TABLE and RENAME to make the changes. Some changes, however, may require you to use CREATE TABLE to recreate the table.

You cannot use ALTER TABLE on an error logging table.

## Making Changes to a Table

Use the RENAME TABLE statement to change the name of a table, temporary table, queue table, or error logging table.

Use the ALTER TABLE statement to perform any of the following functions:

• Add or drop columns on an existing table or temporary table

• Add column default control, FORMAT, and TITLE attributes on an existing table or temporary table

• Add or remove journaling options on an existing table or temporary table

• Add or remove the FALLBACK option on an existing table or temporary table

• Change the DATABLOCKSIZE or percent FREESPACE on an existing table or temporary table

• Add or drop column and table level constraints on an existing table or temporary table

• Change the LOG and ON COMMIT options for a global temporary table

• Modify referential constraints

• Change the properties of the primary index for a table (some cases require an empty table)

• Change the partitioning properties of the primary index for a table, including modifications to the partitioning expression defined for use by a partitioned primary index (some cases require an empty table)

• Regenerate table headers and optionally validate and correct the partitioning of PPI table rows

- Define, modify, or delete the COMPRESS attribute for an existing column
- Add the BLOCKCOMPRESSION option used to modify the temperature-based Block Level Compression (BLC) for a table.
- Change column attributes (that do not affect stored data) on an existing table or temporary table

Restrictions apply to many of the preceding modifications. For a complete list of rules and restrictions on using ALTER TABLE to change the structure or definition of an existing table, see *SQL Data Definition Language*.

To perform any of the following functions, use CREATE TABLE to recreate the table:

- Redefine the primary index or its partitioning for a non-empty table when not allowed for ALTER TABLE
- Change a data type attribute that affects existing data
- Add a column that would exceed the maximum lifetime column count

Interactively, the SHOW TABLE statement can call up the current table definition, which can then be modified and resubmitted to create a new table.

If the stored data is not affected by incompatible data type changes, an INSERT... SELECT statement can be used to transfer data from the existing table to the new table.

# Dropping and Renaming Objects

## Dropping Objects

To drop an object, use the appropriate DDL statement.

| Object | Use |
|---|---|
| Constraint | DROP CONSTRAINT |
| Error logging table | DROP ERROR TABLE |
| | DROP TABLE |
| Hash index | DROP HASH INDEX |
| Join index | DROP JOIN INDEX |
| Macro | DROP MACRO |
| Profile | DROP PROFILE |
| Role | DROP ROLE |
| Secondary index | DROP INDEX |
| Stored procedure | DROP PROCEDURE |

| Object | Use |
|---|---|
| Table | DROP TABLE |
| Global temporary table or volatile table | |
| Primary index | |
| Trigger | DROP TRIGGER |
| User-defined function | DROP FUNCTION |
| User-defined method | ALTER TYPE |
| User-defined type | DROP TYPE |
| View | DROP VIEW |

## Renaming Objects

Teradata SQL provides RENAME statements that you can use to rename some objects. To rename objects that do not have associated RENAME statements, you must first drop them and then recreate them with a new name, or, in the case of primary indexes, use ALTER TABLE.

| Object | Use |
|---|---|
| Hash index | DROP HASH INDEX and then CREATE HASH INDEX |
| Join index | DROP JOIN INDEX and then CREATE JOIN INDEX |
| Macro | RENAME MACRO |
| Primary index | ALTER TABLE |
| Profile | DROP PROFILE and then CREATE PROFILE |
| Role | DROP ROLE and then CREATE ROLE |
| Secondary index | DROP INDEX and then CREATE INDEX |
| Stored procedure | RENAME PROCEDURE |
| Table | RENAME TABLE |
| Global temporary table or volatile table | |
| Queue table | |
| Error logging table | |
| Trigger | RENAME TRIGGER |
| User-Defined Function | RENAME FUNCTION |
| User-Defined Method | ALTER TYPE and then CREATE METHOD |
| User-Defined Type | DROP TYPE and then CREATE TYPE |

| Object | Use |
|--------|-----|
| View | RENAME VIEW |

## Related Topics

For more information on these statements, including rules that apply to usage, see *SQL Data Definition Language.*

# Data Control Language

The SQL Data Control Language (DCL) is a subset of the SQL language and consists of all SQL statements that support the definition of security authorization for accessing database objects.

## Purpose

Data control statements perform the following functions:

- Grant and revoke privileges
- Give ownership of a database to another user

## Rules on Entering DCL Statements

A data control statement can be entered as:

- A single-statement request
- The solitary statement, or as the last statement, in an "explicit transaction" (one or more requests enclosed by user-supplied BEGIN TRANSACTION and END TRANSACTION statement in Teradata mode, or in ANSI mode, one or more requests ending with the COMMIT keyword).
- The solitary statement in a macro

A data control statement cannot be entered as part of a multistatement request.

Successful execution of a data control statement automatically creates and updates entries in the Data Dictionary.

## Teradata SQL DCL Statements

For detailed information about the function, syntax, and usage of Teradata SQL Data Control statements, see *SQL Data Control Language.*

# Data Manipulation Language

The SQL Data Manipulation Language (DML) is a subset of the SQL language and consists of all SQL statements that support the manipulation or processing of database objects.

## Selecting Columns

The SELECT statement returns information from the tables in a relational database. SELECT specifies the table columns from which to obtain the data, the corresponding database (if not defined by default), and the table (or tables) to be accessed within that database.

For example, to request the data from the name, salary, and jobtitle columns of the Employee table, type:

```
SELECT name, salary, jobtitle FROM employee ;
```

The response might be something like the following results table.

| Name | Salary | JobTitle |
|------|--------|----------|
| Newman P | 28600.00 | Test Tech |
| Chin M | 38000.00 | Controller |
| Aquilar J | 45000.00 | Manager |
| Russell S | 65000.00 | President |
| Clements D | 38000.00 | Salesperson |

Note:  The left-to-right order of the columns in a result table is determined by the order in which the column names are entered in the SELECT statement. Columns in a relational table are not ordered logically.

As long as a statement is otherwise constructed properly, the spacing between statement elements is not important as long as at least one pad character separates each element that is not otherwise separated from the next.

For example, the SELECT statement in the above example could just as well be formulated like this:

```
SELECT    name,    salary,jobtitle
FROM employee;
```

Notice that there are multiple pad characters between most of the elements and that a comma only (with no pad characters) separates column name salary from column name jobtitle.

To select all the data in the employee table, you could enter the following SELECT statement:

```
SELECT * FROM employee ;
```

The asterisk specifies that the data in all columns (except system-derived columns) of the table is to be returned.

## Selecting Rows

The SELECT statement retrieves stored data from a table. All rows, specified rows, or specific columns of all or specified rows can be retrieved. When used in a subquery, the SELECT statement can also select rows from a derived table or view.

The FROM, WHERE, ORDER BY, DISTINCT, WITH, GROUP BY, HAVING, and TOP clauses provide for a fine detail of selection criteria.

To obtain data from specific rows of a table, use the WHERE clause of the SELECT statement. That portion of the clause following the keyword WHERE causes a search for rows that satisfy the condition specified.

For example, to get the name, salary, and title of each employee in Department 100, use the WHERE clause:

```
SELECT name, salary, jobtitle FROM employee
   WHERE deptno = 100 ;
```

The response appears in the following table.

| Name | Salary | JobTitle |
|------|--------|----------|
| Chin M | 38000.00 | Controller |
| Greene W | 32500.00 | Payroll Clerk |
| Moffit H | 35000.00 | Recruiter |
| Peterson J | 25000.00 | Payroll Clerk |

To obtain data from a multirow result table in embedded SQL, declare a cursor for the SELECT statement and use it to fetch individual result rows for processing.

To obtain data from the row with the oldest timestamp value in a queue table, use the SELECT AND CONSUME statement, which also deletes the row from the queue table.

## Zero-Table SELECT

Zero-table SELECT statements return data but do not access tables.

For example, the following SELECT statement specifies an expression after the SELECT keyword that does not require a column reference or FROM clause:

```
SELECT 40000.00 / 52.;
```

The response is one row:

```
(40000.00/52.)
-----------------
          769.23
```

Here is another example that specifies an attribute function after the SELECT keyword:

```
SELECT TYPE(sales_table.region);
```

Because the argument to the TYPE function is a column reference that specifies the table name, a FROM clause is not required and the query does not access the table.

The response is one row that might be something like the following:

```
Type(region)
--------------------------------------
INTEGER
```

## Adding Rows

To add a new row to a table, use the INSERT statement. To perform a bulk insert of rows by retrieving the new row data from another table, use the INSERT … SELECT form of the statement.

Defaults and constraints defined by the CREATE TABLE statement affect an insert operation in the following ways.

| WHEN an INSERT statement … | THEN the system … |
|---|---|
| attempts to add a duplicate row<br>• for any unique index<br>• to a table defined as SET (not to allow duplicate rows) | returns an error, with one exception. The system silently ignores duplicate rows that an INSERT … SELECT would create when the:<br>• table is defined as SET<br>• mode is Teradata |
| omits a value for a column for which a default value is defined | stores the default value for that column. |
| omits a value for a column for which both of the following statements are true:<br>• NOT NULL is specified<br>• no default is specified | rejects the operation and returns an error message. |
| supplies a value that does not satisfy the constraints specified for a column or violates some defined constraint on a column or columns | |

If you are performing a bulk insert of rows using INSERT … SELECT, and you want Teradata Database to log errors that prevent normal completion of the operation, use the LOGGING ERRORS option. Teradata Database logs errors as error rows in an error logging table that you create with a CREATE ERROR TABLE statement.

## Updating Rows

To modify data in one or more rows of a table, use the UPDATE statement. In the UPDATE statement, you specify the column name of the data to be modified along with the new value. You can also use a WHERE clause to qualify the rows to change.

Attributes specified in the CREATE TABLE statement affect an update operation in the following ways:

- When an update supplies a value that violates some defined constraint on a column or columns, the update operation is rejected and an error message is returned.
- When an update supplies the value NULL and a NULL is allowed, any existing data is removed from the column.
- If the result of an UPDATE will violate uniqueness constraints or create a duplicate row in a table which does not allow duplicate rows, an error message is returned.

To update rows in a multirow result table in embedded SQL, declare a cursor for the SELECT statement and use it to fetch individual result rows for processing, then use a WHERE CURRENT OF clause in a positioned UPDATE statement to update the selected rows.

Teradata Database supports a special form of UPDATE, called the upsert form, which is a single SQL statement 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. See, however, "Merging Rows" on page 118.

## Deleting Rows

The DELETE statement allows you to remove an entire row or rows from a table. A WHERE clause qualifies the rows that are to be deleted.

For the way to delete rows in a multirow result table in embedded SQL, see *SQL Stored Procedures and Embedded SQL.*

## Merging Rows

The MERGE statement merges a source row set into a target table based on whether any target rows satisfy a specified matching condition with the source row. The MERGE statement is a single SQL statement that includes both UPDATE and INSERT functionality.

| IF the source and target rows … | THEN the merge operation is an … |
| --- | --- |
| satisfy the matching condition | update based on the specified WHEN MATCHED THEN UPDATE clause. |
| do not satisfy the matching condition | insert based on the specified WHEN NOT MATCHED THEN INSERT clause. |

If you are performing a bulk insert or update of rows using MERGE, and you want Teradata Database to log errors that prevent normal completion of the operation, use the LOGGING ERRORS option. Teradata Database logs errors as error rows in an error logging table that you create with a CREATE ERROR TABLE statement.

## Related Topics

For details about selecting, adding, updating, deleting, and merging rows, see *SQL Data Manipulation Language.*

# Subqueries

Subqueries are nested SELECT statements. They can be used to ask a series of questions to arrive at a single answer.

## Example: Three Level Subqueries

The following subqueries, nested to three levels, answer the question "Who manages the manager of Marston?"

```
SELECT Name
FROM Employee
WHERE EmpNo IN
    (SELECT MgrNo
     FROM Department
     WHERE DeptNo IN
        (SELECT DeptNo
         FROM Employee
         WHERE Name = 'Marston A') ) ;
```

The subqueries that pose the questions leading to the final answer are inverted:

*   The third subquery asks the Employee table for the number of Marston's department.
*   The second subquery asks the Department table for the employee number (MgrNo) of the manager associated with this department number.
*   The first subquery asks the Employee table for the name of the employee associated with this employee number (MgrNo).

The result table looks like the following:

```
Name
--------
Watson L
```

This result can be obtained using only two levels of subquery, as the following example shows.

```
SELECT Name
FROM Employee
WHERE EmpNo IN
 (SELECT MgrNo
  FROM Department, Employee
  WHERE Employee.Name = 'Marston A'
  AND Department.DeptNo = Employee.DeptNo) ;
```

In this example, the second subquery defines a join of Employee and Department tables.

This result could also be obtained using a one-level query that uses correlation names, as the following example shows.

```
SELECT M.Name
FROM Employee M, Department D, Employee E
WHERE M.EmpNo = D.MgrNo AND
      E.Name = 'Marston A' AND
      D.DeptNo = E.DeptNo;
```

In some cases, as in the preceding example, the choice is a style preference. In other cases, correct execution of the query may require a subquery.

## For More Information

For more information, see *SQL Data Manipulation Language*.

# 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:

1  Initialization

2  Recursion, or repeated iteration of the logic through the hierarchy

3  Termination

Similarly, a recursive query has three execution phases:

1  Create an initial result set.

2  Recursion based on the existing result set.

3  Final query to return the final result set.

## Specifying 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

## Using the WITH RECURSIVE Clause

Consider the following employee table:

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

The table represents an organizational structure containing a hierarchy of employee-manager data.

The following figure depicts what the employee table looks like hierarchically.

```
employee # = 801
manager employee # = NULL
```

```
employee # = 1003
manager employee # = 801
```

```
employee # = 1019
manager employee # = 801
```

```
employee # = 1016
manager employee # = 801
```

```
employee # = 1008
manager employee # = 1019
```

```
employee # = 1006
manager employee # = 1019
```

```
employee # = 1014
manager employee # = 1019
```

```
employee # = 1011
manager employee # = 1019
```

```
employee # = 1010
manager employee # = 1003
```

```
employee # = 1001
manager employee # = 1003
```

```
employee # = 1004
manager employee # = 1003
```

```
employee # = 1012
manager employee # = 1004
```

```
employee # = 1002
manager employee # = 1004
```

```
employee # = 1015
manager employee # = 1004
```

1101A285

The following recursive query retrieves the employee numbers of all employees who directly or indirectly report to the manager with employee_number 801:

```
WITH RECURSIVE temp_table (employee_number) AS
( SELECT root.employee_number
  FROM employee root
  WHERE root.manager_employee_number = 801
UNION ALL
  SELECT indirect.employee_number
  FROM temp_table direct, employee 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 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 direct, employee 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 clause and extracts the employee information out of temp_table:

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

Here are the results of the recursive query:

```
employee_number
---------------
           1001
           1002
           1003
           1004
           1006
           1008
           1010
           1011
           1012
           1014
           1015
           1016
           1019
```

## Using the RECURSIVE Clause in a CREATE VIEW Statement

Creating a view using the RECURSIVE clause is similar to preceding a query with the WITH RECURSIVE clause.

Consider the employee table that was presented in "Using the WITH RECURSIVE Clause" on page 120. The following statement creates a view named hierarchy_801 using a recursive query that retrieves the employee numbers of all employees who directly or indirectly report to the manager with employee_number 801:

```
CREATE RECURSIVE VIEW hierarchy_801 (employee_number) AS
( SELECT root.employee_number
  FROM employee root
  WHERE root.manager_employee_number = 801
UNION ALL
  SELECT indirect.employee_number
  FROM hierarchy_801 direct, employee indirect
  WHERE direct.employee_number = indirect.manager_employee_number
);
```

The seed statement and recursive statement in the view definition are the same as the seed statement and recursive statement in the previous recursive query that uses the WITH RECURSIVE clause, except that the hierarchy_801 view name is different from the temp_table temporary result name.

To extract the employee information, use the following SELECT statement on the hierarchy_801 view:

```
SELECT * FROM hierarchy_801 ORDER BY employee_number;
```

Here are the results:

```
employee_number
---------------
           1001
           1002
           1003
           1004
           1006
           1008
           1010
           1011
           1012
           1014
           1015
           1016
           1019
```

## Depth Control to Avoid Infinite Recursion

If the hierarchy is cyclic, or if the recursive statement specifies a bad join condition, a recursive query can produce a runaway query that never completes with a finite result. The best practice is to control the depth of the recursion:

- Specify a depth control column in the column list of the WITH RECURSIVE clause or recursive view
- Initialize the column value to 0 in the seed statements
- Increment the column value by 1 in the recursive statements
- Specify a limit for the value of the depth control column in the join condition of the recursive statements

Here is an example that modifies the previous recursive query that uses the WITH RECURSIVE clause of the employee table to limit the depth of the recursion to five cycles:

```
WITH RECURSIVE temp_table (employee_number, depth) AS
( SELECT root.employee_number, 0 AS depth
  FROM employee root
  WHERE root.manager_employee_number = 801
UNION ALL
  SELECT indirect.employee_number, direct.depth+1 AS newdepth
  FROM temp_table direct, employee indirect
  WHERE direct.employee_number = indirect.manager_employee_number
  AND newdepth <= 5
)
SELECT * FROM temp_table ORDER BY employee_number;
```

## Related Topics

| For details on … | See … |
|---|---|
| recursive queries | "WITH RECURSIVE" in *SQL Data Manipulation Language*. |
| recursive views | "CREATE VIEW" in *SQL Data Definition Language*. |

# Query and Workload Analysis Statements

## Data Collection and Analysis

Teradata provides the following SQL statements for collecting and analyzing query and data demographics and statistics:

- BEGIN QUERY LOGGING
- COLLECT DEMOGRAPHICS
- COLLECT STATISTICS
- DROP STATISTICS
- DUMP EXPLAIN
- END QUERY LOGGING

- INITIATE INDEX ANALYSIS
- INITIATE PARTITION ANALYSIS
- INSERT EXPLAIN
- RESTART INDEX ANALYSIS
- SHOW QUERY LOGGING

Collected data can be used in several ways, for example:

- By the Optimizer, to produce the best query plans possible.
- To populate user-defined Query Capture Database (QCD) tables with data used by various utilities to analyze query workloads as part of the ongoing process to re-engineer the database design process.

  For example, the Teradata Index Wizard determines optimal secondary indexes, single-table join indexes, and PPIs to support the query workloads you ask it to analyze.

## Index Analysis and Target Level Emulation

Teradata also provides diagnostic statements that support the Teradata Index Wizard and the cost-based and sample-based components of the target level emulation facility used to emulate a production environment on a test system.

For information on Optimizer cost profiles, see *SQL Request and Transaction Processing*.

Note:  Settings should be changed in the production environment only under the direction of Teradata Support Center personnel.

## Related Topics

For details on BEGIN QUERY LOGGING, END QUERY LOGGING, and SHOW QUERY LOGGING, see *SQL Data Definition Language*. For more information on other query and workload analysis statements, see *SQL Data Manipulation Language*.

For more information on the Teradata Index Wizard, see *Teradata Index Wizard User Guide*.

# Help and Database Object Definition Tools

Teradata SQL provides several powerful tools to get help about database object definitions and summaries of database object definition statement text.

## HELP Statements

Various HELP statements return reports about the current column definitions for named database objects. The reports these statements return can be useful to database designers who need to fine tune index definitions, column definitions (for example, changing data typing to eliminate the necessity of ad hoc conversions), and so on.

For SQL HELP statements, see *SQL Data Definition Language Syntax and Examples*.

## SHOW Statements

Most SHOW statements return a CREATE statement indicating the last data definition statement performed against the named database object. Some SHOW statements, such as SHOW QUERY LOGGING, return other information. These statements are particularly useful for application developers who need to develop exact replicas of existing objects for purposes of testing new software.

For SQL SHOW statements, see *SQL Data Definition Language Syntax and Examples*.

## Incompatible Characters in HELP and SHOW Output

## Example

Consider the following definition for a table named department:

```
CREATE TABLE department, FALLBACK
   (department_number SMALLINT
   ,department_name CHAR(30) NOT NULL
   ,budget_amount DECIMAL(10,2)
   ,manager_employee_number INTEGER
   )
 UNIQUE PRIMARY INDEX (department_number)
,UNIQUE INDEX (department_name);
```

To get the attributes for the table, use the HELP TABLE statement:

```
HELP TABLE department;
```

The HELP TABLE statement returns:

```
Column Name                      Type Comment
------------------------------- ---- -------------------------
department_number                I2   ?
department_name                  CF   ?
budget_amount                    D    ?
manager_employee_number          I    ?
```

To get the CREATE TABLE statement that defines the department table, use the SHOW TABLE statement:

```
SHOW TABLE department;
```

The SHOW TABLE statement returns:

```
CREATE SET TABLE TERADATA_EDUCATION.department, FALLBACK,
   NO BEFORE JOURNAL,
   NO AFTER JOURNAL,
   CHECKSUM = DEFAULT
   (department_number SMALLINT,
    department_name CHAR(30) CHARACTER SET LATIN
                         NOT CASESPECIFIC NOT NULL,
    budget_amount DECIMAL(10,2),
    manager_employee_number INTEGER)
UNIQUE PRIMARY INDEX ( department_number )
UNIQUE INDEX ( department_name );
```

This chapter describes the fundamentals of Teradata Database data handling.

# Invoking SQL Statements

SQL provides several ways to invoke an executable SQL statement:

- Interactively from a terminal
- Embedded within an application program
- Dynamically performed from within an embedded application
- Embedded within a stored procedure or external stored procedure

## Executable SQL Statements

An executable SQL statement performs an action. The action can be on data or on a transaction or some other entity at a higher level than raw data.

Some examples of executable SQL statements include:

- SELECT
- CREATE TABLE
- COMMIT
- CONNECT
- PREPARE

Most executable SQL statements can be performed interactively from a terminal using an SQL query manager like BTEQ or Teradata SQL Assistant.

The following types of executable SQL commands cannot be performed interactively:

- Cursor control and declaration statements
- Dynamic SQL control statements
- Stored procedure control statements and condition handlers
- Connection control statements
- Special forms of SQL statements such as SELECT INTO

These statements can only be used within an embedded SQL or stored procedure application.

## Nonexecutable SQL Statements

A nonexecutable SQL statement is one that declares an SQL statement, object, or host or local variable to the preprocessor or stored procedure compiler. Nonexecutable SQL statements are not processed during program execution.

Examples of nonexecutable SQL statements for embedded SQL applications include:

- DECLARE CURSOR
- BEGIN DECLARE SECTION
- END DECLARE SECTION
- EXEC SQL

Examples of nonexecutable SQL statements for stored procedures include:

- DECLARE CURSOR
- DECLARE

## SQL Requests

A request to Teradata Database can span any number of input lines. Teradata Database can receive and perform SQL statements that are:

- Embedded in a client application program that is written in a procedural language.
- Embedded in a stored procedure.
- Entered interactively through BTEQ or Teradata SQL Assistant interfaces.
- Submitted in a BTEQ script as a batch job.
- Submitted through other supported methods (such as CLIv2, ODBC, and JDBC).
- Submitted from a C or C++ external stored procedure using CLIv2 or a Java external stored procedure using JDBC.

# Transactions

A transaction is a logical unit of work where the statements nested within the transaction either execute successfully as a group or do not execute.

## Transaction Processing Mode

You can perform transaction processing in either of the following session modes:

- ANSI

  In ANSI session mode, transaction processing adheres to the rules defined by the ANSI/ISO SQL specification.

- Teradata

  In Teradata session mode, transaction processing follows the rules defined by Teradata prior to the emergence of the ANSI/ISO SQL standard.

To set the transaction processing mode, use the:

- SessionMode field of the DBS Control Record
- BTEQ command SET SESSION TRANSACTION
- Preprocessor2 TRANSACT() option
- ODBC SessionMode option in the .odbc.ini file
- JDBC TeraDataSource.setTransactMode() method

For detailed information on statement and transaction processing, see *SQL Request and Transaction Processing*.

# Transaction Processing in Teradata Session Mode

A Teradata SQL transaction can be a single Teradata SQL statement, or a sequence of Teradata SQL statements, treated as a single unit of work.

Each request is processed as one of the following transaction types:

- Implicit
- Explicit
- Two-phase commit (2PC)

## Implicit Transactions

An implicit transaction is a request that does not include the BEGIN TRANSACTION and END TRANSACTION statements. The implicit transaction starts and completes all within the SQL request: it is self-contained.

An implicit transaction can be a:

- Single DML statement that affects one or more rows of one or more tables.
- Macro or trigger containing one or more statements.
- Request containing multiple statements separated by SEMICOLON characters. SEMICOLON characters can appear anywhere in the input line. The Parser interprets a SEMICOLON at the end of an input line as the request terminator.

DDL statements are not valid in a multistatement request and are therefore not valid in an implicit multistatement transaction.

## Explicit Transactions

In Teradata session mode, an explicit transaction contains one or more statements enclosed by BEGIN TRANSACTION and END TRANSACTION statements. The first BEGIN TRANSACTION initiates a transaction and the last END TRANSACTION terminates the transaction.

When multiple statements are included in an explicit transaction, you can only specify a DDL statement if it is the last statement in the series.

## 2PC Rules

2PCprotocol is supported in Teradata session mode:

- A 2PC transaction contains one or more DML statements that affect multiple databases and are coordinated externally using the 2PC protocol.

- A DDL statement is not valid in a two-phase commit transaction.

# Transaction Processing in ANSI Session Mode

In ANSI mode, transactions are always implicitly started and explicitly closed.

A transaction initiates when one of the following happens:

- The first SQL statement in a session executes.
- The first statement following the close of a transaction executes.

The COMMIT or ROLLBACK/ABORT statements close a transaction.

If a transaction includes a DDL statement, it must be the last statement in the transaction. DATABASE and SET SESSION are DDL statements. See "Rollback Processing."

If a session terminates with an open transaction, any effects of that transaction are rolled back.

## Two-Phase Commit

Sessions in ANSI session mode do not support Two-Phase Commit (2PC). If an attempt is made to use the 2PC protocol in ANSI session mode, the Logon process aborts and an error returns to the requestor.

# Multistatement Requests

An atomic request containing more than one SQL statement, each terminated by a SEMICOLON character.

## Syntax



HH01A004

## ANSI Compliance

Multistatement requests are non-ANSI/ISO SQL:2011 standard.

## Rules and Restrictions

Teradata Database imposes restrictions on the use of multistatement requests:

- Only one USING modifier is permitted per request, so only one USING modifier can be used per multistatement request.

  This rule applies to interactive SQL only. Embedded SQL and stored procedures do not permit the USING modifier.

- A multistatement request cannot include a DDL statement.

  However, a multistatement request can include one SET QUERY_BAND … FOR TRANSACTION statement if it is the first statement in the request.

- The keywords BEGIN REQUEST and END REQUEST must delimit a multistatement request in a stored procedure.

## Power of Multistatement Requests

The multistatement request is application-independent. It improves performance for a variety of applications that can package more than one SQL statement at a time. BTEQ, CLI, and the SQL preprocessor all support multistatement requests.

Multistatement requests improve system performance by reducing processing overhead. By performing a series of statements as one request, performance for the client, the Parser, and the Database Manager are all enhanced.

Because of this reduced overhead, using multistatement requests also decreases response time. A multistatement request that contains 10 SQL statements could be as much as 10 times more efficient than the 10 statements entered separately (depending on the types of statements submitted).

## Implicit Multistatement Transaction

In Teradata session mode, a multistatement request is one form of implicit transaction. As such, the outcome of an implicit multistatement transaction is typically all-or-nothing. If one statement in the request fails, the entire implicit transaction fails and the system rolls it back.

## Statement Independence for Simple INSERTs

When a multistatement request includes only simple INSERT statements, a failure of one or more INSERTs does not cause the entire request to be rolled back. In these cases, errors are reported for the INSERT statements that failed, so those statements can be resubmitted. INSERT statements that completed successfully are not rolled back.

This behavior is limited to requests submitted directly to the database using an SQL INSERT multistatement request or submitted using a JDBC application request.

# Parallel Step Processing

Teradata Database can perform some requests in parallel (see "Parallel Steps" on page 133). This capability applies both to implicit transactions, such as macros and multistatement requests, and to Teradata-style transactions explicitly defined by BEGIN/END TRANSACTION statements.

Statements in a multistatement request are broken down by the Parser into one or more steps that direct the execution performed by the AMPs. It is these steps, not the actual statements, that are executed in parallel.

A handshaking protocol between the PE and the AMP allows the AMP to determine when the PE can dispatch the next parallel step.

Up to twenty parallel steps can be processed per request if channels are not required, such as a request with an equality constraint based on a primary index value. Up to ten channels can be used for parallel processing when a request is not constrained to a primary index value.

For example, if an INSERT step and a DELETE step are allowed to run in parallel, the AMP informs the PE that the DELETE step has progressed to the point where the INSERT step will not impact it adversely. This handshaking protocol also reduces the chance of a deadlock.

"Parallel Steps" on page 133 illustrates the following process:

1   The statements in a multistatement request are broken down into a series of steps.

2   The Optimizer determines which steps in the series can be executed in parallel.

3   The steps are processed.

Each step undergoes some preliminary processing before it is executed, such as placing locks on the objects involved. These preliminary processes are not performed in parallel with the steps.

## Parallel Steps
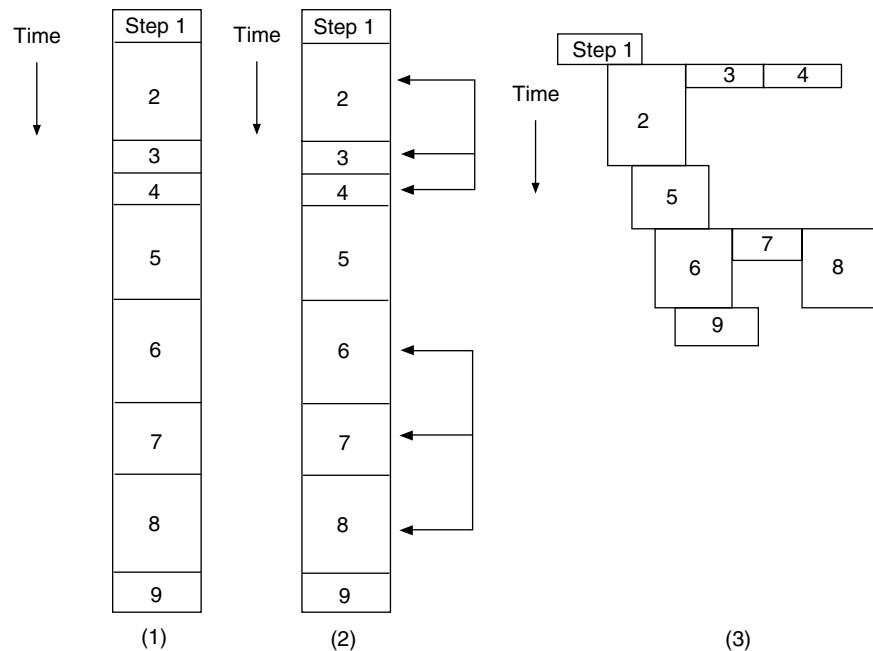


FF02A001

## Related Topics

| For more information on … | See … |
|---|---|
| multistatement requests<br>multistatment request processing | • *SQL Data Manipulation Language.*<br>• *SQL Request and Transaction Processing.* |

# Iterated Requests

## Usage

Iterated requests do not directly impact the syntax of SQL statements. They provide an efficient way to execute the same single-statement DML operation on multiple data records, like the way that ODBC applications execute parameterized statements for arrays of parameter values, for example.

Several Teradata Database client tools and interfaces provide facilities to pack multiple data records in a single buffer with a single DML statement.

For example, suppose you use BTEQ to import rows of data into table *ptable* using the following INSERT statement and USING modifier:

```
USING (pid INTEGER, pname CHAR(12))
INSERT INTO ptable VALUES(:pid, :pname);
```

To repeat the request as many times as necessary to read up to 200 data records and pack a maximum of 100 data records with each request, precede the INSERT statement with the following BTEQ command:

```
.REPEAT RECS 200 PACK 100
```

Note:  The PACK option is ignored if the database being used does not support iterated requests or if the request that follows the REPEAT command is not a DML statement supported by iterated requests. For details, see .

The following tools and interfaces provide facilities that you can use to execute iterated requests.

| Tool/Interface | Facility |
| --- | --- |
| CLIv2 for workstation-attached systems | *using_data_count* field in the DBCAREA data area |
| CLIv2 for mainframe systems | *using-data-count* field in the DBCAREA data area |
| ODBC | Parameter arrays |
| JDBC type 4 driver | Batch operations |
| Microsoft OLE DB Provider for ODBC and Teradata ODBC Driver | Parameter sets |
| BTEQ | • .REPEAT command<br>• .SET PACK command |

## Rules

- The iterated request must consist of a single DML statement from the following list:
  - ABORT
  - DELETE (excluding the positioned form of DELETE)
  - EXECUTE *macro_name*

    The fully-expanded macro must be equivalent to a single DML statement that is qualified to be in an iterated request.
  - INSERT
  - MERGE
  - ROLLBACK
  - SELECT
  - UPDATE (including atomic UPSERT, but excluding the positioned form of UPDATE)
- The DML statement must reference user-supplied input data, either as named fields in a USING modifier or as '?' parameter markers in a parameterized request.

  Note:  Iterated requests do not support the USING modifier with the TOP *n* operator.
- All the data records in a given request must use the same record layout. This restriction applies by necessity to requests where the record layout is given by a single USING modifier in the request text itself; but the restriction also applies to parameterized

requests, where the request text has no USING modifier and does not fully specify the input record.

- The server processes the iterated request as if it were a single multistatement request, with each iteration and its response associated with a corresponding statement number.

### Statement Independence for Simple INSERTs

When an iterated request includes only simple INSERT statements, a failure of one or more INSERTs does not cause the entire request to be rolled back. In these cases, errors are reported for the INSERT statements that failed, so those statements can be resubmitted. INSERT statements that completed successfully are not rolled back.

This behavior is limited to requests submitted directly to the database using an SQL INSERT multistatement request or submitted using a JDBC application request.

### Related Topics

| For more information on … | See … |
|---|---|
| iterated request processing | *SQL Request and Transaction Processing.* |
| which DML statements can be specified in an iterated request | *SQL Data Manipulation Language.* |
| CLIv2 | *Teradata Call-Level Interface Version 2 Reference for Workstation-Attached Systems* |
| ODBC parameter arrays | *ODBC Driver for Teradata User Guide.* |
| JDBC driver batch operations | *Teradata JDBC Driver User Guide.* |
| Microsoft OLE DB Provider for ODBC and the Teradata ODBC Driver | *ODBC Driver for Teradata User Guide.* |
| BTEQ PACK command | *Basic Teradata Query Reference.* |

# Aborting SQL Requests

If an error is found in a request, that request is aborted. Normally, the entire transaction is not aborted. However, some failures will abort the entire transaction.

A single statement or multistatement request that does not include the BEGIN TRANSACTION and END TRANSACTION statements is treated as an implicit transaction. If an error is found in any statement in this type of request, then the entire transaction is aborted.

### Abort Processing

Following are the steps for abort processing:

1   Back out any changes made to the database as a result of any preceding statements in the transaction.

2   Delete any associated spooled output.

3   Release any associated locks.

4   Bypass any remaining statements in the transaction.

## Completed Requests

A request is considered to have completed when either an End of Text character or the request terminator is encountered. The request terminator is a SEMICOLON character. It is the last nonpad character on an input line.

A request terminator is optional except when the request is embedded in an SQL macro or trigger or when it is entered through BTEQ.

# Dynamic and Static SQL

Dynamic SQL is a method of invoking an SQL statement by compiling and performing it at runtime from within an embedded SQL application program or a stored procedure. The specification of data to be manipulated by the statement is also determined at runtime. Static SQL is, by default, any method of invoking an SQL statement that is not dynamic.

## ANSI Compliance

Dynamic SQL is ANSI/ISO SQL:2011-compliant. The ANSI/ISO SQL standard does not define the expression static SQL, but relational database management commonly uses it to contrast with the ANSI-defined expression dynamic SQL.

## Ad Hoc and Hard-Coded Invocation of SQL Statements

Perhaps the best way to think of dynamic SQL is to contrast it with ad hoc SQL statements created and executed from a terminal and with preprogrammed SQL statements created by an application programmer and executed by an application program.

In the case of the ad hoc query, everything legal is available to the requester: choice of SQL statements and clauses, variables and their names, databases, tables, and columns to manipulate, and literals.

In the case of the application programmer, the choices are made in advance and hard-coded into the source code of the application. Once the program is compiled, nothing can be changed short of editing and recompiling the application.

## Dynamic Invocation of SQL Statements

Dynamic SQL offers a compromise between the extremes of ad hoc and hard-coded queries. By choosing to code dynamic SQL statements in the application, the programmer has the

flexibility to allow an end user to select not only the variables to be manipulated at run time, but also the SQL statement to be executed.

As you might expect, the flexibility that dynamic SQL offers a user is offset by more work and increased attention to detail on the part of the application programmer, who needs to set up additional dynamic SQL statements and manipulate information in the SQLDA to ensure a correct result.

This is done by first preparing, or compiling, an SQL text string containing placeholder tokens at run time and then executing the prepared statement, allowing the application to prompt the user for values to be substituted for the placeholders.

## SQL Statements to Set Up and Invoke Dynamic SQL

The embedded SQL statements for preparing and executing an SQL statement dynamically are:

- PREPARE
- EXECUTE
- EXECUTE IMMEDIATE

EXECUTE IMMEDIATE is a special form that combines PREPARE and EXECUTE into one statement. EXECUTE IMMEDIATE can only be used in the case where there are no input host variables.

For details, see *SQL Stored Procedures and Embedded SQL*.

## Related Topics

| For more information on … | See … |
|---|---|
| examples of dynamic SQL code in C, COBOL, and PL/I | *Teradata Preprocessor2 for Embedded SQL Programmer Guide.* |
| embedded SQL statements | *SQL Stored Procedures and Embedded SQL.* |

# Dynamic SQL in Stored Procedures

Stored procedures support of dynamic SQL statements is different from embedded SQL support.

Use the following statement to set up and invoke dynamic SQL in a stored procedure:

```
CALL DBC.SysExecSQL(string_expression)
```

where *string_expression* is any valid string expression that builds an SQL statement.

The string expression consists of string literals, status variables, local variables, input (IN and INOUT) parameters, and for-loop aliases. Dynamic SQL statements are not validated at compile time.

The resulting SQL statement cannot have status variables, local variables, parameters, for-loop aliases, or a USING or EXPLAIN modifier.

## Example

The following example uses dynamic SQL within stored procedure source text:

```
CREATE PROCEDURE new_sales_table( my_table VARCHAR(30),
                                  my_database VARCHAR(30))
BEGIN
  DECLARE sales_columns VARCHAR(128)
    DEFAULT '(item INTEGER, price DECIMAL(8,2), sold INTEGER)';
  CALL DBC.SysExecSQL('CREATE TABLE ' || my_database ||
                 '.' || my_table || sales_columns);
END;
```

A stored procedure can make any number of calls to SysExecSQL. The request text in the string expression can specify a multistatement request, but the call to SysExecSQL must be delimited by BEGIN REQUEST and END REQUEST keywords.

Because the request text of dynamic SQL statements can vary from execution to execution, dynamic SQL provides more usability and conciseness to the stored procedure definition.

## Restrictions

Whether the creator, owner, or invoker of the stored procedure must have appropriate privileges on the objects that the stored procedure accesses depends on whether the CREATE PROCEDURE statement includes the SQL SECURITY clause and which option the SQL SECURITY clause specifies.

The following SQL statements cannot be specified as dynamic SQL in stored procedures:

- ALTER PROCEDURE
- CALL
- CREATE PROCEDURE
- DATABASE
- EXPLAIN modifier
- HELP
- OPEN
- PREPARE
- REPLACE PROCEDURE
- SELECT

- SET ROLE
- SET SESSION ACCOUNT
- SET SESSION COLLATION
- SET SESSION DATEFORM
- SET TIME ZONE
- SHOW
- Cursor statements, including:
  - CLOSE
  - FETCH
  - OPEN

## Related Topics

For rules and usage examples of dynamic SQL statements in stored procedures, see *SQL Stored Procedures and Embedded SQL.*

# Using SELECT With Dynamic SQL

Unlike other executable SQL statements, SELECT returns information beyond statement responses and return codes to the requester.

## DESCRIBE Statement

Because the requesting application needs to know how much (if any) data will be returned by a dynamically prepared SELECT, you must use an additional SQL statement, DESCRIBE, to make the application aware of the demographics of the data to be returned by the SELECT statement (see "DESCRIBE" in *SQL Stored Procedures and Embedded SQL*).

DESCRIBE writes this information to the SQLDA declared for the SELECT statement as follows.

| THIS information … | IS written to this field of SQLDA … |
|---|---|
| number of values to be returned | SQLN |
| column name or label of $n^{th}$ value | SQLVAR |
| column data type of $n^{th}$ value | ($n^{th}$ row in the SQLVAR($n$) array) |
| column length of $n^{th}$ value | |

## General Procedure

An application must use the following general procedure to set up, execute, and retrieve the results of a SELECT statement invoked as dynamic SQL.

1  Declare a dynamic cursor for the SELECT in the form:

```
DECLARE cursor_name CURSOR FOR sql_statement_name
```

2  Declare the SQLDA, preferably using an INCLUDE SQLDA statement.

3  Build and PREPARE the SELECT statement.

4  Issue a DESCRIBE statement in the form:

```
DESCRIBE sql_statement_name INTO SQLDA
```

DESCRIBE performs the following actions:

a  Interrogate the database for the demographics of the expected results.

b  Write the addresses of the target variables to receive those results to the SQLDA.

   This step is bypassed if any of the following occurs:

   • The request does not return any data.

   • An INTO clause was present in the PREPARE statement.

   • The statement returns known columns and the INTO clause is used on the corresponding FETCH statement.

   • The application code defines the SQLDA.

5  Allocate storage for target variables to receive the returned data based on the demographics reported by DESCRIBE.

6  Retrieve the result rows using the following SQL cursor control statements:

- OPEN *cursor_name*

- FETCH *cursor_name* USING DESCRIPTOR SQLDA

- CLOSE *cursor_name*

In this step, results tables are examined one row at a time using the selection cursor. This is because client programming languages do not support data in terms of sets, but only as individual records.

# Event Processing Using Queue Tables

Teradata Database provides queue tables that you can use for event processing. Queue tables are base tables with first-in-first-out (FIFO) queue properties.

When you create a queue table, you define a timestamp column. You can query the queue table to retrieve data from the row with the oldest timestamp.

## Usage

An application can perform peek, FIFO push, and FIFO pop operations on queue tables.

| To perform a … | Use the … |
|---|---|
| FIFO push | INSERT statement. |
| FIFO pop | SELECT AND CONSUME statement. |
| peek | SELECT statement. |

Here is an example of how an application can process events using queue tables:

- Define a trigger on a base table to insert a row into the queue table when the trigger fires.

- From the application, submit a SELECT AND CONSUME statement that waits for data in the queue table.

- When data arrives in the queue table, the waiting SELECT AND CONSUME statement returns a result to the application, which processes the event. Additionally, the row is deleted from the queue table.

## Related Topics

| For more information on … | See … |
|---|---|
| creating queue tables | the CREATE/REPLACE TABLE statement in *SQL Data Definition Language*. |

| For more information on … | See … |
|---|---|
| SELECT AND CONSUME | *SQL Data Manipulation Language.* |

# Manipulating Nulls

Nulls are neither values nor do they signify values. They represent the absence of value. A null is a place holder indicating that no value is present.

You cannot solve for the value of a null because, by definition, it has no value. For example, the expression NULL = NULL has no meaning and therefore can never be true. A query that specifies the predicate WHERE NULL = NULL is not valid because it can never be true. The meaning of the comparison it specifies is not only unknown, but unknowable.

These properties make the use and interpretation of nulls in SQL problematic. The following sections outline the behavior of nulls for various SQL operations to help you to understand how to use them in data manipulation statements and to interpret the results those statements affect.

## NULL Literals

See "NULL Keyword as a Literal" on page 107 for information on how to use the NULL keyword as a literal.

## Nulls and DateTime and Interval Data

A DateTime or Interval value is either atomically null or it is not null. For example, you cannot have an interval of YEAR TO MONTH in which YEAR is null and MONTH is not.

## Rules for the Result of Expressions That Contain Nulls

- When any component of a value expression is null, then the result is null.
- The result of a conditional expression that has a null component is unknown.
- If an operand of any arithmetic operator (such as + or -) or function (such as ABS or SQRT) is null, then the result of the operation or function is null with the exception of ZEROIFNULL. If the argument to ZEROIFNULL is NULL, then the result is 0.
- COALESCE, a special shorthand variant of the CASE expression, returns NULL if all its arguments evaluate to null. Otherwise, COALESCE returns the value of the first argument that has a value that is not null.

For more rules on the result of expressions containing nulls, see the sections that follow and *SQL Functions, Operators, Expressions, and Predicates.*

## Nulls and Comparison Operators

If either operand of a comparison operator is null, then the result is unknown. If either operand is the keyword NULL, an error is returned that recommends using IS NULL or IS NOT NULL instead. The following examples indicate this behavior.

```
5 = NULL
5 <> NULL
NULL = NULL
NULL <> NULL
5 = NULL + 5
```

If the argument of the NOT operator is unknown, the result is also unknown. This translates to FALSE as a final boolean result.

Instead of using comparison operators, use the IS NULL operator to search for fields that contain nulls and the IS NOT NULL operator to search for fields that do not contain nulls. For details, see "Searching for Nulls" on page 142 and "Excluding Nulls" on page 142.

Using IS NULL is different from using the comparison operator =. When you use an operator like =, you specify a comparison between values or value expressions, whereas when you use the IS NULL operator, you specify an existence condition.

## Rules for Nulls and CASE Expressions

- CASE and its related expressions COALESCE and NULLIF can return a null.
- NULL and null expressions are valid as the CASE test expression in a valued CASE expression.
- When testing for NULL, it is best to use a searched CASE expression using the IS NULL or IS NOT NULL operators in the WHEN clause.
- NULL and null expressions are valid as THEN clause conditions.

For details on the rules for nulls in CASE, NULLIF, and COALESCE expressions, see *SQL Functions, Operators, Expressions, and Predicates*.

## Excluding Nulls

To exclude nulls from the results of a query, use the operator IS NOT NULL.

For example, to search for the names of all employees with a value other than null in the jobtitle column, enter the statement.

```
SELECT name
FROM employee
WHERE jobtitle IS NOT NULL ;
```

## Searching for Nulls

To search for columns that contain nulls, use the operator IS NULL.

The IS NULL operator tests row data for the presence of nulls.

For example, to search for the names of all employees who have a null in the deptno column, you could enter the statement:

```
SELECT name
FROM employee
WHERE deptno IS NULL ;
```

This query produces the names of all employees with a null in the deptno field.

## Searching for Nulls and Values that Are Not Null Together

To search for nulls and values that are not null in the same statement, the search condition for nulls must be separate from any other search conditions.

For example, to select the names of all employees with the job title of Vice Pres, Manager, or null, enter the following SELECT statement.

```
SELECT name, jobtitle
FROM employee
WHERE jobtitle IN ('Manager', 'Vice Pres') OR jobtitle IS NULL ;
```

Including NULL in the IN list has no effect because NULL never equals NULL or any value.

## Null Sorts as the Lowest or Highest Value in a Collation

When you use an ORDER BY clause to sort records, Teradata Database sorts null as the lowest or highest value.

If any row has a null in the column being grouped, then all rows having a null are placed into one group.

## NULL and Unique Indexes

For unique indexes, Teradata Database treats nulls as if they are equal rather than unknown (and therefore false).

For single-column unique indexes, only one row may have null for the index value; otherwise a uniqueness violation error occurs.

For multicolumn unique indexes, no two rows can have nulls in the same columns of the index and also have values that are not null and that are equal in the other columns of the index.

For example, consider a two-column index. Rows can occur with the following index values:

| Value of First Column in Index | Value of Second Column in Index |
| --- | --- |
| 1 | null |
| null | 1 |
| null | null |

An attempt to insert a row that matches any of these rows will result in a uniqueness violation.

## Replacing Nulls With Values on Return to Client in Record Mode

When Teradata Database returns information to a client system in record mode, nulls must be replaced with some value for the underlying column because client system languages do not recognize nulls.

The following table shows the values returned for various column data types.

| Data Type | Substitute Value Returned for Null |
|---|---|
| CHARACTER($n$)<br><br>DATE<br><br>TIME<br><br>TIMESTAMP<br><br>INTERVAL | Pad character (or $n$ pad characters for CHARACTER($n$), where $n > 1$) |
| PERIOD(DATE) | 8 binary zero bytes |
| PERIOD(TIME [($n$)]) | 12 binary zero bytes |
| PERIOD(TIME [($n$)] WITH TIME ZONE) | 16 binary zero bytes |
| PERIOD(TIMESTAMP [($n$)] [WITH TIME ZONE]) | 0-length byte string |
| BYTE[($n$)] | Binary zero byte if $n$ omitted else $n$ binary zero bytes |
| VARBYTE($n$) | 0-length byte string |
| VARCHARACTER($n$) | 0-length character string |
| BIGINT<br><br>INTEGER<br><br>SMALLINT<br><br>BYTEINT<br><br>FLOAT<br><br>DECIMAL<br><br>REAL<br><br>DOUBLE PRECISION<br><br>NUMERIC | 0 |

The substitute values returned for nulls are not, by themselves, distinguishable from valid values that are not null. Data from CLI is normally accessed in IndicData mode, in which additional identifying information that flags nulls is returned to the client.

BTEQ uses the identifying information, for example, to determine whether the values it receives are values or just aliases for nulls so it can properly report the results. BTEQ displays nulls as >?, which are not by themselves distinguishable from a CHAR or VARCHAR value of '?'.

## Nulls and Aggregate Functions

With the important exception of COUNT(*), aggregate functions ignore nulls in their arguments. This treatment of nulls is very different from the way arithmetic operators and functions treat them.

This behavior can result in apparent nontransitive anomalies. For example, if there are nulls in either column A or column B (or both), then the following expression is virtually always true.

```
SUM(A) + (SUM B) <> SUM (A+B)
```

In other words, for the case of SUM, the result is never a simple iterated addition if there are nulls in the data being summed.

The only exception to this is the case in which the values for columns A and B are both null in the same rows, because in those cases the entire row is disregarded in the aggregation. This is a trivial case that does not violate the general rule.

The same is true, the necessary changes being made, for all the aggregate functions except COUNT(*).

If this property of nulls presents a problem, you can always do either of the following workarounds, each of which produces the desired result of the aggregate computation SUM(A) + SUM(B) = SUM(A+B).

- Always define NUMERIC columns as `NOT NULL DEFAULT 0`.
- Use the ZEROIFNULL function within the aggregate function to convert any nulls to zeros for the computation, for example

  ```
  SUM(ZEROIFNULL(x) + ZEROIFNULL(y))
  ```

  which produces the same result as this:

  ```
  SUM(ZEROIFNULL(x) + ZEROIFNULL(y)).
  ```

COUNT(*) includes nulls in its result. For details, see *SQL Functions, Operators, Expressions, and Predicates.*

## RANGE_N and CASE_N Functions

Nulls have special considerations in the RANGE_N and CASE_N functions. For details, see *SQL Functions, Operators, Expressions, and Predicates.*

# Session Parameters

The following session parameters can be controlled with keywords or predefined system variables.

| Parameter | Valid Keywords or System Variables |
|---|---|
| SQL Flagger | ON |
| | OFF |

| Parameter | Valid Keywords or System Variables |
|---|---|
| Transaction Mode | ANSI (COMMIT) |
| | Teradata (BTET) |
| Session Collation | ASCII |
| | EBCDIC |
| | MULTINATIONAL |
| | HOST |
| | CHARSET_COLL |
| | JIS_COLL |
| Account and Priority | Account and reprioritization. Within the account identifier, you can specify a performance group or use one of the following predefined performance groups:<br><br>• $R<br>• $H<br>• $M<br>• $L |
| Date Form | ANSIDATE |
| | INTEGERDATE |
| Character Set | Indicates the character set being used by the client.<br><br>You can view site-installed client character sets from DBC.CharSetsV or DBC.CharTranslationsV.<br><br>The following client character sets are permanently enabled:<br><br>• ASCII<br>• EBCDIC<br>• UTF-8<br>• UTF-16<br><br>For more information on character sets, see *International Character Set Support*. |

## SQL Flagger

When enabled, the SQL Flagger assists SQL programmers by notifying them of the use of non-ANSI and non-entry level ANSI/ISO SQL syntax.

Enabling the SQL Flagger can be done regardless of whether you are in ANSI or Teradata session mode.

To set the SQL Flagger on or off for BTEQ, use the .SET SESSION command.

| To set this level of flagging … | Set the flag variable to this value … |
|---|---|
| None | SQLFLAG NONE |

| To set this level of flagging … | Set the flag variable to this value … |
|---|---|
| Entry level | SQLFLAG ENTRY |
| Intermediate level | SQLFLAG INTERMEDIATE |

For more detail on using the SQL Flagger, see .

To set the SQL Flagger on or off for embedded SQL, use the SQLCHECK or -sc and SQLFLAGGER or -sf options when you invoke the preprocessor.

If you are using SQL in other application programs, see the reference manual for that application for instructions on enabling the SQL Flagger.

## Transaction Mode

You can run transactions in either Teradata or ANSI session modes and these modes can be set or changed.

To set the transaction mode, use the .SET SESSION command in BTEQ.

| To run transactions in this mode … | Set the variable to this value … |
|---|---|
| Teradata | TRANSACTION BTET |
| ANSI | TRANSACTION ANSI |

For more detail on transaction semantics, see "Transaction Processing" in *SQL Request and Transaction Processing*.

If you are using SQL in other application programs, see the reference manual for that application for instructions on setting or changing the transaction mode.

## Session Collation

Collation of character data is an important and complex option. Teradata Database provides several named collations. The MULTINATIONAL and CHARSET_COLL collations allow the system administrator to provide collation sequences tailored to the needs of the site.

The collation for the session is determined at logon from the defined default collation for the user. You can change your collation any number of times during the session using the SET SESSION COLLATION statement, but you cannot change your default logon in this way.

Your default collation is assigned via the COLLATION option of the CREATE USER or MODIFY USER statement. This has no effect on any current session, only new logons.

Each named collation can be CASESPECIFIC or NOT CASESPECIFIC. NOT CASESPECIFIC collates lowercase data as if it were converted to uppercase before the named collation is applied.

| Collation Name | Description |
|---|---|
| ASCII | Character data is collated in the order it would appear if converted for an ASCII session, and a binary sort performed. |
| EBCDIC | Character data is collated in the order it would appear if converted for an EBCDIC session, and a binary sort performed. |
| MULTINATIONAL | The default MULTINATIONAL collation is a two-level collation based on the Unicode collation standard.<br><br>Your system administrator can redefine this collation to any two-level collation of characters in the LATIN repertoire.<br><br>For backward compatibility, the following are true:<br><br>• MULTINATIONAL collation of KANJI1 data is single level.<br>• The system administrator can redefine single byte character collation.<br><br>This definition is not compatible with MULTINATIONAL collation of non-KANJI1 data. CHARSET_COLL collation is usually a better solution for KANJI1 data.<br><br>See "ORDER BY Clause" in *SQL Data Manipulation Language*. For information on setting up the MULTINATIONAL collation sequence, see "Collation Sequences" in *International Character Set Support*. |
| HOST | The default. HOST collation defaults are:<br><br>• EBCDIC collation for mainframe systems.<br>• ASCII collation for all others. |
| CHARSET_COLL | Character data is collated in the order it would appear if converted to the current client character set and then sorted in binary order.<br><br>CHARSET_COLL collation is a system administrator-defined collation. |
| JIS_COLL | Character data is collated based on the Japanese Industrial Standards (JIS).<br><br>JIS characters collate in the following order:<br><br>1 JIS X 0201-defined characters in standard order<br>2 JIS X 0208-defined characters in standard order<br>3 JIS X 0212-defined characters in standard order<br>4 KanjiEBCDIC-defined characters not defined in JIS X 0201, JIS X 0208, or JIS X 0212 in standard order<br>5 All remaining characters in Unicode standard order |

For details, see "SET SESSION COLLATION" in *SQL Data Definition Language*.

## Account and Priority

You can dynamically downgrade or upgrade the performance group priority for your account.

Priorities can be downgraded or upgraded at either the session or the request level. For more information, see "SET SESSION ACCOUNT" in *SQL Data Definition Language*.

Note: Changing the performance group for your account changes the account name for accounting purposes because a performance group is part of an account name.

## Date Form

You can change the format in which DATE data is imported or exported in your current session.

DATE data can be set to be treated either using the ANSI date format (DATEFORM=ANSIDATE) or using the Teradata date format (DATEFORM=INTEGERDATE).

For details, see "SET SESSION DATEFORM" in *SQL Data Definition Language*.

## Setting the Client Character Set

To set the client character set, use one of the following:

- From BTEQ, use the BTEQ [.] SET SESSION CHARSET *'name'* command.
- In a CLIv2 application, call CHARSET *name*.
- In the URL for selecting a Teradata JDBC driver connection to a Teradata Database, use the CHARSET=*name* database connection parameter.

where the '*name*' or *name* value is ASCII, EBCDIC, UTF-8, UTF-16, or a name assigned to the translation codes that define an available character set.

If not explicitly requested, the session default is the character set associated with the logon client. This is either the standard client default, or the character set assigned to the client by the database administrator.

## HELP SESSION

The HELP SESSION statement identifies attributes in effect for the current session, including:

- Transaction mode
- Character set
- Collation sequence
- Date form
- Queryband

For details, see "HELP SESSION" in *SQL Data Definition Language*.

# Session Management

Each session is logged on and off via calls to CLIv2 routines or through ODBC or JDBC, which offer a one-step logon-connect function.

Sessions are internally managed by dividing the session control functions into a series of single small steps that are executed in sequence to implement multithreaded tasking. This provides

concurrent processing of multiple logon and logoff events, which can be any combination of individual users, and one or more concurrent sessions established by one or more users and applications.

Once connected and active, a session can be viewed as a work stream consisting of a series of requests between the client and server.

## Session Pools

You can establish session pools, in which an application logs on to Teradata Database and establishes a pooled connection, so users can use application functions that access the database without the need to log on individually. This capability is particularly advantageous for transaction processing in which interaction with Teradata Database consists of many single, short transactions.

Teradata Database identifies a session with a session number, the username of the initiating (application) user, and the logical host identification number of the connection TDP. Also see Trusted Sessions.

## Session Reserve

On a mainframe client, use the ENABLE SESSION RESERVE command from Teradata Director Program to reserve session capacity in the event of a PE failure. To release reserved session capacity, use the DISABLE SESSION RESERVE command.

For more information, see *Teradata Director Program Reference*.

## Session Control

The major functions of session control are session logon and logoff.

Upon receiving a session request, the logon function verifies authorization and returns a yes or no response to the client.

The logoff function terminates any ongoing activity and deletes the session context.

## Trusted Sessions

Applications that use connection pooling can be configured to use trusted sessions, asserting individual end user identities and roles to manage privileges and audit access.

For details on trusted sessions, see *Security Administration*.

## Requests and Responses

Requests are sent to a server to initiate an action. Responses are sent by a server to reflect the results of that action. Both requests and responses are associated with an established session.

A request consists of the following components:

- One or more Teradata SQL statements
- Control information

- Optional USING data

If any operation specified by an initiating request fails, the request is backed out, along with any change that was made to the database. In this case, a failure response is returned to the application.

# Return Codes

SQL return codes provide information about the status of a completed executable SQL DML statement.

## Status Variables for Receiving SQL Return Codes

ANSI/ISO SQL defines two status variables for receiving return codes:

- SQLSTATE
- SQLCODE

SQLCODE is not ANSI/ISO SQL-compliant. The ANSI/ISO SQL-92 standard explicitly deprecates SQLCODE, and the ANSI/ISO SQL-99 standard does not define SQLCODE. The ANSI/ISO SQL committee recommends that new applications use SQLSTATE in place of SQLCODE.

Teradata Database defines a third status variable for receiving the number of rows affected by an SQL statement in a stored procedure:

- ACTIVITY_COUNT

Teradata SQL defines a non-ANSI/ISO SQL Communications Area (SQLCA) that also has a field named SQLCODE for receiving return codes.

| For information on … | See … |
| --- | --- |
| - SQLSTATE<br>- SQLCODE<br>- ACTIVITY_COUNT | "Result Code Variables" in *SQL Stored Procedures and Embedded SQL*. |
| SQLCA | "SQL Communications Area (SQLCA)" in *SQL Stored Procedures and Embedded SQL*. |

## Exception and Completion Conditions

ANSI/ISO SQL defines two categories of conditions that issue return codes:

- Exception conditions
- Completion conditions

## Exception Conditions

An exception condition indicates a statement failure.

A statement that raises an exception condition does nothing more than return that exception condition to the application.

There are as many exception condition return codes as there are specific exception conditions.

For more information about exception conditions, see "Failure Response" on page 156 and "Error Response (ANSI Session Mode Only)" on page 155.

For a complete list of exception condition codes, see *Messages*.

## Completion Conditions

A completion condition indicates statement success.

There are three categories of completion conditions:

* Successful completion
* Warnings
* No data found

A statement that raises a completion condition can take further action such as querying the database and returning results to the requesting application, updating the database, initiating an SQL transaction, and so on.

| Completion Condition | SQLSTATE Return Code | SQLCODE Return Code |
|---|---|---|
| Success | '00000' | 0 |
| Warning | '01901' | 901 |
| | '01800' to '01841' | 901 |
| | '01004' | 902 |
| No data found | '02000' | 100 |

### Related Topics

For more information, see:

* "Statement Responses" on page 153
* "Success Response" on page 154
* "Warning Response" on page 155

## Return Codes for Stored Procedures

The return code values are different in the case of SQL control statements in stored procedures.

The return codes for stored procedures appear in the following table.

| Completion Condition | SQLSTATE Return Code | SQLCODE Return Code |
|---|---|---|
| Successful completion | '00000' | 0 |
| Warning | SQLSTATE value corresponding to the warning code. | Teradata Database warning code. |
| No data found or any other Exception | SQLSTATE value corresponding to the error code. | Teradata Database error code. |

## How an Application Uses SQL Return Codes

An application program or stored procedure tests the status of a completed executable SQL statement to determine its status.

| Condition | Action |
|---|---|
| Successful completion | None. |
| Warning | The statement execution continues.<br><br>If a warning condition handler is defined in the application, the handler executes. |
| No data found or any other exception | Whatever appropriate action is required by the exception.<br><br>If an EXIT handler is defined for the exception, the statement execution terminates.<br><br>If a CONTINUE handler is defined, execution continues after the remedial action. |

# Statement Responses

## Response Types

Teradata Database responds to an SQL request with one of the following condition responses:

- Success response, with optional warning
- Failure response
- Error response (ANSI session mode only)

Depending on the type of statement, Teradata Database also responds with one or more rows of data.

## Multistatement Responses

A response to a request that contains more than one statement, such as a macro, is not returned to the client until all statements in the request are successfully executed.

## Returning a Response

The manner in which the response is returned depends on the interface that is being used.

For example, if an application is using a language preprocessor, then the activity count, warning code, error code, and fields from a selected row are returned directly to the program through its appropriately declared variables. If the application is a stored procedure, then the activity count is returned directly in the ACTIVITY_COUNT status variable.

If you are using BTEQ, then a success, error, or failure response is displayed automatically.

## Response Condition Codes

SQL statements also return condition codes that are useful for handling errors and warnings in embedded SQL and stored procedure applications.

For information about SQL response condition codes, see the following in *SQL Stored Procedures and Embedded SQL*:

- SQLSTATE
- SQLCODE
- ACTIVITY_COUNT

# Success Response

A success response contains an activity count that indicates the total number of rows involved in the result.

For example, the activity count for a SELECT statement is the total number of rows selected for the response. For a SELECT, CALL (when the stored procedure or external stored procedure creates result sets), COMMENT, or ECHO statement, the activity count is followed by the data that completes the response.

An activity count is meaningful for statements that return a result set, for example:

- SELECT
- INSERT
- UPDATE
- DELETE
- HELP
- MERGE
- SHOW
- EXPLAIN
- CREATE PROCEDURE
- REPLACE PROCEDURE
- CALL (when the stored procedure or external stored procedure creates result sets)

For other SQL statements, activity count is meaningless.

## Example

The following interactive SELECT statement returns the successful response message.

```
SELECT AVG(f1)
FROM Inventory;

*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.
Average(f1)
-----------
         14
```

# Warning Response

A success or OK response with a warning indicates either that an anomaly has occurred or informs the user about the anomaly and indicates how it can be important to the interpretation of the results returned.

## Example

Assume the current session is running in ANSI session mode.

If nulls are included in the data for column f1, then the following interactive query returns the successful response message with a warning about the nulls.

```
SELECT AVG(f1) FROM Inventory;

 *** Query completed. One row found. One column returned.
 *** Warning: 2892 Null value eliminated in set function.
 *** Total elapsed time was 1 second.

Average(f1)
-----------
         14
```

This warning response is not generated if the session is running in Teradata session mode.

# Error Response (ANSI Session Mode Only)

An error response occurs when a query anomaly is severe enough to prevent the correct processing of the request.

In ANSI session mode, an error for a request causes the request to rollback, and not the entire transaction.

## Example 1

The following command returns the error message immediately following.

```
.SET SESSION TRANS ANSI;
```

```
   *** Error: You must not be logged on .logoff to change the SQLFLAG
   or TRANSACTION settings.
```

## Example 2

Assume that the session is running in ANSI session mode, and the following table is defined:

```
CREATE MULTISET TABLE inv, FALLBACK,
   NO BEFORE JOURNAL,
   NO AFTER JOURNAL
   (
   item INTEGER CHECK ((item >=10) AND (item <= 20) ))
PRIMARY INDEX (item);
```

You insert a value of 12 into the item column of the inv table.

This is valid because the defined integer check specifies that any integer between 10 and 20 (inclusive) is valid.

```
INSERT INTO inv (12);
```

The following results message returns.

```
   *** Insert completed. One row added....
```

You insert a value of 9 into the item column of the inv table.

This is not valid because the defined integer check specifies that any integer with a value less than 10 is not valid.

```
INSERT INTO inv (9);
```

The following error response returns:

```
   *** Error 5317 Check constraint violation: Check error in field
   inv.item.
```

You commit the current transaction:

```
COMMIT;
```

The following results message returns:

```
   *** COMMIT done. ...
```

You select all rows from the inv table:

```
SELECT * FROM inv;
```

The following results message returns:

```
  *** Query completed. One row found. One column returned.
    item
-------
     12
```

# Failure Response

A failure response is a severe error. The response includes a statement number, an error code, and an associated text string describing the cause of the failure.

## Teradata Session Mode

In Teradata session mode, a failure causes the system to roll back the entire transaction.

If one statement in a macro fails, a single failure response is returned to the client, and the results of any previous statements in the transaction are backed out.

## ANSI Session Mode

In ANSI session mode, a failure causes the system to roll back the entire transaction, for example, when the current request:

*   Results in a deadlock
*   Performs a DDL statement that aborts
*   Executes an explicit ROLLBACK or ABORT statement

## Example 1

The following SELECT statement

```
SELECT * FROM Inventory:;
```

in BTEQ, returns the failure response message:

```
*** Failure 3706 Syntax error: expected something between the word
'Inventory' and ':'.
              Statement# 1, Info =20
*** Total elapsed time was 1 second.
```

## Example 2

Assume that the session is running in ANSI session mode, and the following table is defined:

```
CREATE MULTISET TABLE inv, FALLBACK,
   NO BEFORE JOURNAL,
   NO AFTER JOURNAL
   (
   item INTEGER CHECK ((item >=10) AND (item <= 20) ))
PRIMARY INDEX (item);
```

You insert a value of 12 into the item column of the inv table.

This is valid because the defined integer check specifies that any integer between 10 and 20 (inclusive) is valid.

```
INSERT INTO inv (12);
```

The following results message returns.

```
*** Insert completed. One row added....
```

You commit the current transaction:

```
COMMIT;
```

The following results message returns:

```
*** COMMIT done. ...
```

You insert a valid value of 15 into the item column of the inv table:

```
     INSERT INTO inv (15);
```

The following results message returns.

```
   *** Insert completed. One row added....
```

You can use the ABORT statement to cause the system to roll back the transaction:

```
   ABORT;
```

The following failure message returns:

```
   *** Failure 3514 User-generated transaction ABORT.
                Statement# 1, Info =0
```

You select all rows from the inv table:

```
   SELECT * FROM inv;
```

The following results message returns:

```
   *** Query completed. One row found. One column returned.

     item
   -------
       12
```

# CHAPTER 5 Query Processing

This chapter discusses query processing, including single AMP requests and all AMP requests, and table access methods available to the Optimizer.

## Queries and AMPs

An SQL query, which includes DELETE, INSERT, MERGE, and UPDATE as well as SELECT, can affect one AMP, several AMPs, or all AMPs in the configuration.

| IF a query … | THEN … |
|---|---|
| (involving a single table) uses a unique primary index (UPI) | the row hash can be used to identify a single AMP.<br><br>At most one row can be returned. |
| (involving a single table) uses a nonunique primary index (NUPI) | the row hash can be used to identify a single AMP.<br><br>Any number of rows can be returned. |
| uses a unique secondary index (USI) | one or two AMPs are affected (one AMP if the subtable and base table are on the same AMP).<br><br>At most one row can be returned. |
| uses a nonunique secondary index (NUSI) | if the table has a partitioned primary index (PPI) and the NUSI is the same column set as a NUPI, the query affects one AMP.<br><br>Otherwise, all AMPs take part in the operation and any number of rows can be returned. |

The SELECT statements in subsequent examples reference the following table data.

Employee

| Employee Number | Manager Employee Number | Dept. Number | Job Code | Last Name | First Name | Hire Date | Birth Date | Salary Amount |
|---|---|---|---|---|---|---|---|---|
| PK/UPI | FK | FK | FK | | | | | |
| 1006 | 1019 | 301 | 312101 | Stein | John | 961005 | 631015 | 2945000 |
| 1008 | 1019 | 301 | 312102 | Kanieski | Carol | 970201 | 680517 | 2925000 |
| 1005 | 0801 | 403 | 431100 | Ryan | Loretta | 1061015 | 650910 | 3120000 |
| 1004 | 1003 | 401 | 412101 | Johnson | Darlene | 1061015 | 760423 | 3630000 |
| 1007 | 1005 | 403 | 432101 | Villegas | Arnando | 1050102 | 770131 | 4970000 |
| 1003 | 0801 | 401 | 411100 | Trader | James | 960731 | 670619 | 3755000 |
| 1016 | 0801 | 302 | 321100 | Rogers | Nora | 980310 | 690904 | 5650000 |
| 1012 | 1005 | 403 | 432101 | Hopkins | Paulene | 970315 | 720218 | 3790000 |
| 1019 | 0801 | 301 | 311100 | Kubic | Ron | 980801 | 721211 | 5770000 |
| 1023 | 1017 | 501 | 512101 | Rabbit | Peter | 1040301 | 621029 | 2650000 |
| 1083 | 0801 | 619 | 414221 | Kimble | George | 1010312 | 810330 | 3620000 |
| 1017 | 0801 | 501 | 511100 | Runyon | Irene | 980501 | 611110 | 6600000 |
| 1001 | 1003 | 401 | 412101 | Hoover | William | 1010818 | 700114 | 2552500 |

The meanings of the abbreviations are as follows.

| Abbreviation | Meaning |
|---|---|
| PK | Primary Key |
| FK | Foreign Key |
| UPI | Unique Primary Index |

## Single AMP Request

Assume that a PE receives the following SELECT statement:

```
SELECT last_name
FROM Employee
WHERE employee_number = 1008;
```

Because a unique primary index value is used as the search condition (the column employee_number is the primary index for the Employee table), PE1 generates a single AMP step requesting the row for employee 1008. The AMP step, along with the PE identification, is put into a message, and sent via the BYNET to the relevant AMP (processor).

See for a flow diagram that illustrates this process.

## Flow Diagram of a Single AMP Request

Only one BYNET is shown to simplify the illustration.



1101A762
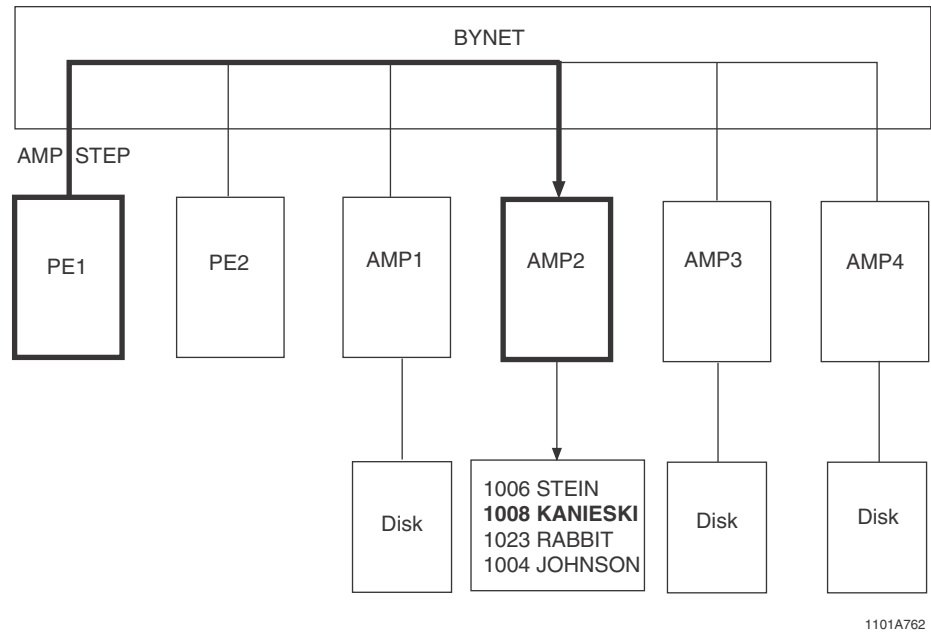
Assuming that AMP2 has the row, it accepts the message. As shown in the next diagram, AMP2 retrieves the row from disk, includes the row and the PE identification in a return message, and sends the message back to PE1 via the BYNET. PE1 accepts the message and returns the response row to the requesting application.

## Flow Diagram of a Single AMP Response to Requesting PE

The following diagram illustrates a single AMP request with partition elimination.



1101D003
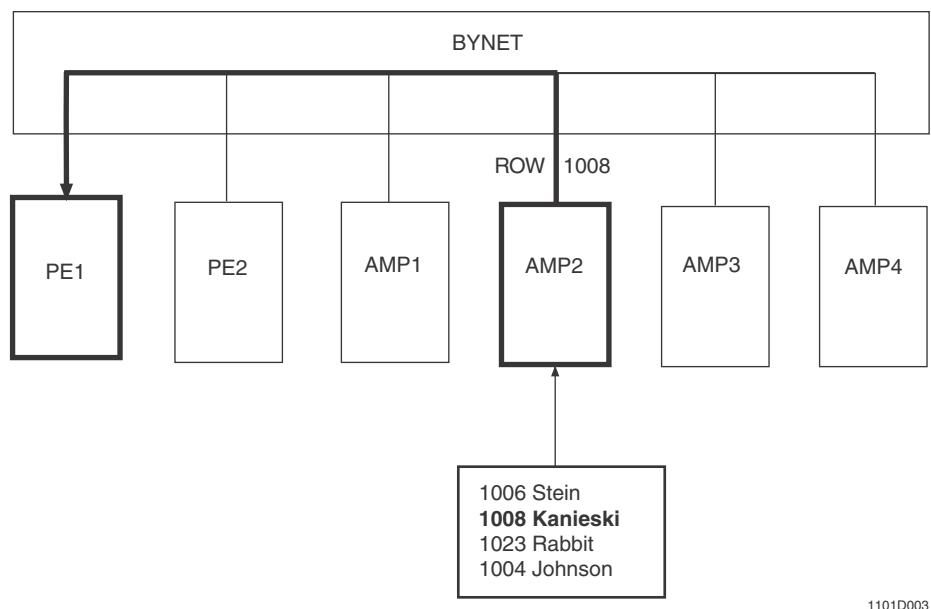
## All AMP Request

Assume PE1 receives a SELECT statement that specifies a range of primary index values as a search condition as shown in the following example:

```
SELECT last_name, employee_number
FROM employee
WHERE employee_number BETWEEEN 1001 AND 1010
ORDER BY last_name;
```

In this case, each value hashes differently, and all AMPs must search for the qualifying rows.

PE1 first parses the request and creates the following AMP steps:

• Retrieve rows between 1001 and 1010

• Sort ascending on last_name

• Merge the sorted rows to form the answer set

PE1 then builds a message for each AMP step and puts that message onto the BYNET. Typically, each AMP step is completed before the next one begins; note, however, that some queries can generate parallel steps.

When PE1 puts the message for the first AMP step on the BYNET, that message is broadcast to all processors as illustrated in the following diagram.
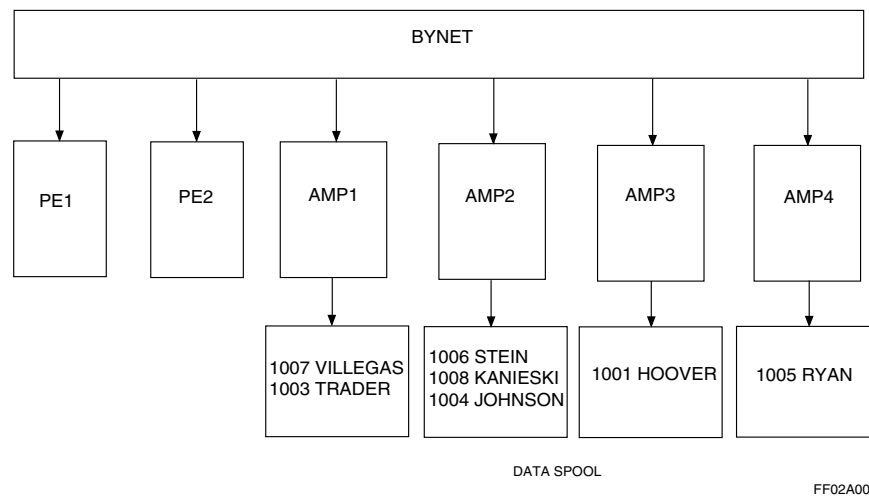


The process is:

1 All AMPs accept the message, but the PEs do not.

2 Each AMP checks for qualifying rows on its disk storage units.

3 If any qualifying rows are found, the data in the requested columns is converted to the client format and copied to a spool file.

4 Each AMP completes the step, whether rows were found or not, and puts a completion message on the BYNET.

The completion messages flow across the BYNET to PE1.

5  When all AMPs have returned a completion message, PE1 transmits a message containing AMP Step 2 to the BYNET.

Upon receipt of Step 2, the AMPs sort their individual answer sets into ascending sequence by *last_name* , as illustrated in the following diagram.

Note:  If partitioned on employee_number, the scan may be limited to a few partitions based on partition elimination.
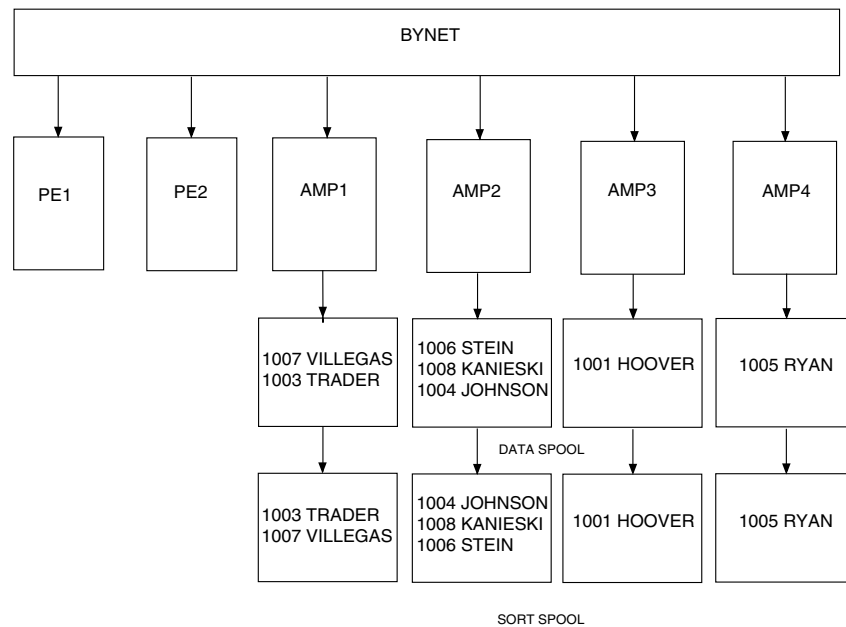


FF02A005

6  Each AMP sorts its answer set, then puts a completion message on the BYNET.

7  When PE1 has received all completion messages for Step 2, it sends a message containing AMP Step 3.

8  Upon receipt of Step 3, each AMP copies the first block from its sorted spool to the BYNET.

Because there can be multiple AMPs on a single node, each node might be required to handle sort spools from multiple AMPs (see the diagram at the top of the next page).
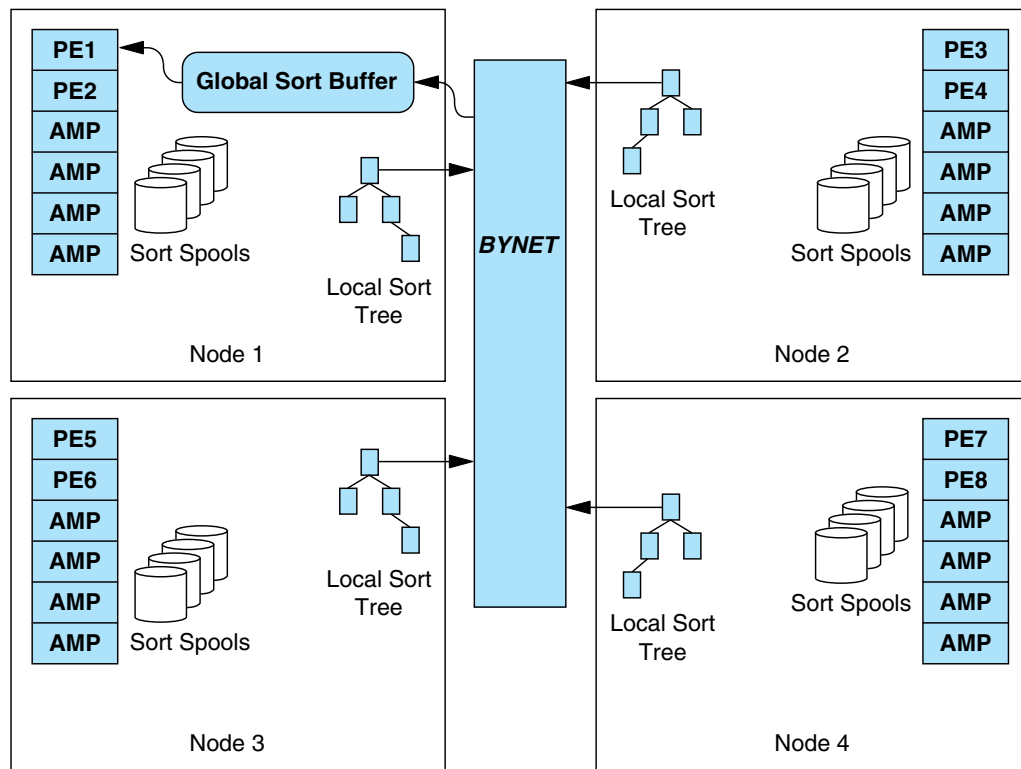
HD03A005

9   Nodes that contain multiple AMPs must first perform an intermediate sort of the spools generated by each of the local AMPs.

When the local sort is complete on each node, the lowest sorting row from each node is sent over the BYNET to PE1. From this point on, PE1 acts as the Merge coordinator among all the participating nodes.

10   The Merge continues with PE1 building a globally sorted buffer.

When this buffer fills, PE1 forwards it to the application and begins building subsequent buffers.

11   When a participant node has exhausted its sort spool, it sends a Done message to PE1.

This causes PE1 to prune this node from the set of Merge participants.

When there are no remaining Merge participants, PE1 sends the final buffer to the application along with an End Of File message.

## Partition Elimination

A PPI can increase query efficiency through partition elimination, where partitions can automatically be skipped because they cannot contain qualifying rows.

Teradata Database supports several types of partition elimination.

| Type | Description |
|---|---|
| Static | Based on constant conditions such as equality or inequality on the partitioning columns. |
| Dynamic | The partitions to eliminate cannot be determined until the query is executed and the data is scanned. |
| Delayed | Occurs with conditions comparing a partitioning column to a USING variable or built-in function such as CURRENT_DATE, where the Optimizer builds a somewhat generalized plan for the query but delays partition elimination until specific values of USING variables and built-in functions are known. |

The degree of partition elimination depends on the:

- Partitioning expressions for the primary index of the table
- Conditions in the query
- Ability of the Optimizer to detect partition elimination

It is not always required that all values of the partitioning columns be specified in a query to have partition elimination occur.

| IF a query … | THEN … |
|---|---|
| specifies values for all the primary index columns | the AMP where the rows reside can be determined and only a single AMP is accessed.<br><br>• If conditions are not specified on the partitioning columns, each partition can be probed to find the rows based on the hash value.<br><br>• If conditions are also specified on the partitioning columns, partition elimination may reduce the number of partitions to be probed on that AMP.<br><br>For an illustration, see "Single AMP Request With Partition Elimination" on page 165. |
| does not specify the values for all the primary index columns | an all-AMP full file scan is required for a table with an NPPI.<br><br>However, with a PPI, if conditions are specified on the partitioning columns, partition elimination may reduce an all-AMP full file scan to an all-AMP scan of only the non-eliminated partitions. |

## Single AMP Request With Partition Elimination

If a SELECT specifies values for all the primary index columns, the AMP where the rows reside can be determined and only a single AMP is accessed.

If conditions are also specified on the partitioning columns, partition elimination may reduce the number of partitions to be probed on that AMP.

Suppose the Employee table is defined with a single-level PPI where the partitioning column is dept_number.

Assume that a PE receives the following SELECT statement:

```
SELECT last_name
FROM Employee
WHERE employee_number = 1023
AND dept_number = 501;
```

The following flow diagram illustrates this process.
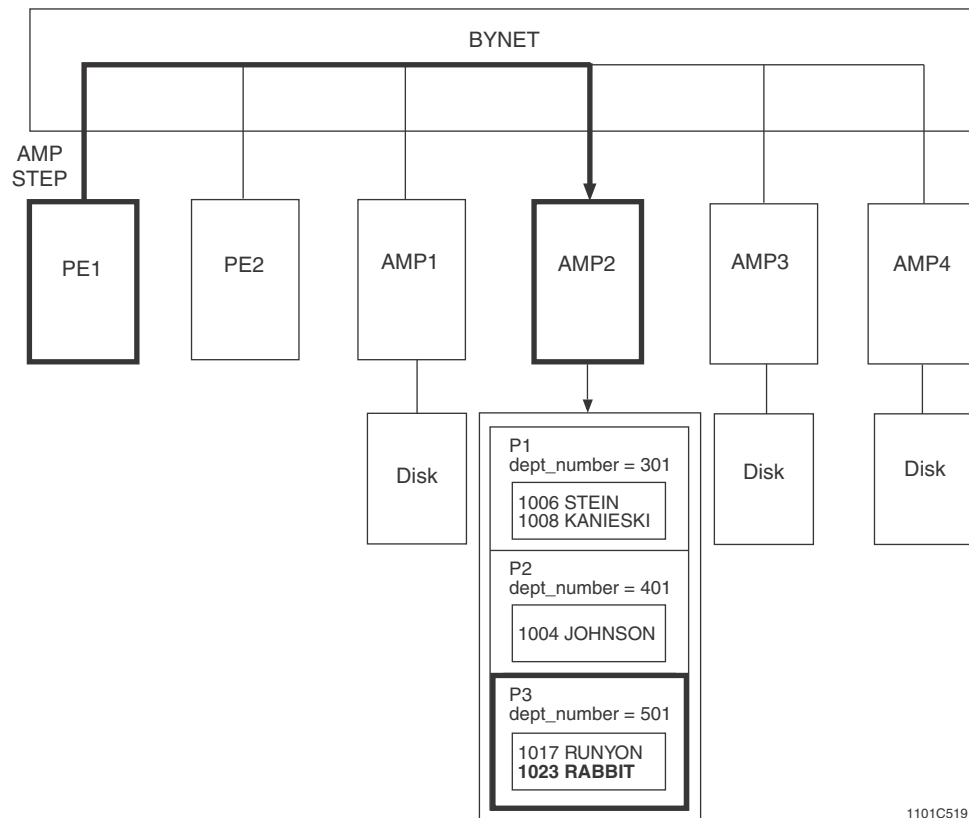


The AMP Step includes the list of partitions (in this case, P3) to access. Partition elimination (in this case, static partition elimination) reduces access to the partitions that satisfy the query requirements. In each partition in the list (in this case, only P3), look for rows with a given row hash value of the PI.

Partition elimination is similar for the Employee table with a multilevel PPI where one partitioning expression uses the dept_number column and another partitioning expression uses the hire_date column.

Assume that a PE receives the following SELECT statement:

```
SELECT last_name
FROM Employee
WHERE employee_number = 1023
AND dept_number = 501
AND hire_date BETWEEN DATE '2006-01-01' AND DATE '2006-12-31';
```

The following flow diagram illustrates this process.



```
                              BYNET

AMP
STEP
        PE1      PE2      AMP1     AMP2     AMP3     AMP4

                         Disk              Disk     Disk

                                  P1
                                  dept_number = 301
                                  1/01/05 ≤ hire_date ≤ 12/31/05

                                     1006 STEIN

                                  P2
                                  dept_number = 301
                                  1/01/06 ≤ hire_date ≤ 12/31/06

                                     1008 KANIESKI

                                  P3
                                  dept_number = 401
                                  1/01/05 ≤ hire_date ≤ 12/31/05

                                     1004 JOHNSON

                                  P5
                                  dept_number = 501
                                  1/01/05 ≤ hire_date ≤ 12/31/05

                                     1017 RUNYON

                                  P6
                                  dept_number = 501
                                  1/01/06 ≤ hire_date ≤ 12/31/06

                                     1023 RABBIT
```
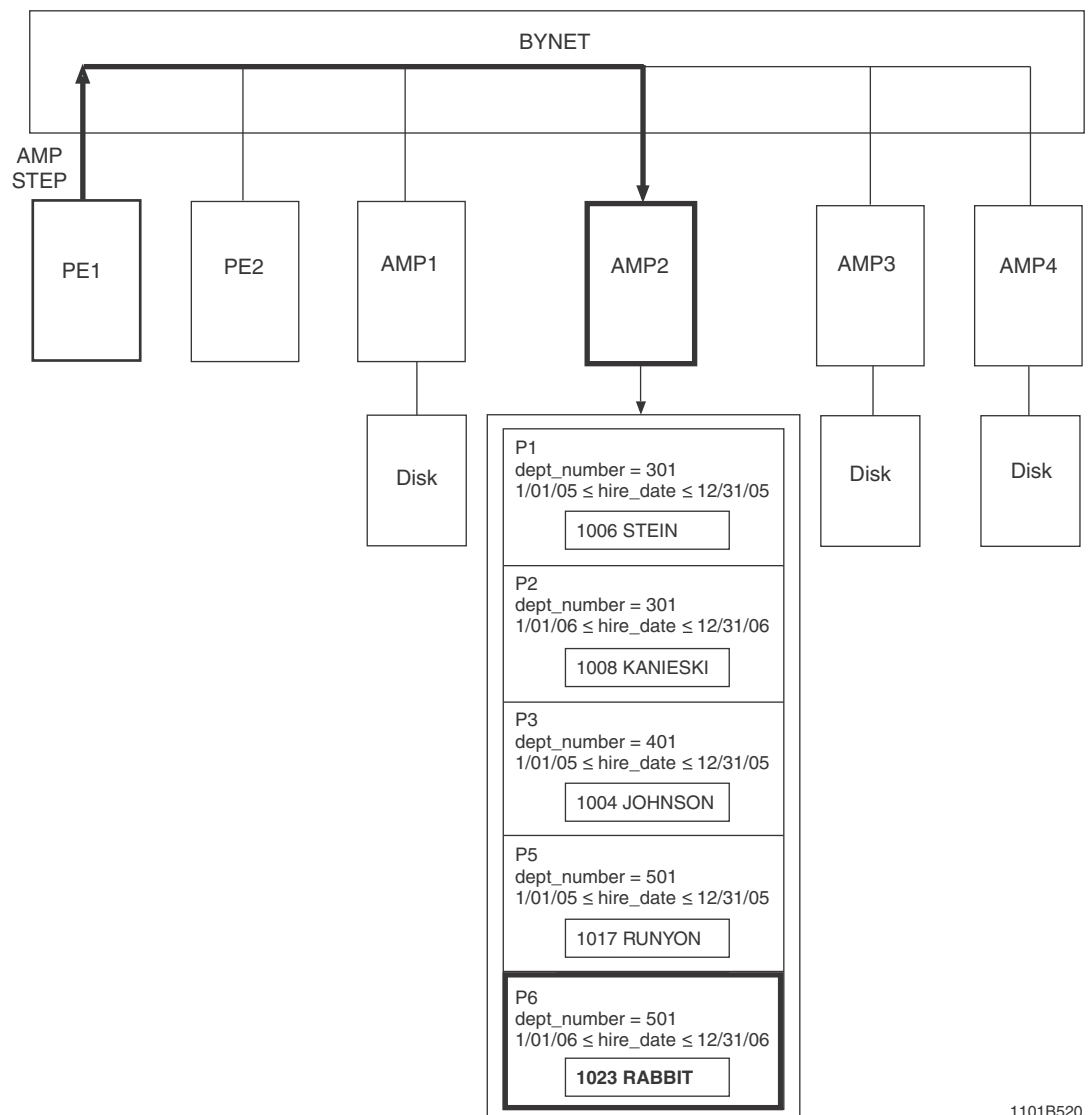
1101B520

No one was hired in department number 401 in 2006, so partition P4 is empty.

For more information on partition elimination, see *Database Design*.

# Table Access

Teradata Database uses indexes and partitions to access the rows of a table. If indexed or partitioned access is not suitable for a query, or if a query accesses a NoPI table that does not have an index defined on it, the result is a full-table scan.

## Access Methods

The following table access methods are available to the Optimizer:

- Unique Primary Index
- Unique Partitioned Primary Index
- Nonunique Primary Index
- Nonunique Partitioned Primary Index
- Unique Secondary Index

- Nonunique Secondary Index
- Join Index
- Hash Index
- Full-Table Scan
- Partition Scan

## Effects of Conditions in WHERE Clause

For a query on a table that has an index defined on it, the predicates or conditions that appear in the WHERE clause of the query determine whether the system can use row hashing, or do a table scan with partition elimination, or whether it must do a full-table scan.

The following functions are applied to rows identified by the WHERE clause, and have no effect on the selection of rows from the base table:

- GROUP BY
- HAVING
- INTERSECT
- MINUS/EXCEPT
- ORDER BY

- QUALIFY
- SAMPLE
- UNION
- WITH ... BY
- WITH

Statements that specify any of the following WHERE clause conditions result in full-table scans (FTS). If the table has a PPI, partition elimination might reduce the FTS access to only the affected partitions.

- nonequality comparisons
- *column_name* IS NOT NULL
- *column_name* NOT IN (explicit list of values)
- *column_name* NOT IN (subquery)
- *column_name* BETWEEN ... AND ...
- condition_1 OR condition_2
- NOT condition_1
- *column_name* LIKE
- column_1 || column_2 = value
- table1.column_x = table1.column_y
- table1.column_x [computation] = value
- table1.column_x [computation] - table1.column_y

- INDEX (*column_name*)
- SUBSTR (*column_name*)
- SUM
- MIN
- MAX
- AVG
- DISTINCT
- COUNT
- ANY
- ALL
- missing WHERE clause

The type of table access that the system uses when statements specify any of the following WHERE clause conditions depends on whether the column or columns are indexed, the type of index, and its selectivity:

- *column_name* = value or constant expression
- *column_name* IS NULL
- *column_name* IN (explicit list of values)
- *column_name* IN (subquery)
- condition_1 AND condition_2
- different data types
- table1.column_x = table2.column_x

In summary, a query influences processing choices:

- A full-table scan (possibly with partition elimination if the table has a PPI) is required if the query includes an implicit range of values, such as in the following WHERE examples.

  When a small BETWEEN range is specified, the Optimizer can use row hashing rather than a full-table scan.

  ```
  ... WHERE column_name [BETWEEN <, >, <>, <=, >=]
  ... WHERE column_name [NOT] IN (SELECT...)
  ... WHERE column_name NOT IN (val1, val2 [,val3])
  ```
- Row hashing can be used if the query includes an explicit value, as shown in the following WHERE examples:

  ```
  ... WHERE column_name = val
  ... WHERE column_name IN (val1, val2, [,val3])
  ```

### Related Topics

| For more information on … | See … |
| --- | --- |
| the efficiency, number of AMPs used, and the number of rows accessed by all table access methods | *Database Design*. |
| strengths and weaknesses of table access methods | *Introduction to Teradata*. |
| full-table scans | . |

# Full-Table Scans

A full-table scan is a retrieval mechanism that touches all rows in a table.

Teradata Database always uses a full-table scan to access the data of a table if a query:

- Accesses a NoPI table that does not have an index defined on it
- Does not specify a WHERE clause

Even when results are qualified using a WHERE clause, indexed or partitioned access may not be suitable for a query, and a full-table scan may result.

A full-table scan is always an all-AMP operation, and should be avoided when possible. Full-table scans may generate spool files that can have as many rows as the base table.

Full-table scans are not something to fear, however. The architecture that Teradata Database uses makes a full-table scan an efficient procedure, and optimization is scalable based on the number of AMPs defined for the system. The sorts of unplanned, ad hoc queries that characterize the data warehouse process, and that often are not supported by indexes, perform very effectively for Teradata Database using full-table scans.

## Accessing Rows in a Full-Table Scan

Because full-table scans necessarily touch every row on every AMP, they do not use the following mechanisms for locating rows:

- Hashing algorithm and hash map
- Primary indexes
- Secondary indexes or their subtables
- Partitioning

Instead, a full-table scan uses the Master Index and Cylinder Index file system tables to locate each data block. Each row within a data block is located by a forward scan.

Because rows from different tables are never mixed within the same data block and because rows never span blocks, an AMP can scan up to 128K bytes of the table on each block read, making a full-table scan a very efficient operation. Data block read-ahead and cylinder reads can also increase efficiency.

## Related Topics

| For more information on … | See … |
| --- | --- |
| full-table scans | *Database Design*. |
| cylinder reads | *Database Administration*. |
| enabling data block read-ahead operations | DBS Control Utility in *Utilities*. |

# Collecting Statistics

The COLLECT STATISTICS (Optimizer form) statement collects demographic data for one or more columns of a base table, hash index, or join index, computes a statistical profile of the collected data, and stores the synopsis in the Data Dictionary.

The Optimizer uses the synopsis data when it generates its table access and join plans.

## Usage

You should collect statistics on newly created, empty data tables. An empty collection defines the columns, indexes, and synoptic data structure for loaded collections. You can easily collect statistics again after the table is populated for prototyping, and again when it is in production.

You can collect statistics on a:

- Unique index, which can be:
  - Primary or secondary
  - Single or multiple column
  - Partitioned or nonpartitioned
- Nonunique index, which can be:
  - Primary or secondary
  - Single or multiple column
  - Partitioned or nonpartitioned
  - With or without COMPRESS fields
- Non-indexed column or set of columns, which can be:
  - Partitioned or nonpartitioned
  - With or without COMPRESS fields
- Join index
- Hash index
- NoPI table
- Temporary table
  - If you specify the TEMPORARY keyword but a materialized table does not exist, the system first materializes an instance based on the column names and indexes you specify. This means that after a true instance is created, you can update (re-collect) statistics on the columns by entering COLLECT STATISTICS and the TEMPORARY keyword without having to specify the desired columns and index.
  - If you omit the TEMPORARY keyword but the table is a temporary table, statistics are collected for an empty base table rather than the materialized instance.
- Sample (system-selected percentage) of the rows of a data table or index, to detect data skew and dynamically increase the sample size when found.
  - The system does not store both sampled and defined statistics for the same index or column set. Once sampled statistics have been collected, implicit re-collection hits the same columns and indexes, and operates in the same mode. To change this, specify any keywords or options and name the columns and/or indexes.

## Related Topics

| For more information on … | See … |
|---|---|
| using the COLLECT STATISTICS statement | *SQL Data Definition Language.* |
| collecting statistics on a join index | *Database Design.* |
| collecting statistics on a hash index | |
| when to collect statistics on base table columns instead of hash index columns | |
| database administration and collecting statistics | *Database Administration.* |

# APPENDIX A  Notation Conventions

This appendix describes the notation conventions used in this book.

Throughout this book, three conventions describe the SQL syntax and code:

* Syntax diagrams, used to describe SQL syntax form, including options. See "Syntax Diagram Conventions" on page 173.

* Square braces in the text, used to 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:

        * without specifying the precision value *n* or scale value *m* specifying precision (n) only

        * specifying both values (n and m)

        * you cannot specify scale without first defining precision.

    * CHARACTER [(n)] means that use of (n) is optional.

        The values for *n* and *m* are integers in all cases

* Japanese character code shorthand notation, used to represent unprintable Japanese characters. See "Character Shorthand Notation In This Book" on page 178.

# Syntax Diagram Conventions

## Notation Conventions

| Item | Definition / Comments |
|------|------------------------|
| Letter | An uppercase or lowercase alphabetic character ranging from A through Z. |
| Number | A digit ranging from 0 through 9.<br>Do not use commas when typing a number with more than 3 digits. |

| Item | Definition / Comments |
|------|----------------------|
| Word | Keywords and variables. <br><br> • UPPERCASE LETTERS represent a keyword. <br>  Syntax diagrams show all keywords for SQL statements in uppercase, unless operating system restrictions require them to be in lowercase. <br> • lowercase letters represent a keyword that you must type in lowercase, such as a Linux command. <br> • Mixed Case letters can be used to represent functions, methods, and other non-SQL keywords that can be entered in uppercase, lowercase or mixed case. <br> • *lowercase italic letters* represent a variable such as a column or table name. <br>  Substitute the variable with a proper value. <br> • **lowercase bold letters** represent an excerpt from the diagram. The excerpt is defined immediately following the diagram that contains it. <br> • UNDERLINED LETTERS represent the default value. This applies to both uppercase and lowercase words. |
| 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:
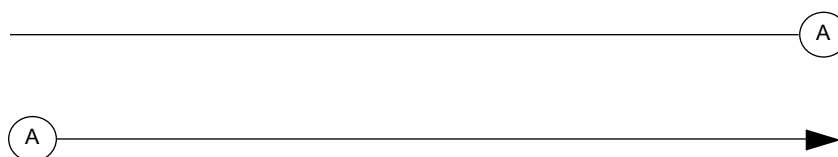


FE0CA002

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:

SHOW ──────────────────────────────────▶

FE0CA003

If you can choose from more than one entry, the choices appear vertically, in a stack. The first entry appears on the main path:

SHOW ──────┬── CONTROLS ──┬──────────────────▶
           └── VERSIONS ──┘

FE0CA005
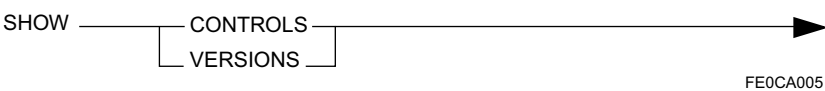
## Optional Entries

You may choose to include or disregard optional entries. Optional entries appear below the main path:

SHOW ──────┬──────────────┬──────────────────▶
           └── CONTROLS ──┘

FE0CA004
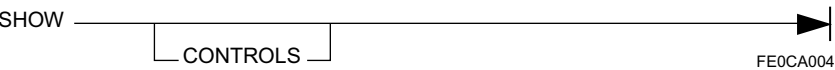
If you can optionally choose from more than one entry, all the choices appear below the main path:

──────┬── READ ───┬──────
      ├── SHARE ──┤
      └── ACCESS ─┘

JC01A010
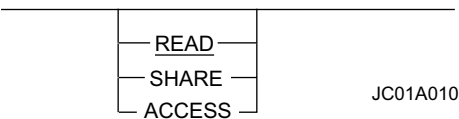
Some commands and statements treat one of the optional choices as a default value. This value is <u>UNDERLINED</u>. It is presumed to be selected if you type the command or statement without specifying one of the options.

## Strings

String literals appear in apostrophes:

────────── 'msgtext ' ──────────

JC01A004
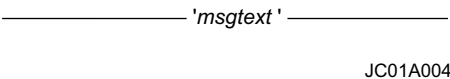
## Abbreviations

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.

```
SHOW ────────┬── CONTROLS ──┬──────────────────────▶│
             └── CONTROL  ──┘
                                            FE0CA042
```

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:

```
        ┌────── , ──────┬─── 3 ───┐
        │   ┌── , ──( 4 )──┐      │
        │   │              │      │
────────┴─( ┴──── cname ────┴── ) ─┴────────
                                    JC01B012
```
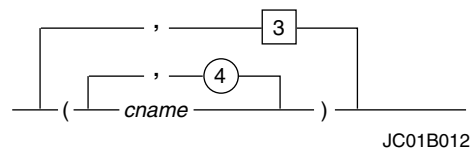
Read loops from right to left.

The following conventions apply to loops:

| IF... | THEN... |
|---|---|
| there is a maximum number of entries allowed | the number appears in a circle on the return path.<br>In the example, you may type *cname* a maximum of 4 times. |
| there is a minimum number of entries required | the number appears in a square on the return path.<br>In the example, you must type at least three groups of column names. |
| a separator character is required between entries | the character appears on the return path.<br>If the diagram does not show a separator character, use one blank space.<br>In the example, the separator character is a comma. |

| IF... | THEN... |
|-------|---------|
| a delimiter character is 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:
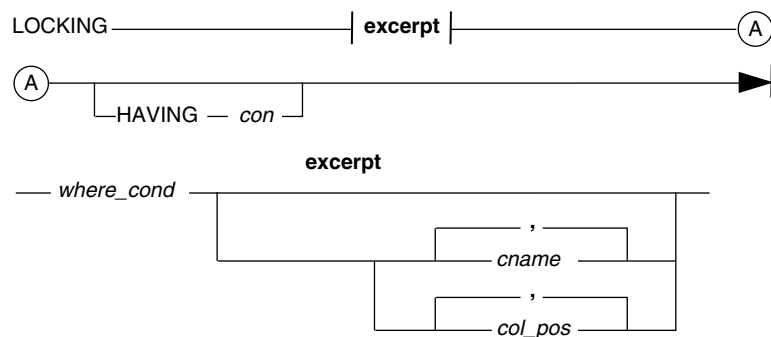


JC01A014

## Multiple Legitimate Phrases

In a syntax diagram, it is possible for any number of phrases to be legitimate:
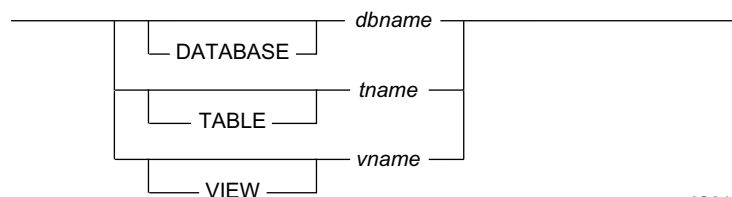


JC01A016

In this example, any of the following phrases are legitimate:

- *dbname*
- DATABASE *dbname*
- *tname*

- TABLE *tname*

- *vname*

- VIEW *vname*

## Sample Syntax Diagram
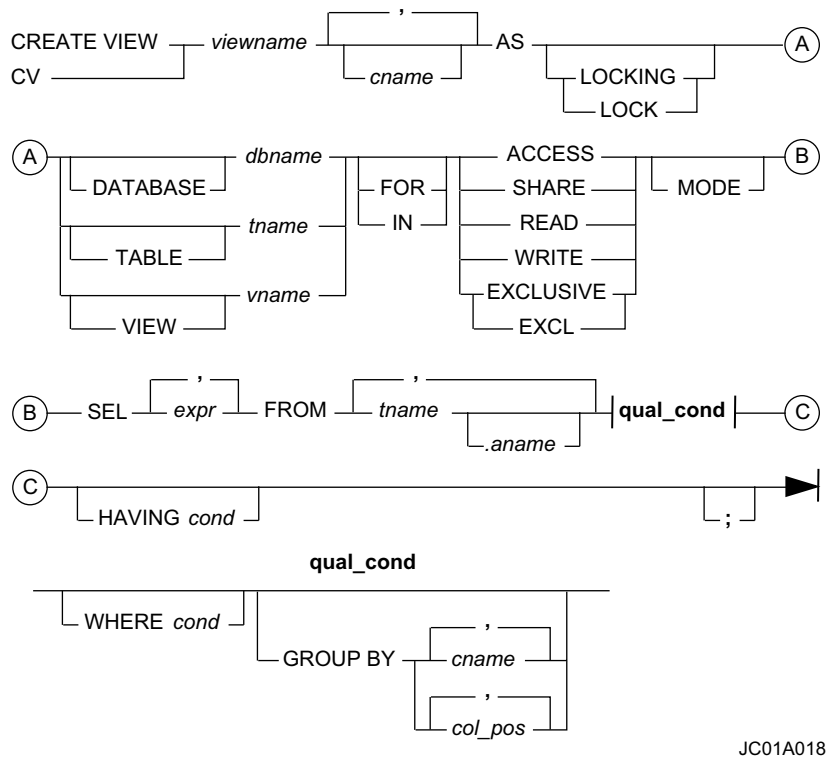


JC01A018

## Diagram Identifier

The alphanumeric string that appears in the lower right corner of every diagram is an internal identifier used to catalog the diagram. The text never refers to this string.

# Character Shorthand Notation In This Book

## Introduction

This book 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 book, 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 *International Character Set Support*.

## Character Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

| Symbol | Encoding | Meaning |
|---|---|---|
| a–z<br>A–Z<br>0–9 | Any | Any single byte Latin letter or digit. |
| <u>a</u>–<u>z</u><br><u>A</u>–<u>Z</u><br><u>0</u>–<u>9</u> | Any | Any fullwidth Latin letter or digit. |
| < | KanjiEBCDIC | Shift Out [SO] (0x0E).<br>Indicates transition from single to multibyte character in KanjiEBCDIC. |
| > | KanjiEBCDIC | Shift In [SI] (0x0F).<br>Indicates transition from multibyte to single byte KanjiEBCDIC. |
| **T** | Any | Any multibyte character.<br>The encoding depends on the current character set.<br>For KanjiEUC, code set 3 characters are always preceded by "$ss_3$". |
| **I** | Any | Any single byte Hankaku Katakana character.<br>In KanjiEUC, it must be preceded by "$ss_2$", 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 server character sets.

| 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 |

# Restricted Words

Teradata Database and ANSI/ISO SQL standards restrict the use of certain words as identifiers because those words may be incorrectly interpreted as SQL keywords. Restricted words should not be used as database object names or as parameters in application programs that interface with the database. The categories of restricted words are:

| Category | Description |
|---|---|
| Reserved words | These words are used as keywords by Teradata Database or ANSI/ISO SQL. They cannot be used as identifiers to name database objects, such as databases, tables, columns, or stored procedures. They also must not be used as macro or stored procedure parameters or local variables, host variables, or correlation names. |
| | Reserved words can be reserved by Teradata Database, by the ANSI/ISO SQL standard, or by both. |
| Future reserved words | These words are likely to be used as Teradata Database, keywords in the future. Like reserved words, future reserved words cannot be used as identifiers. |
| Nonreserved words | These words may become keywords in the future. |
| | Teradata does not recommend using Teradata Database nonreserved words as identifiers if these words are reserved words in the ANSI standard. These words may become Teradata Database reserved words in the future. |

Note:  Teradata Parallel Transporter (Teradata PT) has a different set of restricted words. For a list of Teradata PT restricted words, see *Teradata Parallel Transporter Reference*.

You can use the SQLRestrictedWords view and SQLRestrictedWords_TBF function to see or query the restricted words in the current or previous releases of Teradata Database.

# SQLRestrictedWords View

The SQLRestrictedWords view lists all restricted words for the current release of Teradata Database.

SQLRestrictedWords is created in the SYSLIB database by the DIPDEM script, which is run automatically by the DIP utility when Teradata Database is installed.

The view contains these columns:

| Column Name | Description |
|---|---|
| restricted_word | The restricted word. |
| category | One of the following characters that represents the category of the Teradata restricted word:<br>• R: a Teradata reserved word<br>• F: a Teradata future reserved word<br>• N: a Teradata nonreserved word |
| ANSI_restricted | One of the following characters that represents the ANSI category of the restricted word:<br>• R: an ANSI reserved word<br>• N: an ANSI nonreserved word<br>• T: a word that is neither reserved nor nonreserved by current ANSI standards, but that is restricted by Teradata |

## Usage Notes

To retrieve or query Teradata SQL restricted words from Teradata Database releases other than the current release, see "SQLRestrictedWords_TBF Function" on page 183.

Queries of SQLRestrictedWords are case specific.

## Example 1 - Getting the Restricted Words for the Current Release

This query returns the restricted words for the current Teradata Database release. Because it returns all columns from the view, it includes the Teradata and ANSI categories for the words. A portion of the output is shown below.

```
    SELECT * FROM SYSLIB.SQLRestrictedWords;

restricted_word                 category  ANSI_restricted
-----------------------------   --------  ---------------
ABS                             R         R
ACCOUNT                         R         T
ACCESS_LOCK                     R         T
ABORTSESSION                    R         T
ABORT                           R         T
...                             ...       ...
```

## Example 2 - Getting Nonreserved Words that are ANSI Reserved Words

The following query returns the nonreserved words for the current Teradata Database release that are ANSI reserved words. A portion of the output is shown below.

Note: This result is useful because Teradata does not recommend using Teradata Database nonreserved words as identifiers if these words are reserved words in the ANSI standard. These words may become Teradata Database reserved words in the future.

```
SELECT restricted_word
  FROM SYSLIB.SQLRestrictedWords
  WHERE category='N' AND ANSI_restricted='R';
```

```
restricted_word
-----------------------------
ALLOCATE
CALLED
CONDITION
GLOBAL
MEMBER
...
```

# SQLRestrictedWords_TBF Function

## Purpose

SQLRestrictedWords_TBF is a table function that can be used to query the restricted words for the current or previous releases of Teradata Database. This is useful for migration or upgrade planning, or in cases when you want to back down to the previous Teradata Database release.

For each word, the table indicates the category of restriction (reserved, nonreserved, or future reserved), the Teradata Database release when the word was introduced or dropped as a restricted word, and whether the word is reserved, nonreserved, or neither in the current ANSI/ISO SQL standard.

SQLRestrictedWords_TBF is created in the SYSLIB database by the DIPDEM script, which is run automatically by the DIP utility when Teradata Database is installed.

## Syntax
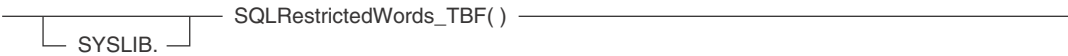
```
────┬─────────────── SQLRestrictedWords_TBF( ) ───────────────────
    └─ SYSLIB. ─┘
```
1101A752

## ANSI Compliance

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

## Result

The output table includes these columns:

| Column Name | Description |
|---|---|
| restricted_word | The restricted word. |
| release_introduced | The release when the restricted word was introduced.<br><br>The release number is presented in this format:<br><br>*MM.mm*<br>where *MM* are 2 digits representing a major release number and *mm* are 2 digits representing a minor release number. For example, '06.02', '12.00', or '13.10'.<br><br>Note: The value of the release_introduced field for restricted words that were introduced in Teradata Database 6.2 or in earlier releases is '06.02'. |
| release_dropped | The release when the restricted word was dropped. The release number is in the same format as for the release_introduced column.<br><br>A NULL value indicates that the restricted word has not been dropped, and is currently a restricted word. |
| category | One of the following characters that represents the category of the Teradata restricted word:<br><br>• R: a Teradata reserved word<br>• F: a Teradata future reserved word<br>• N: a Teradata nonreserved word |
| ANSI_restricted | One of the following characters that represents the ANSI category of the restricted word:<br><br>• R: an ANSI reserved word<br>• N: an ANSI nonreserved word<br>• T: a word that is neither reserved nor nonreserved by current ANSI standards, but that is restricted by Teradata |

## Usage Notes

Table functions can be used only in the FROM clause of an SQL SELECT statement.

For more information on:

• table functions, see the discussion of Table UDFs in *SQL Functions, Operators, Expressions, and Predicates*

• the use of table functions in queries, see the TABLE option of the FROM clause in *SQL Data Manipulation Language*

Queries using the SQLRestrictedWords_TBF function are case specific.

## Example 1: Getting the Restricted Words for a Specific Release

The following query returns the restricted words for Teradata Database 12.0. Note that the query includes words that were restricted in all releases up to and including 12.0, and excludes words that were dropped as restricted words prior to release 12.0. A portion of the output is included below. Note also that the reference to release 12.0 must be entered in the query as '12.00'.

```
SELECT * FROM TABLE (SYSLIB.SQLRestrictedWords_TBF()) AS t1
    WHERE release_introduced <= '12.00' AND
        (release_dropped > '12.00' OR release_dropped IS NULL);
```

| restricted_word | release_introduced | release_dropped | category | ANSI_restricted |
|-----------------|--------------------|-----------------|----------|-----------------|
| SPL | 06.02 | ? | N | T |
| SQLDATA | 12.00 | 13.10 | N | N |
| SQLSTATE | 06.02 | ? | N | R |
| AbortSessions | 12.00 | ? | N | T |
| GetQueryBand | 12.00 | ? | N | T |
| GetQueryBandValue | 12.00 | ? | N | T |
| IdentifyDatabase | 12.00 | ? | N | T |
| IdentifySession | 12.00 | ? | N | T |

Note that even though SQLDATA was dropped as a restricted word for release 13.10, it was a restricted word for release 12.0, so it is returned by the query.

## Example 2: Double Entries in the Restricted Words Table

It is possible to have two rows returned for a specific restricted word if the word was dropped and later reintroduced, or if the restriction category was changed for the word. In this example, the results show that NUMBER was changed from a nonreserved word to a reserved word in release 14.00.

```
SELECT * FROM TABLE (SYSLIB.SQLRestrictedWords_TBF()) AS t1
    WHERE restricted_word = 'NUMBER';
```

| restricted_word | release_introduced | release_dropped | category | ANSI_restricted |
|-----------------|--------------------|-----------------|----------|-----------------|
| NUMBER | 14.00 | ? | R | N |
| NUMBER | 13.00 | 14.00 | N | N |

# APPENDIX C ANSI/ISO SQL Compliance

Teradata Database conforms closely to the ANSI/ISO SQL standard while supporting extensions that enable users to take full advantage of the efficiency benefits of parallelism. Teradata develops new features that conform to existing ANSI/ISO standards and handles differences between Teradata SQL and ANSI/ISO SQL as follows:

| WHEN ... | THEN ... |
| --- | --- |
| the difference between the Teradata SQL dialect and the ANSI/ISO SQL standard for a language feature is slight | the ANSI/ISO SQL is added to Teradata Database feature as an option. |
| the difference between the Teradata SQL dialect and the ANSI/ISO SQL standard for a language feature is significant | both syntaxes are offered and the user has the choice of operating in either Teradata or ANSI mode or of turning off SQL Flagger. The mode can be defined:<br><br>• Persistently<br>  Use the SessionMode field of the DBS Control Record to define session mode characteristics.<br><br>• For a session<br>  Use the BTEQ .SET SESSION TRANSACTION command to control transaction semantics.<br>  Use the BTEQ .SET SESSION SQLFLAG command to control use of the SQL Flagger.<br>  Use the SQL statement SET SESSION DATEFORM to control how data typed as DATE is handled. |

## Terminology Differences Between ANSI/ISO SQL and Teradata SQL

The ANSI/ISO SQL standard and Teradata SQL occasionally use different terminology. The following table lists the more important variances.

| ANSI/ISO SQL | Teradata SQL |
|---|---|
| Base table | Table<br><br>In the ANSI/ISO SQL standard, the term table means:<br><br>• A base table<br>• A viewed table (view)<br>• A derived table |
| Binding style | Not defined, but implicitly includes the following:<br><br>• Interactive SQL<br>• Embedded SQL<br>• ODBC<br>• CLIv2 |
| Authorization ID | User ID |
| Catalog | Dictionary |
| CLI | ODBC<br><br>ANSI CLI is not exactly equivalent to ODBC, but the ANSI standard is heavily based on the ODBC definition. |
| Direct SQL | Interactive SQL |
| Domain | Not defined |
| External routine function | User-defined function (UDF) |
| Module | Not defined |
| Persistent stored module | Stored procedure |
| Schema | User<br>Database |
| SQL database | Relational database |
| Viewed table | View |
| Not defined | Explicit transaction<br><br>ANSI transactions are always implicit, beginning with an executable SQL statement and ending with either a COMMIT or a ROLLBACK statement. |
| Not defined | CLIv2<br><br>Teradata CLIv2 is an implementation-defined binding style. |
| Not defined | Macro<br><br>The function of Teradata Database macros is similar to that of ANSI persistent stored modules without having the loop and branch capabilities stored modules offer. |

# Using the SQL Flagger

Teradata provides the SQL Flagger to help users identify non-standard SQL. SQL Flagger always permits statements flagged as non-entry-level or non-compliant ANSI/ISO SQL to execute. Its task is to return a warning message to the requestor noting the noncompliance.

Flagging is enabled by a client application before a session is logged on and generally is used only to assist in checking for ANSI compliance in code that must be portable across multiple vendor environments.

The SQL Flagger is disabled by default. You can enable or disable it using any of the following procedures, depending on your application.

| For this software ... | Use these commands or options ... | To turn the SQL Flagger ... |
|---|---|---|
| BTEQ | .[SET] SESSION SQLFLAG ENTRY | to entry-level ANSI |
|  | .[SET] SESSION SQLFLAG NONE | off |
|  | For details on using BTEQ commands, see *Basic Teradata Query Reference*. | |
| Preprocessor2 | SQLFLAGGER(ENTRY) | to entry-level ANSI |
|  | SQLFLAGGER(NONE) | off |
|  | For details on setting Preprocessor options, see *Teradata Preprocessor2 for Embedded SQL Programmer Guide.* | |
| CLI | set lang_conformance = '2'  set lang_conformance to '2' | to entry-level ANSI |
|  | set lang_conformance = 'N' | off |
|  | For details on setting the conformance field, see *Teradata Call-Level Interface Version 2 Reference for Mainframe-Attached Systems* and *Teradata Call-Level Interface Version 2 Reference for Workstation-Attached Systems*. | |

# SQL Statements and SQL Requests

In Teradata SQL, a *statement,* a syntactic concept, has three components:

- A statement keyword, such as SELECT, CREATE TABLE, or ALTER PROCEDURE
- Zero or more expressions, functions, keywords, clauses, or phrases, such as WHERE, PRIMARY KEY, or UNIQUE PRIMARY INDEX
- A semicolon

In Teradata SQL, a *request* defines a unit of work that is transmitted from a client application to Teradata Database in a single message. A request is a *semantic* concept that has several components, including:

- One or more SQL statements

- Request-level CLIv2 options parcel
- Metadata about the request data

All individual SQL statements, called single statement requests, are also individual SQL requests, but not all SQL requests are also SQL statements because one request can contain an unlimited number of SQL statements. This special case is called a multistatement request, and it is the primary superficial characteristic that distinguishes a statement from a request in the Teradata world. For more information, see "Multistatement Requests" on page 130.

# Performance Considerations

This Appendix provides suggestions for improving database query performance.

# Using the 2PC Protocol

Two-Phase Commit (2PC) is an IMS and CICS protocol for committing update transactions processed by multiple systems that do not share the same locking and recovery mechanism.

For detailed information on 2PC, see *Teradata Director Program Reference*.

## Performance Impact

Consider the following disadvantages of using the 2PC protocol:

- Performance may decrease because, at the point of synchronization, up to two additional messages are exchanged between the coordinator and participant, in addition to the normal messages that update the database

  If your original SQL request took longer to complete than your other requests, the performance impact due to the 2PC overhead will be less noticeable.

- • If Teradata Database restarts, and a session using the 2PC protocol ends up in an IN-DOUBT state, Teradata Database holds data locks indefinitely until you resolve the IN-DOUBT session. During this time, other work could be blocked if it accesses the same data for which Teradata Database holds those locks.

  To resolve this situation, perform the following steps:

  - Use the COMMIT/ROLLBACK command to resolve manually the IN-DOUBT sessions.
  - Use the RELEASE LOCKS command.
  - Use the RESTART command to restart your system.

2PC causes no system overhead when it is disabled.

# System Validated Object Names

The system determines the validity of object names based on the rules presented in the topics beginning with "Object Names" on page 78, and according to the settings of the related DBS Control fields.

The entries in the following topics define whether various specifications in common SQL DDL statements are subject to object naming rules. The listed names follow the same object naming rules when specified in DCL and DML statements.

## Names Subject to Object Naming Rules

| Name | Example Usage |
|---|---|
| ACCOUNT (*acc*) | CREATE USER u1 as perm=10e6, password = pass1, account=**'*acc2*'**, default journal table=u1jnl; |
| | MODIFY USER u1 AS ACCOUNT=**'*acc1*'**; |
| | BEGIN QUERY LOGGING WITH SQL ON ALL ACCOUNT =**'*acc1*'**; |
| | CREATE PROFILE prof5 AS PERM=10e5 PASSWORD=pass5 ACCOUNT= '**_acc3_**'; |
| ATTRIBUTE (*att*) | CREATE TYPE udt2 AS (**_att1_** INT, **_att2_** DATE) NOT FINAL; |
| | ALTER TYPE udt2 ADD ATTRIBUTE **_att3_** FLOAT; |
| AUTHORIZATION (*auth*) | CREATE AUTHORIZATION db1.**_auth1_** as INVOKER USER 'bdUsr' PASSWORD 'bdPsswd'; |
| COLUMN (*col*) | CREATE FUNCTION fn1 (NumRows INTEGER)<br>RETURNS TABLE<br>(*col1* INTEGER,<br> *col2* INTEGER)<br>LANGUAGE C<br>NO SQL<br>PARAMETER STYLE SQL<br>EXTERNAL NAME 'SS!easy!easy.c'; |
| | CREATE TABLE t1 (*col1* int); |

| Name | Example Usage |
|------|---------------|
| CONSTRAINT (*con*) | `CREATE TABLE t2 (col1 int CONSTRAINT `**`con2`**`  CHECK (i=1));`<br>`CREATE TABLE t3 (col3 int CONSTRAINT `**`con3`**<br>`REFERENCES t1(c1));`<br>`ALTER TABLE t1 ADD CONSTRAINT `*`con1`*` UNIQUE (c1);` |
| *CONSTRAINT* (*con*)<br><br>(row level security constraint) | `CREATE CONSTRAINT `*`conA`*` data_type,[NULL|NOT NULL],`<br>`VALUES value_name:integer_code ...`<br>`[, value_name:integer_code],`<br>`Insert SYSLIB.insert_udf_name ,`<br>`Update SYSLIB.update_udf_name ,`<br>`Delete SYSLIB.delete_udf_name ,`<br>`Select SYSLIB.select_udf_name ;` |
| *DATABASE* (*db*) | `CREATE DATABASE `*`db1`*` as perm=10e6;` |
| *FUNCTION* (*fn*) | `CREATE FUNCTION `*`fn2`*` (integer, float)`<br>`RETURNS FLOAT`<br>`LANGUAGE C`<br>`NO SQL`<br>`EXTERNAL NAME   'sp:/Teradata/tdbs_udf/usr/`<br>`second1.so';` |
| | `RENAME FUNCTION fn1 AS `*`fn2`*`;` |
| | `ALTER SPECIFIC FUNCTION `*`fn1`*` EXECUTE PROTECTED;` |
| | `CREATE CAST (udt1 as udt2) WITH FUNCTION`<br>`sysudtlib.`*`fn1`*`(udt1);` |
| | `CREATE ORDERING FOR udt1`<br>`ORDER FULL BY MAP WITH FUNCTION SYSUDTLIB.`*`fn3`*`;` |
| | `CREATE TRANSFORM FOR abov_strInt`<br>`abov_strInt_IO (TO SQL WITH SPECIFIC FUNCTION`<br>`SYSUDTLIB.abov_StrIntToSQL,`<br>`FROM SQL WITH SPECIFIC FUNCTION SYSUDTLIB.`*`fn4`*`);` |
| GLOP Set (*glop*) | `CREATE GLOP SET db.`*`glop1`*`;` |
| INDEX (*idx*) | `CREATE INDEX `*`idx2`*` (c1) on t1;` |
| | `COLLECT STAT USING SAMPLE ON db1.`*`idx1`*<br>`COLUMN col1 FROM db2.t1;` |
| | `CREATE TABLE t1 (col1 int, col2 int)`<br>`INDEX `*`idx1`*` (c2);` |
| | `COLLECT STAT INDEX `*`idx1`*` ON tab1;` |
| MACRO | `CREATE MACRO m1 AS (SELECT 'abc'; );` |
| | `RENAME MACRO m1 to m2;` |

| Name | Example Usage |
|------|---------------|
| METHOD (*mth*) | ```
CREATE CONSTRUCTOR METHOD mth1
(P1 INTEGER)
FOR abov_strInt
EXTERNAL NAME
'SS!udt_strintcons!udt_strintcons.c!F!strintcons';
``` |
| | ```
CREATE TYPE udt1 AS (P1 INTEGER)
NOT FINAL CONSTRUCTOR METHOD mth2 (P1 INTEGER)
RETURNS udt1
SELF AS RESULT
LANGUAGE C
DETERMINISTIC
NO SQL;
``` |
| | ```
ALTER SPECIFIC METHOD mth2 FOR UDT_name EXECUTE
NOT PROTECTED;
``` |
| PARAMETER (*pm*) | ```
CREATE MACRO m1 (pm1 int) AS (SEL :pm1; );
``` |
| | ```
CREATE FUNCTION Find_Text (pm1 VARCHAR (500), pm2
VARCHAR (500))
RETURNS CHAR
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
EXTERNAL NAME
'SS:pattern2:/home/i18n/ca3/v2r5/UDF/tests/SrcNI/
pattern2/pattern2.c:SI:pattern2:/home/i18n/ca3/
v2r5/UDF/tests/SrcNI/pattern2/
pattern2.h:SL:curses';
``` |
| | ```
CREATE PROCEDURE xsp_cr003(IN pm1 varchar(20), OUT
pm2 VARCHAR(20))
LANGUAGE C
PARAMETER STYLE SQL
EXTERNAL NAME 'CS!xsp_cr003!xsp_cr003.c';
``` |
| | ```
CREATE TYPE abov_strlnt AS (pm2 INTEGER)
NOT FINAL CONSTRUCTOR METHOD abov_strlnt (P1
INTEGER)
RETURNS abov_strlnt
SELF AS RESULT
LANGUAGE C
DETERMINISTIC
NO SQL;
``` |
| | ```
REPLACE PROCEDURE sp3(out pm1 integer)
BEGIN
  DECLARE var1 INTEGER DEFAULT 10;
  SET p1 = var1;
END;
``` |
| PASSWORD (*pass*) | ```
CREATE USER u1 as perm=10e6,
password = pass1 , account='acc1';
``` |
| | ```
MODIFY USER u1 AS PASSWORD=pass1;
``` |

| Name | Example Usage |
|---|---|
| PLAN_DIRECTIVE (*pd*) | ```
INSERT PLAN_DIRECTIVE pd1 IN pd1cat
(sel   'pd_dip002_TestID1_1_pd',
       t100k_b.i1, pd_db1.t100k_b.i2,
pd_db1.t100k_b.i5
from  t100k_b, pd_db1.t200_a
where
       t100k_b.i2 = pd_db1.t200_a.i2
;
)
WITH '1: NESTED JOIN(DUPED( SCAN(
PD_DB1.T100K_B)),'
'INDEXED WITH NO ROWIDLIST( INDEX(I2)
PD_DB1.T200_A))'
COMMENT 'PD for pd_dip002_TestID1_1_pd'
``` |
| PROCEDURE (sp) | ```
CREATE PROCEDURE sp2 ( a varchar(20), OUT result1
VARCHAR(20))
LANGUAGE C
PARAMETER STYLE SQL
EXTERNAL NAME 'CS!xsp_cr003!xsp_cr003.c';
```<br><br>```
RENAME PROCEDURE sp2 TO sp3;
```<br><br>```
ALTER PROCEDURE sp3 LANGUAGE C COMPILE ONLY;
``` |
| PROFILE (*prof*) | ```
CREATE PROFILE prof5 AS
    PERM=10e5
    PASSWORD=pass5
    ACCOUNT= 'acc5';
``` |
| PROXYUSER (*pxyuser*)<br><br>PROXYROLE (*pxyrole*) | ```
SET QUERY_BAND='PROXYUSER=pxyuser1;
PROXYROLE=pxyrole1;' FOR SESSION;
```<br><br>```
BEGIN TRANSACTION;
SET QUERY_BAND='PROXYROLE=pxyrole2;' FOR
TRANSACTION;
SELECT *
FROM table1;
```<br>Note: The reserved query band names PROXYUSER and PROXYROLE are not permitted in the profile query band. |
| QUERY (*qry*) | ```
WITH qry1(a) AS (SELECT col1 FROM tab1)
SELECT a FROM qry1;
``` |
| ROLE (*role*) | ```
CREATE ROLE role1;
``` |
| EXTERNAL ROLE (*extrole*) | ```
CREATE EXTERNAL ROLE extrole2;
``` |

| Name | Example Usage |
|------|---------------|
| TABLE (*t*) | `CREATE TABLE t1 (col1 int);` |
| | `CREATE ERROR TABLE t2 FOR t1;` |
| | `RENAME TABLE t1 AS t3;` |
| | `MODIFY USER u1 AS DEFAULT JOURNAL TABLE=t1;` |
| | `CREATE TRIGGER trig5 AFTER UPDATE ON t3`<br>`REFERENCING OLD_TABLE AS t1 NEW_TABLE AS t2FOR`<br>`EACH Statement`<br>`  When (35 <= (sel X.price from t2 X INNER JOIN t1`<br>`Y ON`<br>`                         X.pubyear= Y.pubyear))`<br>`(`<br>`   INSERT t4 SELECT t2.titles, t2.price FROM t2;`<br>`);` |
| | `BEGIN LOADING db1.t3`<br>`ERRORFILES db1.t1, db1.t2`<br>`INSERT INTO db1.t3.*;` |
| | `BEGIN DELETE MLOAD TABLES db1.t1`<br>`WITH db1.t2`<br>`ERRORTABLES db1.lt3;` |
| | `BEGIN IMPORT MLOAD TABLES db1.t1`<br>`WORKTABLES db1.t2`<br>`ERRORTABLES db1.t3 db1.t4 ;` |
| | `CREATE USER u1 as perm=10e6,`<br>`password = pass1, default journal table=t1;` |
| TRANSFORM_GROUP (*tg*) | `CREATE TRANSFORM FOR abov_strInt`<br>`tg1 (TO SQL WITH SPECIFIC FUNCTION`<br>`SYSUDTLIB.abov_StrIntToSQL,`<br>`FROM SQL WITH SPECIFIC FUNCTION`<br>`SYSUDTLIB.abov_StrIntFromSQL);` |
| | `DROP TRANSFORM tg2 FOR arsv_point_t ;` |
| TRIGGER (*trig*) | `CREATE TRIGGER trig1 AFTER  UPDATE OF (i)`<br>`ON t1 REFERENCING OLD AS OLDROW`<br>`NEW AS NEWROW FOR EACH ROW`<br>`WHEN (i > 10)`<br>`(INSERT INTO t1log VALUES (OLDROW.i, NEWROW.i); );` |
| | `RENAME TRIGGER trig1 TO trig2 ;` |

| Name | Example Usage |
|---|---|
| UDT (*udt*) | `CREATE TYPE udt15 AS VARCHAR(15) FINAL;` |
| | `CREATE CAST (udt1 as udt2) WITH FUNCTION`<br>`sysudtlib.abov_DistFloattoDistInt(udt1)` |
| | `CREATE ORDERING FOR udt3`<br>`ORDER FULL BY MAP WITH FUNCTION`<br>`SYSUDTLIB.abov_StrIntOrdering;` |
| | `CREATE TRANSFORM FOR udt3`<br>`abov_strInt_IO (TO SQL WITH SPECIFIC`<br>`FUNCTION SYSUDTLIB.abov_StrIntToSQL,`<br>`FROM SQL WITH SPECIFIC FUNCTION`<br>`SYSUDTLIB.abov_StrIntFromSQL);` |
| | `DROP TRANSFORM transform_io FOR udt4 ;` |
| USER_NAME (*u*) | `CREATE USER u1 as perm=10e6,`<br>`password = pass1, account='u1account' ;` |
| USING VARIABLE NAME (*usingvar*) | `.import data file = c:\temp\fil1.data`<br>`using (usingvar1 int, usingvar2 char(10))`<br>`insert db1.tab1(:usingvar1, :usingvar2);` |
| VIEW (*v*) | `CREATE VIEW v1 AS SELECT 'abc' col1;` |
| | `REPLACE VIEW v1 AS SELECT 'abc' col1;` |
| | `RENAME VIEW v1 AS v2;` |

# Names Not Subject to Object Naming Rules

The following items are not subject to object naming rules.

| Name | Example Usage |
|---|---|
| CHEC, inKPOINT (*chkpt*) | `CREATE USER u3 AS PERM=1E6,`<br>`PASSWORD=u3, DEFAULT JOURNAL TABLE = jnl1;`<br>`CHECKPOINT jnl1, NAME chkpt1 ;` |
| COLLATION SEQUENCE (*COLL*) | `SET SESSION COLLATION ASCII` |
| JAR (*jar*) | `CALL SQLJ.INSTALL_JAR`<br>`('CJ:tempJar_a.jar','jar1',0);` |
| TRANSLATION (*trans*) | `TRANSLATE(c1 USING trans1)` |

# Glossary

**2PC**    Two-Phase Commit

**AMP**    Access Module Processor vproc

**ANSI**    American National Standards Institute

**BLOB**    Binary Large Object

**BTEQ**    Basic Teradata Query facility

**BYNET**    Banyan Network - High speed interconnect

**CJK**    Chinese, Japanese, and Korean

**CLIv2**    Call Level Interface Version 2

**CLOB**    Character Large Object

**cs0, cs1, cs2, cs3**    Four code sets (codeset 0, 1, 2, and 3) used in EUC encoding.

**distinct type**    A UDT that is based on a single predefined data type

**E2I**    External-to-Internal

**EUC**    Extended UNIX Code

**external routine**    UDF, UDM, or external stored procedure that is written using C, C++, or Java

**external stored procedure**    a stored procedure that is written using C, C++, or Java

**FK**    Foreign Key

**HI**    Hash Index

**I2E**    Internal-to-External

**JI**    Join Index

**JIS**    Japanese Industrial Standards

**LOB**    Large Object

**LT/ST**    Large Table/Small Table (join)

**NoPI tables**    Tables that are defined with no primary index (PI)

**NPPI**    Nonpartitioned Primary Index

**NUPI**    Nonunique Primary Index

**NUSI**  Nonunique Secondary Index

**OLAP**  Online Analytical Processing

**OLTP**  Online Transaction Processing

**PDE**  Parallel Database Extensions

**PE**  Parsing Engine vproc

**PI**  Primary Index

**PK**  Primary Key

**PPI**  Partitioned Primary Index

**predefined type**  Teradata Database system type such as INTEGER and VARCHAR

**QCD**  Query Capture Database

**RDBMS**  Relational Database Management System

**SDF**  Specification for Data Formatting

**stored procedure**  a stored procedure that is written using SQL statements

**structured type**  A UDT that is a collection of one or more fields called attributes, each of which is defined as a predefined data type or other UDT (which allows nesting)

**UCS-2**  Universal Coded Character Set containing 2 bytes

**UDF**  User-Defined Function

**UDM**  User-Defined Method

**UDT**  User-Defined Type

**UPI**  Unique Primary Index

**USI**  Unique Secondary Index

**vproc**  Virtual Process

# Index