TERADATA

Teradata Database

SQL Fundamentals

Release 13.0
B035-1141-098A
March 2010

Preface

Purpose

SQL Fundamentals describes basic Teradata SQL concepts, including data handling, SQL data definition, control, and manipulation, and the SQL lexicon.

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 Release

This book supports Teradata® Database 13.0.

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 13.0 March 2010	Updated Appendix E with changes to SQL syntax.

Release	Description
Teradata Database 13.0 November 2009	• Deleted concurrent transaction limit from Appendix C. • Deleted system maximum for replication groups from Appendix C.
Teradata Database 13.0 August 2009	Provided correct system maximum for replication groups.
Teradata Database 13.0 April 2009	• Added material to support No Primary Index (NoPI) tables • Added material to support Period and Geospatial data types • Changed material on UDFs to include Java UDFs and dynamic UDTs • Changed material on stored procedure access rights and default database • Added new material to support changes to triggers • Modified material on archiving and restoring views, macros, stored procedures, and triggers • Updated list of DDL and DCL statements • Removed restrictions regarding compression on foreign key columns and the USI primary key columns for soft and batch referential integrity • Updated ANSI SQL:2003 references to ANSI SQL:2008, including reserved and nonreserved words • Updated Appendix E with changes to SQL syntax
Teradata Database 12.0 October 2007	Changed QUERY_BAND from a reserved word to a nonreserved word

Teradata Database 12.0
September 2007
	• Changed topic on index hash mapping to reflect increased hash bucket size of 20 bits (1048576 hash buckets)
	• Changed the rules on naming objects in Chapter 2 in support of the UNICODE Data Dictionary feature
	• Changed the description of profiles to include the option to specify whether to restrict a password string from containing certain words
	• Added material to support mutilevel partitioned primary indexes
	• Added material to support error logging tables for errors that occur during INSERT … SELECT and MERGE operations
	• Removed material on Express Logon
	• Updated Appendix E with the following:
	  • New statements for the online archive feature: LOGGING ONLINE ARCHIVE ON and LOGGING ONLINE ARCHIVE OFF
	  • New statements for error logging tables: CREATE ERROR TABLE and DROP ERROR TABLE
	  • LOGGING ERRORS option for INSERT … SELECT and MERGE
	  • New syntax for CREATE PROCEDURE and REPLACE PROCEDURE to support C and C++ external stored procedures that use CLIv2 to execute SQL, Java external stored procedures, and dynamic result sets

Release	Description

	• Updated Appendix E with the following:
	  • New SET QUERY_BAND statement
	  • New syntax for ALTER TABLE, CREATE TABLE, and CREATE JOIN INDEX to support multilevel partitioned primary indexes

Additional Information

URL	Description
www.info.teradata.com/	Use the Teradata Information Products Publishing Library site to: • View or download a manual: 1 Under **Online Publications**, select **General Search**. 2 Enter your search criteria and click **Search**. • Download a documentation CD-ROM: 1 Under **Online Publications**, select **General Search**. 2 In the **Title or Keyword** field, enter *CD-ROM*, and click **Search**. • Order printed manuals: Under **Print & CD Publications**, select **How to Order**.
www.teradata.com	The Teradata home page provides links to numerous sources of information about Teradata. Links include: • Executive reports, case studies of customer experiences with Teradata, and thought leadership • Technical information, solutions, and expert advice • Press releases, mentions and media resources
www.teradata.com/t/TEN/	Teradata Customer Education designs, develops and delivers education that builds skills and capabilities for our customers, enabling them to maximize their Teradata investment.
www.teradataatyourservice.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 e-mail: teradata-books@lists.teradata.com

References to Microsoft Windows and Linux

This book refers to “Microsoft Windows” and “Linux.” For Teradata Database 13.0, these references mean:

- “Windows” is Microsoft Windows Server 2003 64-bit.
- “Linux” is SUSE Linux Enterprise Server 9 and SUSE Linux Enterprise Server 10.

Table of Contents

CHAPTER 1 Objects

This chapter describes the objects you use to store, manage, and access data in Teradata Database.

Databases and Users

Definitions

A *database* is a collection of related tables, views, triggers, indexes, stored procedures, user-defined functions, and macros. A database also contains an allotment of space from which users can create and maintain their own objects, or other users or databases.

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

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

The difference between users and databases in Teradata Database has important implications for matters related to privileges, but neither the differences nor their implications are easy to understand. This is particularly true with respect to understanding fully the consequences of implicitly granted privileges.

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

Definitions

A *table* is what 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.

The CREATE TABLE statement specifies a table name, one or more column names, and the attributes of each column. CREATE TABLE can also specify datablock size, percent freespace, 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 and one or more secondary indexes.

If you define a table and you do not specify a PRIMARY INDEX clause, NO PRIMARY INDEX clause, PRIMARY KEY constraint, or UNIQUE constraint, 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 NoPI table without a primary index, use DBS Control to change the value of the PrimaryIndexDefault General field. For details, see *Utilities*.)

For details on indexes, see "Indexes" on page 33. For details on NoPI tables, see "No Primary Index (NoPI) Tables" on page 17.

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 three 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 stroring 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 18 and "Volatile Tables" on page 23.

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. • a marker row that you can use to determine the number of error rows for the request. 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. • an error row for each error that was detected prior to index maintenance. The absence of a marker row in the error logging table means the request was aborted.
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*.
primary indexes	"Primary Indexes" on page 38.
secondary indexes	"Secondary Indexes" on page 42.

Global Temporary Tables

Introduction

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 … 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

For 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 …	Does this …	The result is this …
1	1	The DBA creates a global temporary table definition in the database scheme named globdb.gt1 according to the following CREATE TABLE statement: `CREATE GLOBAL TEMPORARY TABLE globdb.gt1, LOG (f1 INT NOT NULL PRIMARY KEY, f2 DATE, f3 FLOAT) ON COMMIT PRESERVE ROWS;`	The global temporary table definition is created and stored in the database schema.

Step	Session ...	Does this ...	The result is this ...
2	1	User_1 logs on an SQL session and references globdb.gt1 using the following INSERT statement: `INSERT globdb.gt1 (1, 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. Immediately upon materialization, the table is populated with a single row having the following values. f1=1 f2=980101 f3=11.1 This means that the contents of this local instance of the global temporary table definition is *not* empty when it is created. 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. `SELECT * FROM globdb.gt1;`	No rows are returned because Session 2 has not yet materialized a local instance of globdb.gt1.
4	2	User_2 issues the following INSERT statement: `INSERT globdb.gt1 (2, 980202, 22.2);`	Session 2 creates a local instance of the global temporary table definition globdb.gt1. The table is populated, immediately upon materialization, with a single row having the following values. f1=2 f2=980202 f3=22.2 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 again issues the following SELECT statement: `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 ...	Does this ...	The result is this ...
1	3	User_1 logs on another SQL session 3 and issues the following SELECT statement: `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: `INSERT globdb.gt1 (3, 980303, 33.3);`	Session 3 created a local instance of the global temporary table definition globdb.gt1. The table is populated, immediately upon materialization, with a single row having the following values. f1=3 f2=980303 f3=33.3 From this point on, any INSERT/DELETE/UPDATE statement that references globdb.gt1 in Session 3 maps to this local instance of the table.
3	3	User_1 again issues the following SELECT statement: `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: `CREATE INDEX (f2) ON TEMPORARY globdb.gt1;`	An empty local global temporary table named globdb.gt1 is created for Session 4. This is one of only two cases in which a local instance of a global temporary table is materialized without data. The other would be a COLLECT STATISTICS statement—in this case, the following statement: `COLLECT STATISTICS ON TEMPORARY globdb.gt1;`
5	4	User_2 issues the following SELECT statement: `SELECT * FROM globdb.gt1;`	No rows are returned because the local instance of globdb.gt1 for Session 4 is empty.

Step	Session ...	Does this ...	The result is this ...
6	4	User_2 issues the following SHOW TABLE statement: `SHOW TABLE globdb.gt1;`	`CREATE SET GLOBAL TEMPORARY TABLE globdb.gt1, FALLBACK, LOG ( f1 INTEGER NOT NULL, f2 DATE FORMAT 'YYYY-MM-DD', f3 FLOAT) UNIQUE PRIMARY INDEX (f1) ON COMMIT PRESERVE ROWS;`
7	4	User_2 issues the following SHOW TEMPORARY TABLE statement: `SHOW TEMPORARY TABLE globdb.gt1;`	`CREATE SET GLOBAL TEMPORARY TABLE globdb.gt1, FALLBACK, LOG ( f1 INTEGER NOT NULL, f2 DATE FORMAT 'YYYY-MM-DD', f3 FLOAT) UNIQUE PRIMARY INDEX (f1) INDEX (f2) ON COMMIT PRESERVE ROWS;` 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

Creating 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 Maintenence 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	In Session 1, you …	In Session 2, you …	The result is this …
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.
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.

Stage	In Session 1, you …	In Session 2, you …	The result is this …
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. 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. **Note:** Volatile tables take precedence over permanent tables in the same database in a session. 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). 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.

Columns

Definition

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. For more information on PPIs, see “Partitioned and Nonpartitioned Primary Indexes” on page 36.
PARTITION#L1 through PARTITION#L15	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. For more information on row identifiers, see “Row Hash and RowID” on page 33.

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 27.
CREATE TABLE and the column definition clause	*SQL Data Definition Language*.
system-derived and system-generated columns	*Database Design*.

Data Types

Introduction

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.

The set of data values that a column defines can belong to one of the following groups of data types:

- Numeric
- Character
- Datetime
- Interval
- Byte
- UDT
- Period
- Geospatial

Numeric Data Types

A numeric value is either an exact numeric number (integer or decimal) or an approximate numeric number (floating point). Use the following SQL data types to specify numeric values.

Type	Description
BIGINT	Represents a signed, binary integer value from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
INTEGER	Represents a signed, binary integer value from -2,147,483,648 to 2,147,483,647.
SMALLINT	Represents a signed binary integer value in the range -32768 to 32767.
BYTEINT	Represents a signed binary integer value in the range -128 to 127.

Type	Description
REAL	Represent a value in sign/magnitude form.
DOUBLE PRECISION	
FLOAT	
DECIMAL [(*n*[,*m*])]	Represent a decimal number of *n* digits, with *m* of those *n* digits to the right of the decimal point.
NUMERIC [(*n*[,*m*])]	

Character Data Types

Character data types represent characters that belong to a given character set. Use the following SQL data types to specify character data.

Type	Description
CHAR[(*n*)]	Represents a fixed length character string for Teradata Database internal character storage.
VARCHAR(*n*)	Represents a variable length character string of length *n* for Teradata Database internal character storage.
LONG VARCHAR	LONG VARCHAR specifies the longest permissible variable length character string for Teradata Database internal character storage.
CLOB	Represents a large character string. A character large object (CLOB) column can store character data, such as simple text, HTML, or XML documents.

DateTime Data Types

DateTime values represent dates, times, and timestamps. Use the following SQL data types to specify DateTime values.

Type	Description
DATE	Represents a date value that includes year, month, and day components.
TIME [WITH TIME ZONE]	Represents a time value that includes hour, minute, second, fractional second, and [optional] time zone components.
TIMESTAMP [WITH TIIME ZONE]	Represents a timestamp value that includes year, month, day, hour, minute, second, fractional second, and [optional] time zone components.

Interval Data Types

An interval value is a span of time. There are two mutually exclusive interval type categories.

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

Byte Data Types

Byte data types store raw data as logical bit streams. For any machine, BYTE, VARBYTE, and BLOB data is transmitted directly from the memory of the client system.

Type	Description
BYTE	Represents a fixed-length binary string.
VARBYTE	Represents a variable-length binary string.
BLOB	Represents a large binary string of raw bytes. A binary large object (BLOB) column can store binary objects, such as graphics, video clips, files, and documents.

BLOB is ANSI SQL:2008-compliant. BYTE and VARBYTE are Teradata extensions to the ANSI SQL:2008 standard.

UDT Data Types

UDT data types are custom data types that you define to model the structure and behavior of data that your application deals with. Teradata Database supports *distinct* and *structured* UDTs.

Type	Description
Distinct	A UDT that is based on a single predefined data type, such as INTEGER or VARCHAR.
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).

For more details on UDTs, including a synopsis of the steps you take to develop and use UDTs, see "User-Defined Types" on page 74.

Period Data Types

A period data type represents a set of contiguous time granules that extends from a beginning bound up to but not including an ending bound.

Type	Description
PERIOD(DATE)	Represents an anchored duration of DATE elements that include year, month, and day components.
PERIOD(TIME[(*n*)] [WITH TIME ZONE])	Represents an anchored duration of TIME elements that include hour, minute, second, fractional second, and [optional] time zone components.
PERIOD(TIMESTAMP[(*n*)] [WITH TIME ZONE])	Represents an anchored duration of TIMESTAMP elements that include year, month, day, hour, minute, second, fractional second, and [optional] time zone components.

Geospatial Data Types

Geospatial data types provide a way for applications that manage, analyze, and display geographic information to interface with Teradata Database. Use the following data types to represent geographic information.

<table>
<tr><th>Type</th><th>Description</th></tr>
<tr><td>ST_Geometry</td><td>A Teradata proprietary internal UDT that can represent any of the following geospatial types:
<table>
<tr><th>Type</th><th>Description</th></tr>
<tr><td>ST_Point</td><td>0-dimensional geometry that represents a single location in two-dimensional coordinate space.</td></tr>
<tr><td>ST_LineString</td><td>1-dimensional geometry usually stored as a sequence of points with a linear interpolation between points.</td></tr>
<tr><td>ST_Polygon</td><td>2-dimensional geometry consisting of one exterior boundary and zero or more interior boundaries, where each interior boundary defines a hole.</td></tr>
<tr><td>ST_GeomCollection</td><td>Collection of zero or more ST_Geometry values.</td></tr>
<tr><td>ST_MultiPoint</td><td>0-dimensional geometry collection where the elements are restricted to ST_Point values.</td></tr>
<tr><td>ST_MultiLineString</td><td>1-dimensional geometry collection where the elements are restricted to ST_LineString values.</td></tr>
<tr><td>ST_MultiPolygon</td><td>2-dimensional geometry collection where the elements are restricted to ST_Polygon values.</td></tr>
</table></td></tr>
<tr><td>ST_Geometry (continued)</td><td>
<table>
<tr><th>Type</th><th>Description</th></tr>
<tr><td>GeoSequence</td><td>Extension of ST_LineString that can contain tracking information, such as time stamps, in addition to geospatial information.</td></tr>
</table>
The ST_Geometry type supports methods for performing geometric calculations.</td></tr>
<tr><td>MBR</td><td>A Teradata proprietary internal UDT that provides a way to obtain the minimum bounding rectangle (MBR) of a geometry for tessellation purposes.</td></tr>
</table>

The implementation of the ST_Geometry type closely follows *ISO/IEC 13249-3, Information technology — Database languages — SQL Multimedia and Application Packages — Part 3: Spatial*, referred to as SQL/MM Spatial. The GeoSequence and MBR types are extensions to SQL/MM Spatial.

Related Topics

For detailed information on data types, see *SQL Data Types and Literals* and *SQL Geospatial Types*.

Keys

Definitions

Term	Definition
Primary Key	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
Foreign Key	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 53.

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 38 and "Secondary Indexes" on page 42.

Differences Between 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

Definition

An index is a mechanism that the SQL query optimizer can use to make table access more performant. 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.

Difference Between Classic Indexes and Teradata Database Indexes

The classic index for a relational database is itself a file made up of rows having two 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 “NUSI Bit Mapping” on page 45.

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 "Partitioned and Nonpartitioned Primary Indexes" on page 36.

For tables with no PPI ...	For tables with a PPI ...
For tables with no 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.	For tables defined with a PPI, 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. 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. 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

Perhaps not so obvious is the disadvantage of using 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

Teradata Database provides four different index types:

- Primary index

  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 types of unique index.

- 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 "Join Indexes" on page 36.

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 37.

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.

  You can use the Teradata Statistics Wizard client utility to recommend which columns and indexes to collect statistics on for particular query workloads. The Statistics Wizard also recommends when to recollect statistics.

- 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*
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* • *Performance Management*
using the Teradata Index Wizard client utility	• *Teradata Index Wizard User Guide* • *SQL Request and Transaction Processing*
collecting and maintaining accurate database statistics	"COLLECT STATISTICS" in *SQL Data Definition Language*
using the Teradata Statistics Wizard client utility	*Teradata Statistics Wizard User Guide*

Primary Indexes

Introduction

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 33).

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.

WHEN a CREATE TABLE statement defines a …			
Primary Index	Primary Key	Unique Column Constraint	THEN 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. If the data type of the first column in the table is UDT or 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.

Generally, other considerations can include the following:

- 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, see *Database Design*.

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, UDT, ST_Geometry, MBR, or Period data type.
- A partitioning column cannot be a column that has a GRAPHIC or VARGRAPHIC data type or specifies the CHARACTER SET GRAPHIC data type attribute.
- 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:

- Character comparison (implicit or explicit) is not allowed.
- The expression for the RANGE_N test value must evaluate to BYTEINT, SMALLINT, INTEGER, or DATE.
- 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 Summary

Teradata Database primary indexes have the following 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

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

- *Database Design*
- *Performance Management*

Secondary Indexes

Introduction

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.

Differences Between 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)` `ON customer_table;`
NUSI	`CREATE INDEX (customer_name)` `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. For value-ordered NUSIs, the rows are sorted by NUSI value order.
The appropriate AMP receives the data and creates a row in the index subtable. If the AMP receives a row with a duplicate index value, an error is reported.	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. 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` `WHERE customer_number=12;`
NUSI	`SELECT * FROM customer_table` `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.

USI Access Process	NUSI Access Process
The process is complete. 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. 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 ...	See ...
multiple secondary index access	*Database Design*
composite secondary index access	
other aspects of index selection	

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* • *SQL Data Manipulation Language*

Secondary Index Summary

Teradata SQL secondary indexes have the following 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.

Summary of USI and NUSI Properties

Unique and nonunique secondary indexes have the following properties.

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

For More Information About Secondary Indexes

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

Also consult the following manuals for more detailed information on using secondary indexes to enhance the performance of your databases:

- *Database Design*
- *Performance Management*

Join Indexes

Introduction

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 fast path final access point 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

Join indexes affect the following Teradata Database functions and features.

- 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.

Comparison of 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.
- 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.
- Create explicit unique indexes on its columns.

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* • *Performance Management* • *SQL Data Definition Language*
• database design considerations for join indexes • improving join index performance	*Database Design*

Hash Indexes

Introduction

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.

Hash Index	Single-Table Join 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. 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.

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*

Comparison of 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 49.

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.

For More Information About Hash Indexes

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

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

Referential Integrity

Introduction

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 non-null value for a *referencing* column cannot exist in a table if an equal value does not exist in a *referenced* column.

Varieties of Referential Integrity Enforcement Supported by Teradata Database

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.

Terms Related to Referential Integrity

The following terms are used to explain the concept of referential integrity.

<table>
<tr><th>Term</th><th>Definition</th></tr>
<tr><td>Child Table</td><td>A table where the referential constraints are defined.
Child table and referencing table are synonyms.</td></tr>
<tr><td>Parent Table</td><td>The table referenced by a child table.
Parent table and referenced table are synonyms.</td></tr>
<tr><td>Primary Key</td><td rowspan="2">A unique identifier for a row of a table.</td></tr>
<tr><td>UNIQUE Alternate Key</td></tr>
<tr><td>Foreign Key</td><td>A column set in the child table that is also the primary key (or a UNIQUE alternate key) in the parent table.
Foreign keys can consist of as many as 64 different columns.</td></tr>
<tr><td>Referential Constraint</td><td>A constraint defined on a column set or a table to ensure referential integrity.
For example, consider the following table definition:
<pre>CREATE TABLE A
(A1 CHAR(10) REFERENCES B (B1), /* 1 */
 A2 INTEGER
FOREIGN KEY (A1,A2) REFERENCES C /* 2 */
PRIMARY INDEX (A1));</pre>This CREATE TABLE statement specifies the following referential integrity constraints.
<table>
<tr><th>This constraint ...</th><th>Is defined at this level ...</th></tr>
<tr><td>1</td><td>column.
Implicit foreign key A1 references the parent key B1 in table B.</td></tr>
<tr><td>2</td><td>table.
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.
Both parent table columns must also be defined as NOT NULL.</td></tr>
</table>
</td></tr>
</table>

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. Teradata Database automatically enforces referential integrity.
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 Are Allowed

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

If the FOREIGN KEYS in TableA are on columns defined as nullable, then rows could be inserted into TableA with nulls for the FOREIGN KEY columns. Once the appropriate rows exist in TableB, the nulls of the FOREIGN KEY columns in TableA could then be updated to contain non-null values which match the TableB values.

References Can Be 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.

For this data manipulation activity …	The system verifies that …
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.

An ALTER TABLE statement adds a foreign key reference to a table.
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.
	When the system parses ALTER TABLE, it defines an error table that:
	• Has the same columns and primary index as the target table of the ALTER TABLE statement.
	• Has a name that is the same as the target table name suffixed with the reference index number.
	A reference index number is assigned to each foreign key constraint for a table.
	To determine the number, use one of the following system views.
	  • RI_Child_TablesV
	  • RI_Distinct_ChildrenV
	  • RI_Distinct_ParentsV
	  • RI_Parent_TablesV
	• Is created under the same user or database as the table being altered.
	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.
	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).
	It is your responsibility to:
	• Correct data values in the referenced or referencing tables so that full referential integrity exists between the two tables.
	Use the rows in the error table to define which corrections to make.
	• 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 dictionary definition of that table is also restored. The 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 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 further 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 are used to 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.

Reasons to Use 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 on Views

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

Definition

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 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.
• enable a trigger • disable a trigger • change the creation timestamp for a trigger	ALTER TRIGGER. 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. 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 for a Trigger

The general process flow for a trigger is as follows. 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 as follows:

   - 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 on Using Triggers

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 ...
• guidelines for creating triggers • conditions that cause triggers to fire • trigger action that occurs when a trigger fires • the trigger action time • when to use row triggers and when to use statement triggers	CREATE TRIGGER in *SQL Data Definition Language*.
• temporarily disabling triggers • enabling triggers • changing the creation timestamp of a trigger	ALTER TRIGGER in *SQL Data Definition Language*.
permanently removing triggers from the system	DROP TRIGGER in *SQL Data Definition Language*.

Macros

Introduction

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.

Definition

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.

Performing a macro is similar to performing a multistatement request (see "Multistatement Requests" on page 135).

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

Macros in an embedded SQL program are performed in one of the following ways.

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 -or- • the PREPARE and EXECUTE statements to specify dynamic execution. Use DESCRIBE to verify that the single statement of the macro is not a data returning statement.

IF the macro ...	THEN use ...
• consists of multiple statements • returns data	a cursor, either static or dynamic. 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 further 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

Introduction

Stored procedures are called Persistent Stored Modules in the ANSI 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 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: CREATE PROCEDURE

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: REPLACE PROCEDURE

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: CREATE PROCEDURE":

```
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.

Stored procedures 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*. • *Database Administration*.

External Stored Procedures

Introduction

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.

User-Defined Functions

Introduction

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

User-defined functions (UDFs) allow you to extend SQL by writing your own functions in the C, C++, or Java programming language, installing them on the database, and then using them like standard SQL functions.

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

UDF Types

Teradata Database supports three types of UDFs.

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

Usage

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

Definition

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

An administrator can define a profile and assign it to a group of users who share the same settings.

Advantages of Using Profiles

Use profiles to:

- 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
  - 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

Definition

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

Use roles to:

- 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 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

Introduction

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 UDFs. The benefit of dynamic UDTs is that they significantly increase the number of input arguments that you can pass in to UDFs.

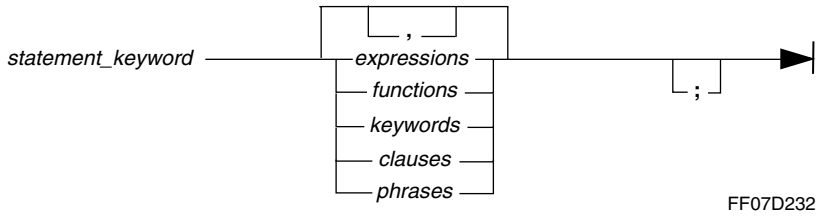
CHAPTER 2 Basic SQL Syntax and Lexicon

This chapter explains the syntax and lexicon for Teradata SQL, a single, unified, nonprocedural language that provides capabilities for queries, data definition, data modification, and data control of Teradata Database.

Structure of an SQL Statement

Syntax

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



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. 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. The semicolon separates statements in a multistatement request and terminates a request when it is the last nonblank character on an input line in BTEQ. Note that the request terminator is required for a request defined in the body of a macro. For a discussion of macros and their use, see "Macros" on page 62.

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 the 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 77
keywords	
expressions	"Expressions" on page 78
functions	"Functions" on page 104
separators	"Separators" on page 106
terminators	"Terminators" on page 108

SQL Lexicon Characters

Client Character Data

The characters that make up the SQL lexicon can be represented on the client system in ASCII, EBCDIC, UTF8, UTF16, or in an installed user-defined character set.

If the client system character data is not ASCII, then it is converted by Teradata Database to an internal form for processing and storage. Data returned to the client system is converted to the client character set.

Server Character Data

The internal forms used for character support are described in *International Character Set Support*.

The notation used for Japanese characters is described in:

- "Character Shorthand Notation Used In This Book"
- Appendix A: "Notation Conventions"

Case Sensitivity

See the following topics in *SQL Data Types and Literals*:

- "Default Case Sensitivity of Character Columns"
- "CASESPECIFIC Phrase"
- "UPPERCASE Phrase"
- "Character String Literals"

See the following topics in *SQL Functions, Operators, Expressions, and Predicates*:

- "LOWER Function"
- "UPPER Function"

Keywords

Introduction

Keywords are words that have special meanings in SQL statements. There are two types of keywords: *reserved* and *nonreserved*. Reserved keywords are included in the list of reserved words that you cannot use to name database objects. Although you can use nonreserved keywords as object names, doing so may result in possible confusion.

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.

Keywords

Other keywords appear throughout a statement as modifiers (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 to be identical:

```
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. Fullwidth letters are not valid regardless of the character set being used.

For a list of Teradata SQL keywords, see Appendix B: “Restricted Words.”

Reserved Words and Object Names

Note that you cannot use reserved words to name database objects. Because new reserved words are frequently added to new releases of Teradata Database, you may experience a problem with database object names that were valid in prior releases but then become nonvalid in a new release.

The workaround for this is to do one of the following things:

- Put the newly nonvalid name in double quotes.
- Rename the object.

In either case you must change your applications.

Expressions

Introduction

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

Scalar Expressions

A scalar expression, or value 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. For details, see “Zero-Table SELECT” on page 120.

Related Topics

For more information on ...	See ...
• CASE expresssions • arithmetic expressions • logical expressions • datetime expressions • interval expressions • character expresssions • byte expressions	*SQL Functions, Operators, Expressions, and Predicates.*
data type conversions	*SQL Functions, Operators, Expressions, and Predicates.*
query expressions	*SQL Data Manipulation Language.*

Names

Introduction

In Teradata SQL, various database objects such as tables, views, stored procedures, macros, columns, and collations are identified by a name.

The set of valid names depends on whether the system is enabled for Japanese language support.

Rules

The rules for naming Teradata Database database objects on systems enabled for standard language support are as follows.

- You must define and reference each *object*, such as user, database, or table, by a name.
- In general, names consist of 1 to 30 characters.
- Names can appear as a sequence of characters within double quotes, as a hexadecimal name literal, and as a Unicode delimited identifier. Such names have fewer restrictions on the characters that can be included. The restrictions are described in "QUOTATION MARKS Characters and Names" on page 80 and "Internal Hexadecimal Representation of a Name" on page 81.
- Unquoted names have the following syntactic restrictions:
  - They may only include the following characters:
    - Uppercase or lowercase letters (A to Z and a to z)
    - Digits (0 through 9)
    - The special characters DOLLAR SIGN ($), NUMBER SIGN (#), and LOW LINE (_)

    - They must not begin with a digit.
    - They must not be a reserved word.
- Systems that are enabled for Japanese language support allow more characters to be used for names, but determining the maximum number of characters allowed in a name becomes much more complex (see "Name Validation on Systems Enabled with Japanese Language Support" on page 83).
- Names that define databases and objects must observe the following rules.
    - Databases, users, and profiles must have unique names.
    - Tables, views, stored procedures, join or hash indexes, triggers, user-defined functions, or macros can take the same name as the database or user in which they are created, but cannot take the same name as another of these objects in the same database or user.
    - Roles can have the same name as a profile, table, column, view, macro, trigger, table function, user-defined function, external stored procedure, or stored procedure; however, role names must be unique among users and databases.
    - Table and view columns must have unique names.
    - Parameters defined for a macro or stored procedure must have unique names.
    - Secondary indexes on a table must have unique names.
    - Named constraints on a table must have unique names.
    - Secondary indexes and constraints can have the same name as the table they are associated with.
- CHECK constraints, REFERENCE constraints, and INDEX objects can also have *assigned* names. Names are optional for these objects.
- Names are not case-specific (see "Case Sensitivity and Names" on page 82).

QUOTATION MARKS Characters and Names

Enclosing names in QUOTATION MARKS characters (U+0022) greatly increases the valid set of characters for defining names.

Pad characters and special characters can also be included. For example, the following strings are both valid names.

- "Current Salary"
- "D'Augusta"

The QUOTATION MARKS characters are not part of the name, but they *are* required, if the name is not valid otherwise.

For example, these two names are identical, even though one is enclosed within QUOTATION MARKS characters.

- This_Name
- "This_Name"

Object names may only contain characters that are translatable to a specific subset of the UNICODE server character set. For a list of characters from the UNICODE server character

set that are valid in object names, see the following text files that are available on 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

The object name must not consist entirely of blank characters. In this context, a blank character is any of the following:

- NULL (U+0000)
- CHARACTER TABULATION (U+0009)
- LINE FEED (U+000A)
- LINE TABULATION (U+000B)
- FORM FEED (U+000C)
- CARRIAGE RETURN (U+000D)
- SPACE (U+0020)

All of the following examples are valid names.

- Employee
- job_title
- CURRENT_SALARY
- DeptNo
- Population_of_Los_Angeles
- Totaldollars
- "Table A"
- "Today's Date"

Note: If you use quoted names, the QUOTATION MARKS characters that delineate the names are *not* counted in the length of the name and are *not* stored in Dictionary tables used to track name usage.

If a Dictionary view is used to display such names, they are displayed without the double quote characters, and if the resulting names are used without adding double quotes, the likely outcome is an error report.

For example, "D'Augusta" might be the name of a column in the Dictionary view DBC.ColumnsV, and the HELP statements that return column names return the name as D'Augusta (without being enclosed in QUOTATION MARKS characters).

Internal Hexadecimal Representation of a Name

You can also create and reference object names by their internal hexadecimal representation in the Data Dictionary using hexadecimal name literals or Unicode delimited identifiers.

Object names are stored in the Data Dictionary using the UNICODE server character set.

For backward compatibility, object names are processed internally as LATIN strings on systems enabled with standard language support and as KANJI1 strings on systems enabled with Japanese language support.

Hexadecimal name literals and Unicode delimited identifiers are subject to the same restrictions as quoted names and may only contain characters that are translatable to a specific subset of the UNICODE server character set. For a list of characters from the UNICODE server character set that are valid in object names, see the following text files that are available on 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

Here is an example of a hexadecimal name literal that represents the name **TAB1** using fullwidth 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 '#';
```

The best practice is to use Unicode delimited identifiers because the hexadecimal values are the same across all supported character sets. The hexadecimal name representation using hexadecimal name literals varies depending on the character set.

For more information on the syntax and usage notes for hexadecimal name literals and Unicode delimited identifiers, see *SQL Data Types and Literals*.

Case Sensitivity and Names

Names are not case-dependent—a name cannot be used twice by changing its case. Any mix of uppercase and lowercase can be used when referencing symbolic names in a request.

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;
```

The case in which a column name is defined can be important. The column name is the default title of an output column, and symbolic names are returned in the same case in which they were defined.

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

```
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, such as the following example, the column names are returned exactly as defined they were defined, for example, *last_name*, then *first_name*.

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

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 statement.

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

Name Validation on Systems Enabled with Japanese Language Support

Introduction

A system that is enabled with Japanese language support allows thousands of additional characters to be used for names, but also introduces additional restrictions.

Rules

Unquoted names can use the following characters when Japanese language support is enabled:

- Any character valid in an unquoted name under standard language support:
  - Uppercase or lowercase letters (A to Z and a to z)
  - Digits (0 through 9)
  - The special characters DOLLAR SIGN ($), NUMBER SIGN (#), and LOW LINE (_)
- The fullwidth (zenkaku) versions of the characters valid for names under standard language support:
  - Fullwidth uppercase or lowercase letters (A to Z and a to z)
  - Fullwidth digits (0 through 9)
  - The special characters fullwidth DOLLAR SIGN ($), fullwidth NUMBER SIGN (#), and fullwidth LOW LINE (_)
- Fullwidth (zenkaku) and halfwidth (hankaku) Katakana characters and sound marks.
- Hiragana characters.
- Kanji characters from JIS-x0208.

The length of a name is restricted in a complex fashion. For details, see "Calculating the Length of a Name" on page 85.

Names cannot begin with a digit or a fullwidth digit.

Charts of the supported Japanese character sets, the Teradata Database internal encodings, the valid character ranges for Japanese object names and data, and the nonvalid character ranges for Japanese data and object names are documented in *International Character Set Support*.

Rules for Quoted Names and Internal Hexadecimal Representation of Names

As described in "QUOTATION MARKS Characters and Names" on page 80 and "Internal Hexadecimal Representation of a Name" on page 81, a name can also appear as a sequence of characters within double quotation marks, as a hexadecimal name literal, or as a Unicode delimited identifier. Such names have fewer restrictions on the characters that can be included.

Object names are stored in the Data Dictionary using the UNICODE server character set. For backward compatibility, object names are processed internally as KANJI1 strings on systems enabled with Japanese language support.

Object names may only contain characters that are translatable to a specific subset of the UNICODE server character set. For a list of characters from the UNICODE server character set that are valid in object names on systems enabled with Japanese language support, see the following text file that is available on CD and on the Web at http://www.info.teradata.com.

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

The list of valid UNICODE characters in object names applies to characters that translate from Japanese client character sets, predefined non-Japanese multibyte client character sets such as UTF8, UTF16, TCHBIG5_1R0, and HANGULKSC5601_2R4, and extended site-defined multibyte client character sets.

The object name must not consist entirely of blank characters. In this context, a blank character is any of the following:

- NULL (U+0000)
- CHARACTER TABULATION (U+0009)
- LINE FEED (U+000A)
- LINE TABULATION (U+000B)
- FORM FEED (U+000C)
- CARRIAGE RETURN (U+000D)
- SPACE (U+0020)

Cross-Platform Integrity

Because object names are stored in the Data Dictionary using the UNICODE server character set, you can access objects from heterogeneous clients. For example, consider a table name of ﾃｰﾌﾞﾙ 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
```

How Validation Occurs

Name validation occurs when the object is created or renamed, as follows:

- User names, database names, and account names are verified during the CREATE/MODIFY USER and CREATE/MODIFY DATABASE statements.
- Names of work tables and error tables are validated by the MultiLoad and FastLoad client utilities.
- Table names and column names are verified during the CREATE/ALTER TABLE and RENAME TABLE statements. View and macro names are verified during the CREATE/RENAME VIEW and CREATE/RENAME MACRO statements.

  Stored procedure and external stored procedure names are verified during the execution of CREATE/RENAME/REPLACE PROCEDURE statements.

  UDF names are verified during the execution of CREATE/RENAME/REPLACE FUNCTION statements.

  Hash index names are verified during the execution of CREATE HASH INDEX. Join index names are verified during the execution of CREATE JOIN INDEX.
- Alias object names used in the SELECT, UPDATE, and DELETE statements are verified. The validation occurs only when the SELECT statement is used in a CREATE/REPLACE VIEW statement, and when the SELECT, UPDATE, or DELETE TABLE statement is used in a CREATE/REPLACE MACRO statement.

Calculating the Length of a Name

The length of a name is measured by the physical bytes of its internal representation, not by the number of viewable characters. Under the KanjiEBCDIC character sets, the Shift-Out and Shift-In characters that delimit a multibyte character string are included in the byte count.

For example, the following table name contains six logical characters of mixed single byte characters/multibyte characters, defined during a KanjiEBCDIC session:

```
<TAB1>QR
```

All single byte characters, including the Shift-Out/Shift-In characters, are translated into the Teradata Database internal encoding, based on JIS-x0201. Under the KanjiEBCDIC character sets, all multibyte characters remain in the client encoding.

Thus, the processed name is a string of twelve bytes, padded on the right with the single byte space character to a total of 30 bytes.

The internal representation is as follows:

```
0E 42E3 42C1 42C2 42F1 0F 51 52 20 20 20 20 20 20 20 20 20 20 20 ...
<   T   A   B   1   > Q  R
```

To ensure upgrade compatibility, an object name created under one character set cannot exceed 30 bytes in *any* supported character set.

For example, a single Katakana character occupies 1 byte in KanjiShift-JIS. However, when KanjiShift-JIS is converted to KanjiEUC, each Katakana character occupies two bytes. Thus, a 30-byte Katakana name in KanjiShift-JIS expands in KanjiEUC to 60 bytes, which is illegal.

The formula for calculating the correct length of an object name is as follows:

```
Length = ASCII + (2*KANJI) + MAX (2*KATAKANA, (KATAKANA + S2M + M2S))
```

where:

This variable ...	Represents the number of ...
ASCII	single-byte ASCII characters in the name.
KANJI	double-byte characters in the name from the JIS-x0208 standard.
KATAKANA	single-byte Hankaku Katakana characters in the name.
S2M	transitions from ASCII or KATAKANA to JIS-x0208.
M2S	transitions from JIS-x0208 to ASCII or KATAKANA.

Examples of Validating Japanese Object Names

The following tables illustrate valid and nonvalid object names under the Japanese character sets: KanjiEBCDIC, KanjiEUC, and KanjiShift-JIS. The meanings of ASCII, KATAKANA, KANJI, S2M, M2S, and LEN are defined in "Calculating the Length of a Name" on page 85.

KanjiEBCDIC Object Name Examples

Name	ASCII	Kanji	Katakana	S2M	M2S	LEN	Result
<**ABCDEFGHIJKLMN**>	0	14	0	1	1	30	Valid.
<**ABCDEFGHIJ**>kl<**MNO**>	2	13	0	2	2	32	Not valid because LEN > 30.
<**ABCDEFGHIJ**>kl<>	2	10	0	2	2	26	Not valid because consecutive SO and SI characters are not allowed.
<**ABCDEFGHIJ**><**K**>	0	11	0	2	2	26	Not valid because consecutive SI and SO characters are not allowed.
ABCDEFGHIJKLMNO	0	0	15	0	0	30	Valid.
<**ABCDEFGHIJ**>**KLMNO**	0	10	5	1	1	30	Valid.
<Δ>	0	1	0	1	1	6	Not valid because the double byte space is not allowed.

KanjiEUC Object Name Examples

Name	ASCII	Kanji	Katakana	S2M	M2S	LEN	Result
ABCD**EFGH**IJKL**MOPQ**	6	10	0	3	3	32	Not valid because LEN > 30 bytes.
ABCDEFGHIJKL**M**	6	7	0	2	2	24	Valid.
ss_2**ABCDEFGHIJKL**	0	11	1	1	1	25	Valid.
Ass_2**BCDEFGHIJKLMN**	0	13	1	2	2	31	Not valid because LEN > 30 bytes.
ss_3**C**	0	0	0	1	1	2	Not valid because characters from code set 3 are not allowed.

KanjiShift-JIS Object Name Examples

Name	ASCII	Kanji	Katakana	S2M	M2S	LEN	Result
ABCDEFGHIJK**LMNOPQR**	6	5	7	1	1	30	Valid.
ABCD**EFG**HI**JK**L**MNOPQR**	6	5	7	4	4	31	Not valid because LEN > 30 bytes.

Related Topics

For charts of the supported Japanese character sets, the Teradata Database internal encodings, the valid character ranges for Japanese object names and data, and the nonvalid character ranges for Japanese data and object names, see *International Character Set Support*.

Object Name Translation and Storage

Object names are stored in the dictionary tables using the UNICODE server character set. For backward compatibility, object names on systems enabled with Japanese language support are processed internally as KANJI1 strings using the following translation conventions.

Character Type	Description
Single byte	All single byte characters in a name, including the KanjiEBCDIC Shift-Out/Shift-In characters, are translated into the Teradata Database internal representation (based on JIS-x0201 encoding).

<table>
<tr><th>Character Type</th><th>Description</th></tr>
<tr><td>Multibyte</td><td>Multibyte characters in object names are handled according to the character set in effect for the current session, as follows.
<table>
<tr><th>Multibyte Character Set</th><th>Description</th></tr>
<tr><td>KanjiEBCDIC</td><td>Each multibyte character within the Shift-Out/Shift-In delimiters is processed without translation; that is, it remains in the client encoding. The name string must have matched (but not consecutive) Shift-Out and Shift-In delimiters.</td></tr>
<tr><td>KanjiEUC</td><td>Under code set 1, each multibyte character is translated from KanjiEUC to KanjiShift-JIS. Under code set 2, byte ss_2 (0x8E) is translated to 0x80; the second byte is left unmodified.
This translation preserves the relative ordering of code set 0, code set 1, and code set 2.</td></tr>
<tr><td>KanjiShift-JIS</td><td>Each multibyte character is processed without translation; it remains in the client encoding.</td></tr>
</table>
For information on non-Japanese multibyte character sets, see International Character Set Support.</td></tr>
</table>

Object Name Comparisons

In comparing two names, the following rules apply:

- Names are case insensitive.

  A simple Latin lowercase letter is equivalent to its corresponding simple Latin uppercase letter. For example, 'a' is equivalent to 'A'.

  A fullwidth Latin lowercase letter is equivalent to its corresponding fullwidth Latin uppercase letter. For example, the fullwidth letter 'a' is equivalent to the fullwidth letter 'A'.
- Although different character sets have different physical encodings, multibyte characters that have the same logical presentation compare as equivalent.

  For example, the following strings illustrate the internal representation of two names, both of which were defined with the same logical multibyte characters. However, the first name was created under the KANJIEBCDIC5026_0I client character set, and the second name was created under KANJISJIS_0S.

```
KanjiEBCDIC:      0E 42E3 42C1 42C2 42F1 0F
KanjiShift-JIS:   8273 8260 8261 8250
```

Finding the Internal Hexadecimal Representation for Object Names

Introduction

The CHAR2HEXINT function converts a character string to its internal hexadecimal representation. You can use this function to find the internal representation of any Teradata 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: Names that can Form Hexadecimal Name Literals

Here is an example that shows how to find the internal representation of the Dbase table name in hexadecimal notation.

```
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';
```

Here is the result:

```
Internal Hex Representation                                  Name
------------------------------------------------------------ --------------
4462617365202020202020202020202020202020202020202020202020 Dbase
```

You can use the result of the CHAR2HEXINT function on an object name from the DBC.Tables view to form a hexadecimal name literal:

```
'4462617365'XN
```

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

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

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

Example 2: Names that can Form Unicode Delimited Identifiers

Here is an example that performs the same query as the preceding example on the DBC.TablesV view.

```
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';
```

Here is the result:

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

You can use the result of the CHAR2HEXINT function on an object name from the DBC.TablesV view to form a Unicode delimited identifier:

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

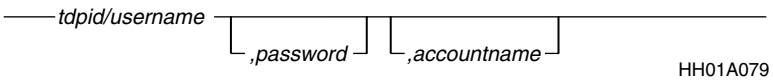
For more information on the DBC.TablesV view, see *Data Dictionary*.

Specifying Names in a Logon String

Purpose

Identifies a user to Teradata Database and, optionally, permits the user to specify a particular account to log onto.

Syntax



where:

Syntax element …	Specifies …
tdp_id/username	the client TDP the user wishes to use to communicate with Teradata Database and the name by which Teradata Database knows the user. The *username* parameter can contain mixed single byte and multibyte characters if the current character set permits them.
password	an optional (depending on how the user is defined) password required to gain access to Teradata Database. The *password* parameter can contain mixed single byte and multibyte characters if the current character set permits them.

Syntax element …	Specifies …
accountname	an optional account name or account string that specifies a user account or account and performance-related variable parameters the user can use to tailor the session being logged onto. The *accountname* parameter can contain mixed single byte and multibyte characters if the current character set permits them.

Teradata Database does not support the hexadecimal representation of a *username*, a *password*, or an *accountname* in a logon string.

For example, if you attempt to log on as user DBC by entering '444243'XN, the logon is not successful and an error message is generated.

Password Control Options

To control password security, site administrators can set password attributes such as the minimum and maximum number of characters allowed in the password string and the use of digits and special characters.

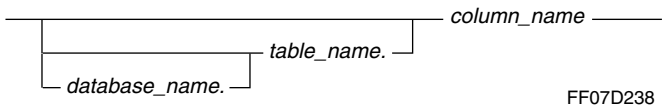
For information on managing passwords and password control options, see *Security Administration*.

Standard Form for Data in Teradata Database

Introduction

Data in Teradata Database is presented to a user according to the relational model, which models data as two dimensional tables with rows and columns. Each row of a table is composed one or more columns identified by column name. Each column contains a data item (or a null) having a single data type.

Syntax for Referencing a Column



where:

Syntax element ...	Specifies ...
database_name	a qualifying name for the database in which the table and column being referenced is stored. Depending on the ambiguity of the reference, *database_name* might or might not be required. See "Unqualified Object Names" on page 93.
table_name	a qualifying name for the table in which the column being referenced is stored. Depending on the ambiguity of the reference, *table_name* might or might not be required. See "Unqualified Object Names" on page 93.
column_name	one of the following: • The name of the column being referenced • The alias of the column being referenced • The keyword PARTITION See "Column Alias" on page 92.

Definition: Fully Qualified Column Name

A fully qualified name consists of a database name, table name, and column name.

For example, a fully qualified reference for the Name column in the Employee table of the Personnel database is:

```
Personnel.Employee.Name
```

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. For details, see "Names" on page 79.

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
```

Unqualified Object Names

Definition

An unqualified object name is an object such as a table, column, trigger, macro, or stored procedure reference that is not fully qualified. 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 "Default Database" on page 95)
- 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.

Rules for Name Resolution

The following rules govern name resolution:

- Name resolution is performed statement by statement.
- 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.
- An ambiguous unqualified name returns an error to the requestor.

Related Topics

For more information on …	See …
default databases	"Default Database" on page 95.

For more information on ...	See ...
the DATABASE statement	*SQL Data Definition Language*.
the CREATE USER statement	
the MODIFY USER statement	
the CREATE PROCEDURE statement and the SQL SECURITY clause	

Default Database

Definition

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, however; 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. • 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. • MODIFY USER, with a PROFILE clause. • MODIFY PROFILE, with a DEFAULT DATABASE clause.
add a default database when one had not been established previously	

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 as follows:

```
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*.
the CREATE USER statement	
the MODIFY USER statement	
fully-qualified names	"Standard Form for Data in Teradata Database" on page 91.
	"Unqualified Object Names" on page 93.
using profiles to define a default database	"Profiles" on page 70.

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 single quotation marks followed by the characters XI1 for a BYTEINT, XI2 for a SMALLINT, XI4 for an INTEGER, or XI8 for a BIGINT.

Type	Description
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 • YEAR TO MONTH • MONTH	Represent a time span that can include a number of years and months.
Day-Time	• DAY • DAY TO HOUR • DAY TO MINUTE • DAY TO SECOND • HOUR • HOUR TO MINUTE • HOUR TO SECOND • MINUTE • MINUTE TO SECOND • 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 single quotes. A zero-length character literal is represented by two consecutive single quotes ('').
Hexadecimal character literal	A hexadecimal character literal specifies a string of 0 to 62000 hexadecimal digits enclosed with single quotation marks 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 quoted string.

Object Name Literals

Hexadecimal name literals and Unicode delimited identifiers provide a way to create and reference object names by their internal representation in the Data Dictionary.

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_DATE
- CURRENT_ROLE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- CURRENT_USER
- DATABASE
- DATE
- PROFILE
- ROLE
- SESSION
- TIME
- USER

Related Topics

For more information on ...	See ...
numeric, DateTime, interval, character, graphic, period, object name and hexadecimal literals	*SQL Data Types and Literals*.
built-in functions	*SQL Functions, Operators, Expressions, and Predicates*.

NULL Keyword as a Literal

Null

A *null* represents any of three things:

- 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 SQL:2008-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 148.

Operators

Introduction

SQL operators are used to express logical and arithmetic operations. Operators of the same precedence are evaluated from left to right. See "SQL Operations and Precedence" on page 103 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.

Definitions

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
lowest	logical	logical OR logical

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.

Related Topics

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

Delimiters

Introduction

Delimiters are special characters having meanings that depend on context.

The function of each delimiter appears in the following table.

Delimiter	Name	Purpose
( )	LEFT PARENTHESIS 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. 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*. • Separates table names from a particular column name, such as *employee.deptno*). • In numeric constants, the period is the decimal point. • Separates DateTime fields in a DateTime type. • Separates a method name from a UDT expression in a method invocation.

Delimiter	Name	Purpose
;	SEMICOLON	• Separates statements in multi-statement requests. • Separates statements in a stored procedure body. • Separates SQL procedure statements in a triggered SQL statement in a trigger definition. • Terminates requests submitted via utilities such as BTEQ. • Terminates embedded SQL statements in C or PL/I applications.
’	APOSTROPHE	• Defines the boundaries of character string constants. • To include an APOSTROPHE character or show possession in a title, double the APOSTROPHE characters. • Also separates DateTime fields in a DateTime type.
“	QUOTATION MARK	Defines the boundaries of nonstandard names.
/	SOLIDUS	Separates DateTime fields in a DateTime type.
B b	Uppercase B 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.

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 "Comments" on page 106.
- 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 SQL:2008 standard.

Comments

Introduction

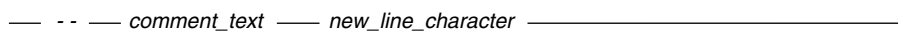
You can embed comments within an SQL request anywhere a pad character can occur.

The SQL parser and the preprocessor recognize the following types of ANSI SQL:2008-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.



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 */.

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

Definition

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 SQL:2008 standard.

Example

For 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 ; ) ;
```

Null Statements

Introduction

A *null statement* is a statement that 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;
```

CHAPTER 3 SQL Data Definition, Control, and Manipulation

This chapter describes the functional families of the SQL language.

SQL Functional Families and Binding Styles

Introduction

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.

Definition: 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.

Definition: Binding Style

The ANSI 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	• "Embedded SQL" on page 112 • *Teradata Preprocessor2 for Embedded SQL Programmer Guide* • *SQL Stored Procedures and Embedded SQL*
stored procedures	• "Stored Procedures" on page 65 • *SQL Stored Procedures and Embedded SQL*
ODBC	*ODBC Driver for Teradata User Guide*
JDBC	*Teradata JDBC Driver User Guide*

Embedded SQL

You can execute SQL statements from within client application programs. The expression embedded SQL refers to SQL statements executed or declared from within a client application.

An embedded Teradata SQL client program consists of the following:

- Client programming language statements
- One or more embedded SQL statements
- Depending on the host language, one or more embedded SQL declare sections

  SQL declare sections are optional in COBOL and PL/I, but *must* be used in C.

A special prefix, EXEC SQL, distinguishes the SQL language statements embedded into the application program from the host programming language.

Embedded SQL statements must follow the rules of the host programming language concerning statement continuation and termination, construction of variable names, and so forth. Aside from these rules, embedded SQL is host language-independent.

Details of Teradata Database support for embedded SQL are described in *SQL Stored Procedures and Embedded SQL*.

Data Definition Language

Definition

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 of Data Definition Language Statements

Data definition language statements perform the following functions:

- Create, drop, rename, and alter tables
- Create, drop, rename, and replace stored procedures, user-defined functions, views, and macros
- Create, drop, and alter user-defined types
- Create, drop, and replace user-defined methods
- Create and drop indexes
- Create, drop, and modify users and databases
- Create, drop, alter, rename, and replace triggers
- Create, drop, and set roles
- Create, drop, and modify profiles
- Collect statistics on a column set or index
- Establish a default database
- Comment on database objects
- Set a different collation sequence, account priority, DateForm, time zone, and database for the session
- 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
- Create, drop, and alter replication groups for which Teradata Replication Solutions captures data changes
- Create, drop, and replace rule sets associated with replication groups

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 Data Definition Statements

DDL statements include the following:

- ALTER FUNCTION
- ALTER METHOD
- ALTER PROCEDURE
- ALTER REPLICATION GROUP
- ALTER TABLE
- ALTER TRIGGER
- ALTER TYPE
- BEGIN LOGGING
- COMMENT
- CREATE AUTHORIZATION
- CREATE CAST
- CREATE DATABASE
- CREATE ERROR TABLE
- CREATE FUNCTION
- CREATE GLOP SET
- CREATE HASH INDEX
- CREATE INDEX
- CREATE JOIN INDEX
- CREATE MACRO
- CREATE METHOD
- CREATE ORDERING
- CREATE PROCEDURE
- CREATE PROFILE
- CREATE REPLICATION GROUP
- CREATE REPLICATION RULESET
- CREATE ROLE
- CREATE TABLE
- CREATE TRANSFORM
- CREATE TRIGGER
- CREATE TYPE
- CREATE USER
- CREATE VIEW
- DATABASE
- DELETE DATABASE
- DELETE USER
- DROP AUTHORIZATION
- DROP CAST
- DROP DATABASE
- DROP ERROR TABLE
- DROP FUNCTION
- DROP GLOP SET
- DROP HASH INDEX
- DROP INDEX
- DROP JOIN INDEX
- DROP MACRO
- DROP ORDERING
- DROP PROCEDURE
- DROP PROFILE
- DROP REPLICATION GROUP
- DROP REPLICATION RULESET
- DROP ROLE
- DROP TABLE
- DROP TRANSFORM
- DROP TRIGGER
- DROP TYPE
- DROP USER
- DROP VIEW
- END LOGGING
- LOGGING ONLINE ARCHIVE ON/OFF
- MODIFY DATABASE
- MODIFY PROFILE
- MODIFY USER
- RENAME FUNCTION
- RENAME MACRO
- RENAME PROCEDURE
- RENAME TABLE
- RENAME TRIGGER
- RENAME VIEW
- REPLACE CAST
- REPLACE FUNCTION
- REPLACE MACRO
- REPLACE METHOD
- REPLACE ORDERING
- REPLACE PROCEDURE
- REPLACE REPLICATION RULESET

(DDL statements, *continued*)

- REPLACE TRANSFORM
- REPLACE TRIGGER
- REPLACE VIEW
- SET QUERY_BAND
- SET ROLE
- SET SESSION
- SET TIME ZONE

Related Topics

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

Introduction

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.

How to Make Changes

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
- 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.

To drop this type of database object ...	Use this SQL statement ...
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
Table	DROP TABLE
Global temporary table or volatile table	
Primary index	
Trigger	DROP TRIGGER

To drop this type of database object …	Use this SQL statement …
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.

To rename this type of database 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
View	RENAME VIEW

Related Topics

For further information on these statements, including rules that apply to usage, see *SQL Data Definition Language*.

Data Control Language

Definition

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 of Data Control Statements

Data control statements perform the following functions:

- Grant and revoke privileges
- Give ownership of a database to another user

Rules on Entering Data Control 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 Data Control Statements

Data control statements include the following:

- GIVE
- GRANT
- GRANT CONNECT THROUGH
- GRANT LOGON
- REVOKE
- REVOKE CONNECT THROUGH
- REVOKE LOGON

Related Topics

For detailed information about the function, syntax, and usage of Teradata SQL Data Control statements, see *SQL Data Control Language*.

Data Manipulation Language

Definition

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. 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 • for any unique index • 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: • table is defined as SET • 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: • NOT NULL is specified • 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. Alternatively, use the MERGE statement.

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.

To delete rows in a multirow result table in embedded SQL, use the following process:

1. Declare a cursor for the SELECT statement.
2. Fetch individual result rows for processing using the cursor you declared.
3. Use a WHERE CURRENT OF clause in a positioned DELETE statement to delete the selected rows.

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

Introduction

Subqueries are nested SELECT statements. They can be used to ask a series of questions to arrive at a single answer.

Three Level Subqueries: Example

The following subqueries, nested to three levels, are used to 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

Introduction

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.

Two Ways to Specify a Recursive Query

You can specify a recursive query by:

- Preceding a query with the WITH RECURSIVE clause
- Creating a view using the RECURSIVE clause in a CREATE VIEW statement

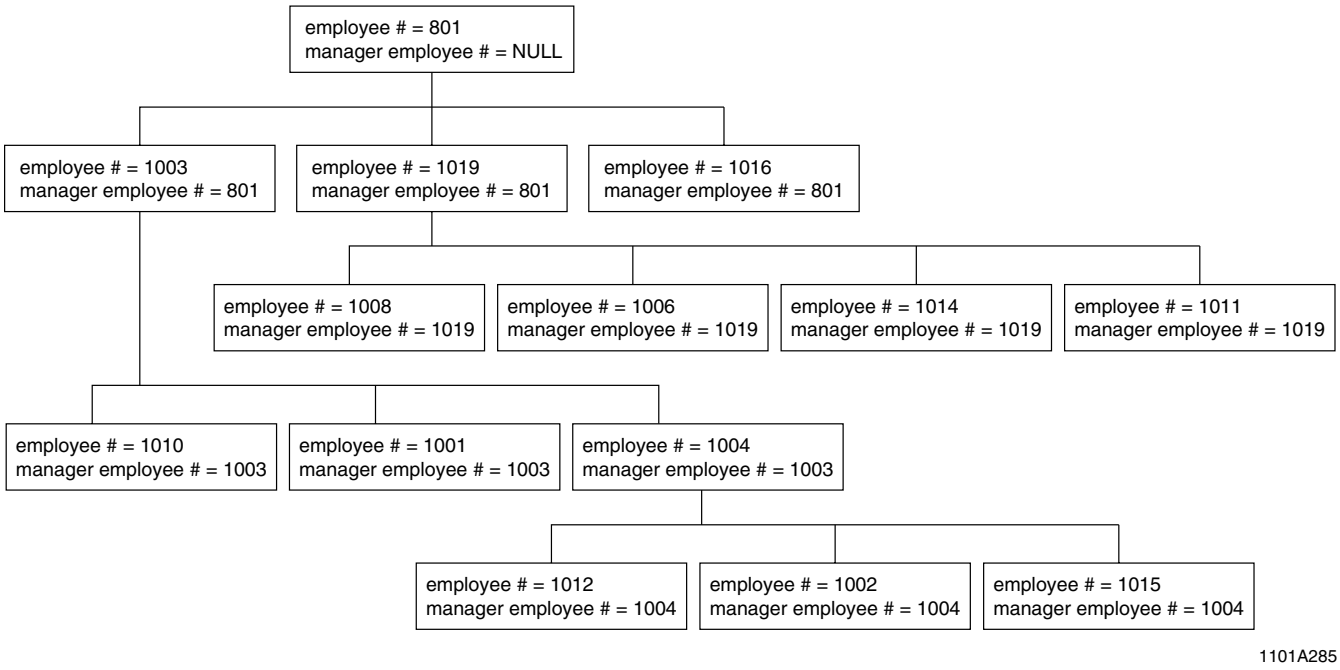
Using the WITH RECURSIVE Clause

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.



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 124. 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 as follows:

- 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 reengineer 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 andsample-based components of the target level emulation facility used to emulate a production environment on a test system:

- DIAGNOSTIC HELP PROFILE
- DIAGNOSTIC SET PROFILE
- DIAGNOSTIC COSTPRINT
- DIAGNOSTIC DUMP COSTS
- DIAGNOSTIC HELP COSTS
- DIAGNOSTIC SET COSTS
- DIAGNOSTIC DUMP SAMPLES
- DIAGNOSTIC HELP SAMPLES
- DIAGNOSTIC SET SAMPLES
- DIAGNOSTIC “Validate Index”

After configuring the test environment and enabling it with the appropriate production system statistical and demographic data, you can perform various workload analyses to determine optimal sets of secondary indexes to support those workloads in the production environment.

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

Introduction

Teradata SQL provides several powerful tools to get help about database object definitions and summaries of database object definition statement text.

HELP Statements

The various HELP statements return reports about the current column definitions for named database objects. The reports returned by these statements 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.

IF you want to get ...	THEN use ...
the attributes of a column, including whether it is a single-column primary or secondary index and, if so, whether it is unique	HELP COLUMN
the attributes for a specific named constraint on a table	HELP CONSTRAINT
the attributes, sorted by object name, for all tables, views, join and hash indexes, stored procedures, user-defined functions, and macros in a specified database	HELP DATABASE and HELP USER
column information of the error logging table for a specified data table	HELP ERROR TABLE
the specific function name, list of parameters, data types of the parameters, and any comments associated with the parameters of a user-defined function	HELP FUNCTION
the data types of the columns defined by a particular hash index	HELP HASH INDEX
the attributes for the indexes defined for a table or join index	HELP INDEX
the attributes of the columns defined by a particular join index	HELP JOIN INDEX
the attributes for the specified macro	HELP MACRO
the specific name, list of parameters, data types of the parameters, and any comments associated with the parameters of a user-defined method	HELP METHOD
the attributes for the specified join index or table	HELP TABLE
the attribute and format parameters for each parameter of the procedure or just the creation time attributes for the specified procedure	HELP PROCEDURE

IF you want to get …	THEN use …
the attributes of the specified replication group and its member tables	HELP REPLICATION GROUP
the attributes for the specified trigger	HELP TRIGGER
information on the type, attributes, methods, cast, ordering, and transform of the specified user-defined type	HELP TYPE
the attributes for a specified view	HELP VIEW
the attributes for the requested volatile table	HELP VOLATILE TABLE

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.

IF you want to get the data definition statement most recently used to create, replace, or modify a specified …	THEN use …
error logging table	SHOW ERROR TABLE
	SHOW TABLE
hash index	SHOW HASH INDEX
join index	SHOW JOIN INDEX
macro	SHOW MACRO
stored procedure or external stored procedure	SHOW PROCEDURE
table	SHOW TABLE
trigger	SHOW TRIGGER
user-defined function	SHOW FUNCTION
user-defined method	SHOW METHOD
user-defined type	SHOW TYPE
view	SHOW VIEW

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 );
```

Related Topics

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

CHAPTER 4 SQL Data Handling

This chapter describes the fundamentals of Teradata Database data handling.

Invoking SQL Statements

Introduction

One of the primary issues that motivated the development of relational database management systems was the perceived need to create database management systems that could be queried not just by predetermined, hard-coded requests but also interactively by well-formulated *ad hoc* queries.

SQL addresses this issue by offering several ways to invoke an executable 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 is one that 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 are the following:

- SELECT
- CREATE TABLE
- COMMIT
- CONNECT
- PREPARE

Most, but not all, executable SQL statements can be performed interactively from a terminal using an SQL query manager like BTEQ or Teradata SQL Assistant (formerly called Queryman).

Types of executable SQL commands that cannot be performed interactively are the following:

- 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.

Some 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

Requests

Introduction

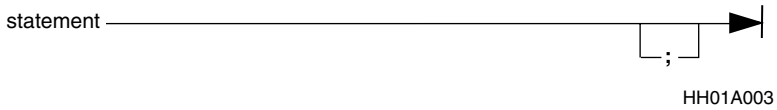
A *request* to Teradata Database consists of one or more SQL *statements* and can span any number of input lines. Teradata Database can receive and perform SQL *statements* that are:

- 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.

Single Statement Requests

A single statement request consists of a statement keyword followed by one or more expressions, other keywords, clauses, and phrases. A single statement request is treated as a solitary unit of work.

Here is the syntax for a single statement request:



Multistatement Requests

A *multistatement request* consists of two or more statements separated by SEMICOLON characters.

Multistatement requests are non-ANSI standard.

Here is the syntax for a multistatement request:

For more information, see "Multistatement Requests" on page 138.

Iterated Requests

An *iterated request* is a single DML statement with multiple data records.

Iterated requests do not directly impact the syntax of SQL statements. They provide a more performant way of processing DML statements that specify the USING request modifier to import or export data from Teradata Database.

For more information, see "Iterated Requests" on page 141.

ANSI Session Mode

If an error is found in a request, then that request is aborted. Normally, the entire transaction is not aborted; however, some failures will abort the entire transaction.

Teradata Session Mode

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 the request, then the entire transaction is aborted.

Abort processing proceeds as follows:

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.

Complete Requests

A request is considered *complete* 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.

Transactions

Introduction

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
- Teradata

In ANSI session mode, transaction processing adheres to the rules defined by the ANSI SQL specification. In Teradata session mode, transaction processing follows the rules defined by Teradata prior to the emergence of the ANSI 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

Related Topics

The next few pages highlight some of the differences between transaction processing in ANSI session mode and transaction processing in Teradata session mode.

For detailed information on statement and transaction processing, see *SQL Request and Transaction Processing*.

Transaction Processing in ANSI Session Mode

Introduction

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.

Note that DATABASE and SET SESSION are DDL statements. See “Rollback Processing” in *SQL Request and Transaction Processing*.

If a session terminates with an open transaction, then any effects of that transaction are rolled back.

Two-Phase Commit (2PC)

Sessions in ANSI session mode do not support 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.

Transaction Processing in Teradata Session Mode

Introduction

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 one of the following:

- A single DML statement that affects one or more rows of one or more tables
- A macro or trigger containing one or more statements
- A request containing multiple statements separated by SEMICOLON characters. Each SEMICOLON character can appear anywhere in the input line. The Parser interprets a SEMICOLON character at the end of an input line as a 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.

Two-Phase Commit (2PC) Rules

Two-phase commit (2PC) protocol 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.

Multistatement Requests

Definition

An atomic request containing more than one SQL statement, each terminated by a SEMICOLON character.

Syntax

HH01A004

ANSI Compliance

Multistatement requests are non-ANSI SQL:2008 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, an implicit transaction can consist of a multistatement request. The outcome of an implicit multistatement transaction is all or nothing. If one statement in the request fails, the entire implicit transaction fails and the system rolls it back.

Parallel Step Processing

Teradata Database can perform some requests in parallel (see “Parallel Steps” on page 140). 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.

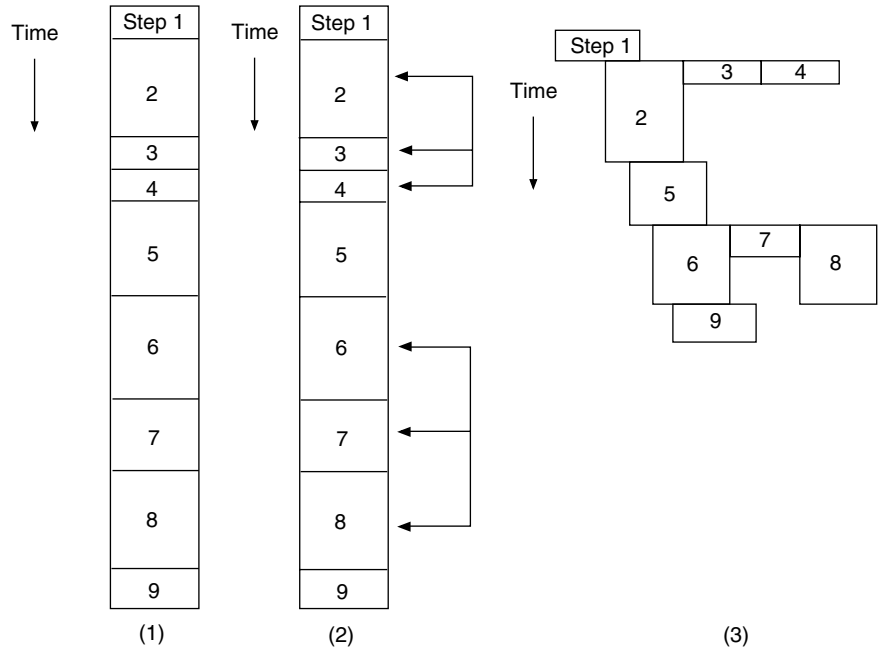
For example, if an INSERT step and a DELETE step are allowed to run in parallel, the AMP informs the PE that the DELETE step has progressed to the point where the INSERT step will not impact it adversely. This handshaking protocol also reduces the chance of a deadlock.

"Parallel Steps" on page 140 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



Iterated Requests

Definition

A single DML statement with multiple data records.

Usage

An iterated request is an atomic request consisting of a single SQL DML statement with multiple sets (records) of data.

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 "Rules" on page 142.

The following tools and interfaces provide facilities that you can use to execute iterated requests.

Tool/Interface	Facility
CLIv2 for network-attached systems	*using_data_count* field in the DBCAREA data area
CLIv2 for channel-attached systems	*Using-data-count* field in the DBCAREA data area
ODBC	Parameter arrays
JDBC type 4 driver	Batch operations
OLE DB Provider for Teradata	Parameter sets
BTEQ	• .REPEAT command • .SET PACK command

Rules

The following rules apply to iterated requests:

- 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 note that 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.

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 Channel-Attached Systems* • *Teradata Call-Level Interface Version 2 Reference for Network-Attached Systems*
ODBC parameter arrays	*ODBC Driver for Teradata User Guide*
JDBC driver batch operations	*Teradata JDBC Driver User Guide*
OLE DB Provider for Teradata parameter sets	*OLE DB Provider for Teradata Installation and User Guide*
BTEQ PACK command	*Basic Teradata Query Reference*

Dynamic and Static SQL

Definitions

Term	Definition
Dynamic 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	Static SQL is, by default, any method of invoking an SQL statement that is not dynamic.

ANSI Compliance

Dynamic SQL is ANSI SQL:2008-compliant.

The ANSI 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.

This description applies directly to all executable SQL statements except SELECT, which requires additional handling.

Note that SELECT INTO *cannot* be invoked dynamically.

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

Overview

The way stored procedures support dynamic SQL statements is different from the way embedded SQL does.

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 (n^{th} row in the SQLVAR(n) array)
column data type of n^{th} value	
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*

Note that in step 6, 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

Introduction

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*
SELECT AND CONSUME	*SQL Data Manipulation Language*

Manipulating Nulls

Introduction

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 101 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.

Result of Expressions That Contain Nulls

Here are some general 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 non-null argument.

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
```

Note that 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 150 and "Excluding Nulls" on page 149.

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.

Nulls and CASE Expressions

The following rules apply to 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 Non-Nulls Together

To search for nulls and non-nulls 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 Value in a Collation

When you use an ORDER BY clause to sort records, Teradata Database sorts null as the lowest value. Sorting nulls can vary from RDBMS to RDBMS. Other systems may sort null as the 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 non-null values 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

Value of First Column in Index	Value of Second Column in Index
null	null

An attempt to insert a row that matches any of these rows will result in a uniqueness violation.

Teradata Database Replaces 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*) DATE TIME TIMESTAMP 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 INTEGER SMALLINT BYTEINT FLOAT DECIMAL REAL DOUBLE PRECISION NUMERIC	0

The substitute values returned for nulls are not, by themselves, distinguishable from valid non-null values. 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. Note that 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(*) *does* include 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

Introduction

The following session parameters can be controlled with keywords or predefined system variables.

Parameter	Valid Keywords or System Variables
SQL Flagger	ON
	OFF
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: • $R • $H • $M • $L
Date Form	ANSIDATE
	INTEGERDATE
Character Set	Indicates the character set being used by the client. You can view site-installed client character sets from DBC.CharSetsV or DBC.CharTranslationsV. The following client character sets are permanently enabled: • ASCII • EBCDIC • UTF8 • UTF16 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 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
Entry level	SQLFLAG ENTRY
Intermediate level	SQLFLAG INTERMEDIATE

For more detail on using the SQL Flagger, see "SQL Flagger" on page 227.

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. Your system administrator can redefine this collation to any two-level collation of characters in the LATIN repertoire. For backward compatibility, the following are true: • MULTINATIONAL collation of KANJI1 data is single level. • The system administrator can redefine single byte character collation. This definition is not compatible with MULTINATIONAL collation of non-KANJI1 data. CHARSET_COLL collation is usually a better solution for KANJI1 data. 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 as follows: • EBCDIC collation for channel-connected systems. • 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. CHARSET_COLL collation is a system administrator-defined collation.

Collation Name	Description
JIS_COLL	Character data is collated based on the Japanese Industrial Standards (JIS). JIS characters collate in the following order: 1 JIS X 0201-defined characters in standard order 2 JIS X 0208-defined characters in standard order 3 JIS X 0212-defined characters in standard order 4 KanjiEBCDIC-defined characters *not* defined in JIS X 0201, JIS X 0208, or JIS X 0212 in standard order 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 that 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*.

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, UTF8, UTF16, 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

Introduction

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

For channel-connected applications, you can establish *session pools*, which are collections of sessions that are logged on to Teradata Database in advance (generally at the time of TDP initialization) for use by applications that require a 'fast path' logon. This capability is particularly advantageous for transaction processing in which interaction with Teradata Database consists of many single, short transactions.

TDP identifies each session with a unique session number. Teradata Database identifies a session with a session number, the username of the initiating user, and the logical host identification number of the connection (LAN or mainframe channel) associated with the controlling TDP or mTDP.

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 further 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, where all sessions use the same Teradata Database user, can use trusted sessions, where multi-tier applications can assert user identities and roles to manage access rights and audit queries.

For details on trusted sessions, see *Database 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

Introduction

SQL return codes provide information about the status of a completed executable SQL DML statement.

Status Variables for Receiving SQL Return Codes

ANSI SQL defines two status variables for receiving return codes:

- SQLSTATE
- SQLCODE

SQLCODE is not ANSI SQL-compliant. The ANSI SQL-92 standard explicitly deprecates SQLCODE, and the ANSI SQL-99 standard does not define SQLCODE. The ANSI 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 SQL Communications Area (SQLCA) that also has a field named SQLCODE for receiving return codes.

For information on ...	See ...
• SQLSTATE • SQLCODE • 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 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 165 and "Error Response (ANSI Session Mode Only)" on page 163.

For a complete list of exception condition codes, see the *Messages* book.

Completion Conditions

A completion condition indicates statement success.

There are three categories of completion conditions:

- Successful completion
- Warnings
- No data found

For more information, see:

- "Statement Responses" on page 161
- "Success Response" on page 162
- "Warning Response" on page 163

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.

For this type of completion condition ...	The value for this return code is ...	
	SQLSTATE	SQLCODE
Success	'00000'	0
Warning	'01901'	901
	'01800' to '01841'	901
	'01004'	902
No data found	'02000'	100

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.

For this type of condition ...	The value for this return code is ...	
	SQLSTATE	SQLCODE
Successful completion	'00000'	0
Warning	SQLSTATE value corresponding to the warning code.	the Teradata Database warning code.
No data found or any other Exception	SQLSTATE value corresponding to the error code.	the 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.

IF the statement raises this type of condition ...	THEN the application or condition handler takes the following remedial action ...
Successful completion	none.
Warning	the statement execution continues. If a warning condition handler is defined in the application, the handler executes.

IF the statement raises this type of condition ...	THEN the application or condition handler takes the following remedial action ...
No data found or any other exception	whatever appropriate action is required by the exception. If an EXIT handler has been defined for the exception, the statement execution terminates. If a CONTINUE handler has been 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.

How a Response Is Returned to the User

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

Definition

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
- 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

Definition

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)

Definition

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

Definition

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.

Query Processing

Introduction

An SQL query (the definition for "query" here 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. 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. 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). 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. 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

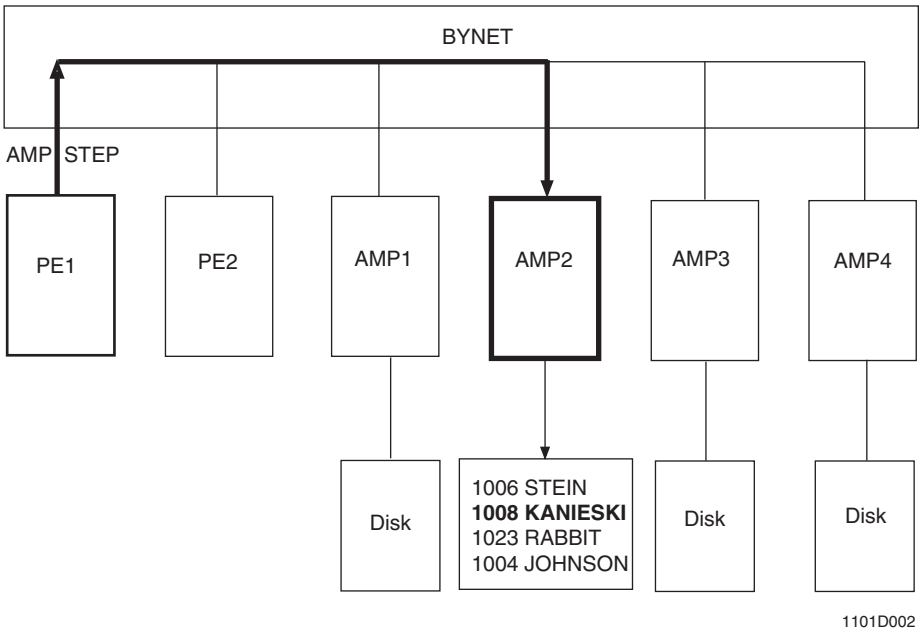
Assume that a PE receives the following SELECT statement:

```
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).

This process is illustrated by the graphic under “Flow Diagram of a Single AMP Request” on page 169. Only one BYNET is shown to simplify the illustration.

Flow Diagram of a Single AMP Request



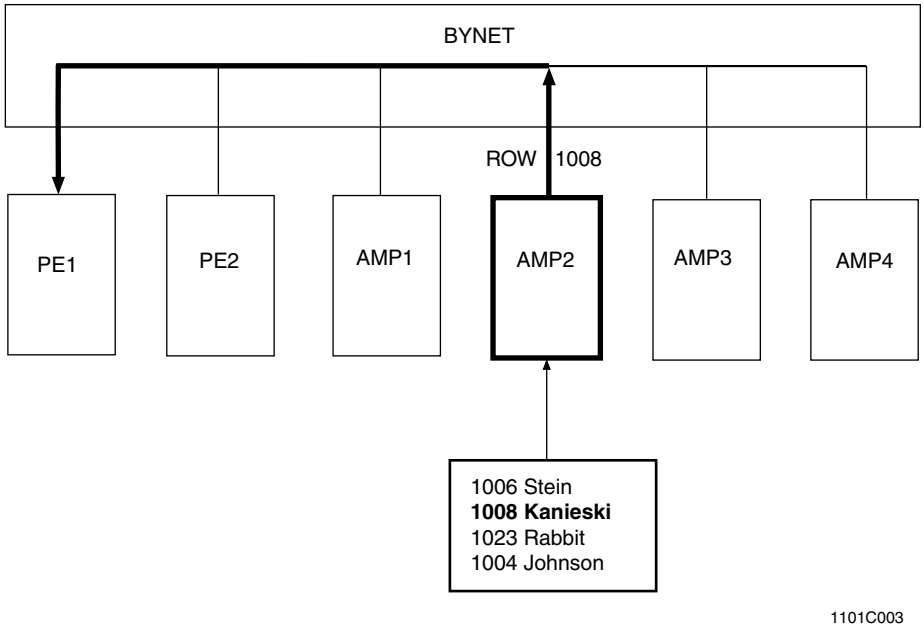
Assuming that AMP2 has the row, it accepts the message.

As illustrated by the graphic under “Single AMP Response to Requesting PE” on page 170, 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.

For an illustration of a single AMP request with partition elimination, see “Single AMP Request With Partition Elimination” on page 174.

Single AMP Response to Requesting PE



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.

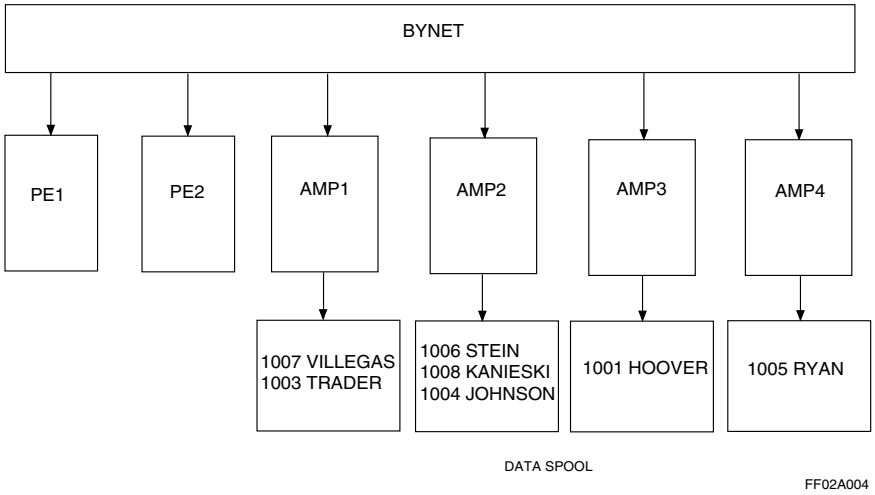
PE1 first parses the request and creates the following AMP steps:

- 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 by “Figure 1: Flow Diagram for an All AMP Request” on page 171.

Figure 1: Flow Diagram for an All AMP Request



The process is as follows:

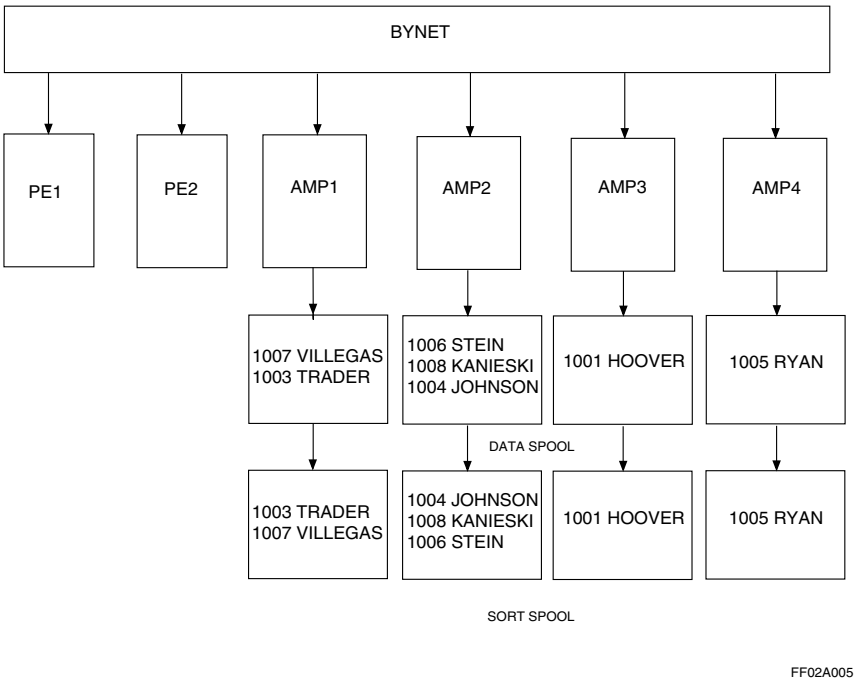
1. All AMPs accept the message, but the PEs do not.
2. Each AMP checks for qualifying rows on its disk storage units.
3. If any qualifying rows are found, the data in the requested columns is converted to the client format and copied to a spool file.
4. Each AMP completes the step, whether rows were 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* (see "Figure 2: Flow Diagram for an AMP Sort" on page 172).

   Note: If partitioned on employee_number, the scan may be limited to a few partitions based on partition elimination.

Figure 2: Flow Diagram for an AMP Sort



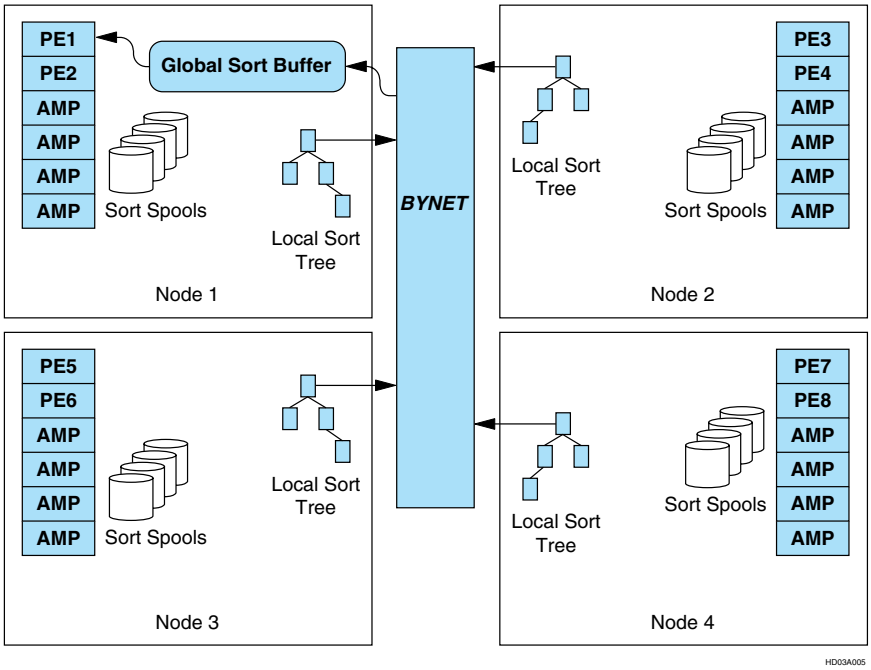
6 Each AMP sorts its answer set, then puts a completion message on the BYNET.

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 "Figure 3: Flow Diagram for a BYNET Merge" on page 173).

Figure 3: Flow Diagram for a BYNET Merge



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.

Type	Description
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.

<table>
<tr><th>IF a query ...</th><th>THEN ...</th></tr>
<tr><td>specifies values for all the primary index columns</td><td>the AMP where the rows reside can be determined and only a single AMP is accessed.
<table>
<tr><th>IF conditions are ...</th><th>THEN ...</th></tr>
<tr><td>not specified on the partitioning columns</td><td>each partition can be probed to find the rows based on the hash value.</td></tr>
<tr><td>also specified on the partitioning columns</td><td>partition elimination may reduce the number of partitions to be probed on that AMP.</td></tr>
</table>
For an illustration, see “Single AMP Request With Partition Elimination” on page 174.</td></tr>
<tr><td>does not specify the values for all the primary index columns</td><td>an all-AMP full file scan is required for a table with an NPPI.
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.</td></tr>
</table>

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.

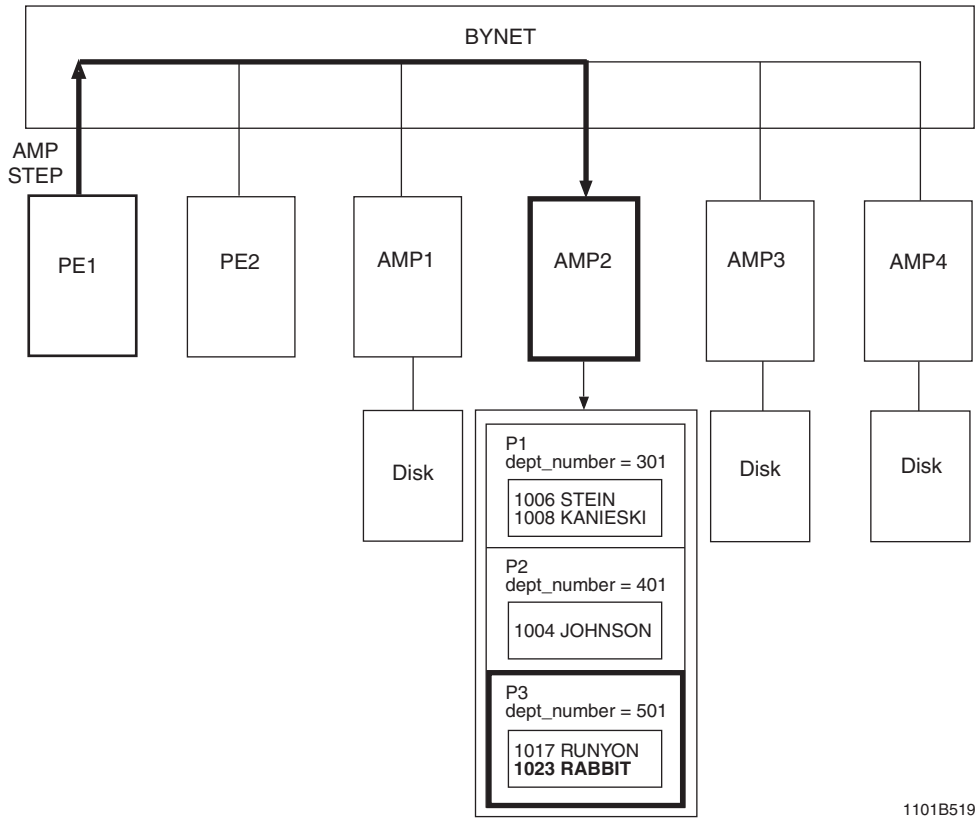
If conditions are also specified on the partitioning columns, partition elimination may reduce the number of partitions to be probed on that AMP.

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 diagram illustrates this process.



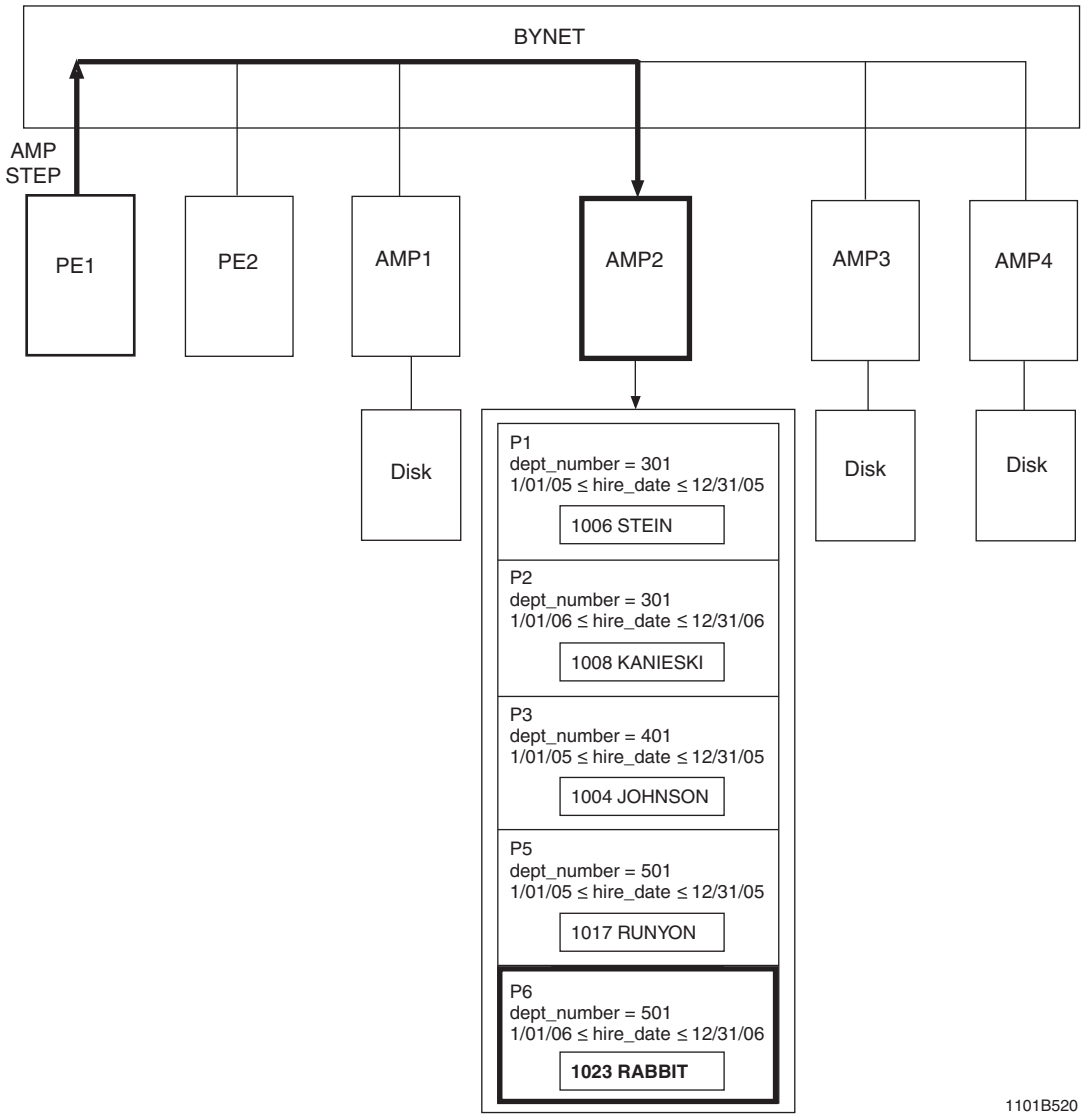
The AMP Step includes the list of partitions (in this case, P3) to access. Partition elimination (in this case, static partition elimination) reduces access to the partitions that satisfy the query requirements. In each partition in the list (in this case, only P3), look for rows with a given row hash 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 diagram illustrates this process.



(Note that 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 as follows:

- 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.

  Note that 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" on page 179
index access	"Indexes" on page 33

Full-Table Scans

Introduction

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.

How a Full-Table Scan Accesses Rows

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 file system tables known as the Master Index and Cylinder Index 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 are used to describe the SQL syntax and code:

- Syntax diagrams, used to describe SQL syntax form, including options. See "Syntax Diagram Conventions" on page 187.
- 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,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 Used In This Book" on page 188.

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. Do not use commas when typing a number with more than 3 digits.

Item	Definition / Comments
Word	Keywords and variables. • UPPERCASE LETTERS represent a keyword. Syntax diagrams show all keywords in uppercase, unless operating system restrictions require them to be in lowercase. • lowercase letters represent a keyword that you must type in lowercase, such as a UNIX command. • *lowercase italic letters* represent a variable such as a column or table name. Substitute the variable with a proper value. • **lowercase bold letters** represent an excerpt from the diagram. The excerpt is defined immediately following the diagram that contains it. • 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.

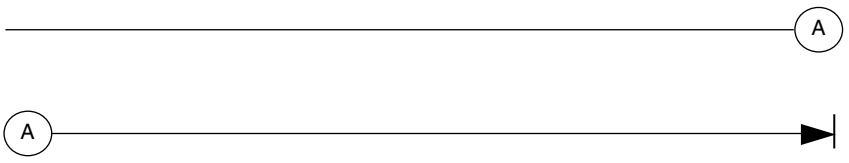
Paths

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.

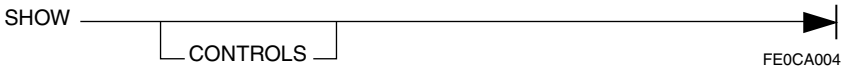
Required Entries

Required entries appear on the main path:

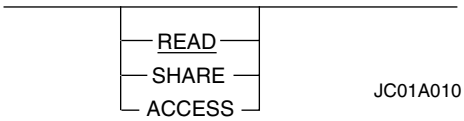
If you can choose from more than one entry, the choices appear vertically, in a stack. The first entry appears on the main path:

Optional Entries

You may choose to include or disregard optional entries. Optional entries appear below the main path:



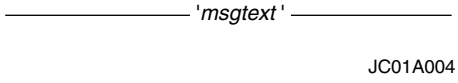
If you can optionally choose from more than one entry, all the choices appear below the main path:



Some commands and statements treat one of the optional choices as a default value. This value is UNDERLINED. It is presumed to be selected if you type the command or statement without specifying one of the options.

Strings

String literals appear in single quotes:



Abbreviations

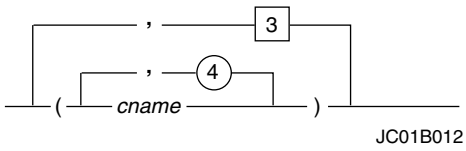
If a keyword or a reserved word has a valid abbreviation, the unabbreviated form always appears on the main path. The shortest valid abbreviation appears beneath.

In the above syntax, the following formats are valid:

- SHOW CONTROLS
- SHOW CONTROL

Loops

A loop is an entry or a group of entries that you can repeat one or more times. Syntax diagrams show loops as a return path above the main path, over the item or items that you can repeat:



Read loops from right to left.

The following conventions apply to loops:

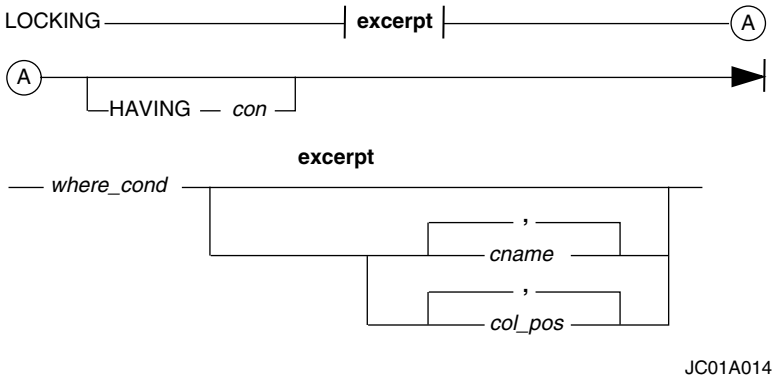
IF...	THEN...
there is a maximum number of entries allowed	the number appears in a circle on the return path. In the example, you may type *cname* a maximum of 4 times.
there is a minimum number of entries required	the number appears in a square on the return path. In the example, you must type at least three groups of column names.
a separator character is required between entries	the character appears on the return path. If the diagram does not show a separator character, use one blank space. 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

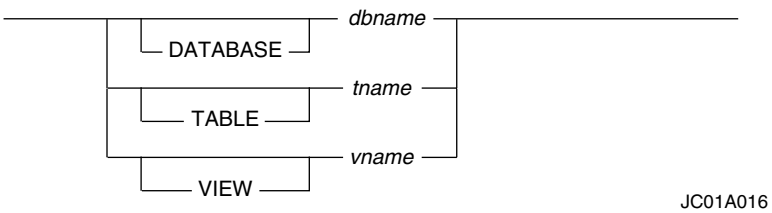
Sometimes a piece of a syntax phrase is too large to fit into the diagram. Such a phrase is indicated by a break in the path, marked by (|) terminators on each side of the break. The name for the excerpted piece appears between the terminators in boldface type.

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:



Multiple Legitimate Phrases

In a syntax diagram, it is possible for any number of phrases to be legitimate:



In this example, any of the following phrases are legitimate:

- *dbname*
- DATABASE *dbname*
- *tname*

- TABLE *tname*
- *vname*
- VIEW *vname*

Sample Syntax Diagram

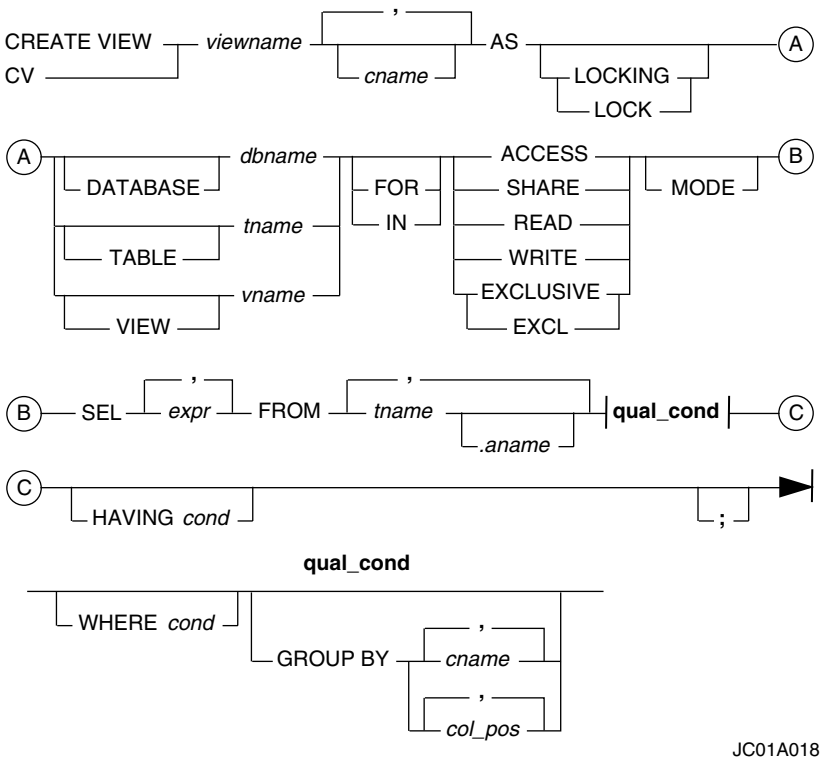


Diagram Identifier

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 Used 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 SMALL 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 A–Z 0–9	Any	Any single byte Latin letter or digit.
a–z A–Z 0–9	Unicode compatibility zone	Any fullwidth Latin letter or digit.
<	KanjiEBCDIC	Shift Out [SO] (0x0E). Indicates transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEBCDIC	Shift In [SI] (0x0F). Indicates transition from multibyte to single byte KanjiEBCDIC.
T	Any	Any multibyte character. The encoding depends on the current character set. For KanjiEUC, code set 3 characters are sometimes preceded by "ss_3".
I	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by "ss_2", forming an individual multibyte character.
Δ	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

APPENDIX B Restricted Words

This appendix details restrictions on the use of certain terminology in SQL queries and in other user application programs that interface with Teradata Database.

Teradata Reserved Words

Teradata Database reserved words cannot be used as identifiers to name host variables, correlations, local variables in stored procedures, objects (such as databases, tables, columns, or stored procedures), or parameters, such as macro or stored procedure parameters, because Teradata Database already uses the word and might misinterpret it.

The following table explains the notation that appears in the list of reserved words.

Notation	Explanation
AR above and to the right of a word	This word is also an ANSI SQL:2008 reserved word. For a list of ANSI SQL:2008 reserved words, see "ANSI SQL:2008 Reserved Words" on page 198.
AN above and to the right of a word	This word is also an ANSI SQL:2008 nonreserved word. For a list of ANSI SQL:2008 nonreserved words, see "ANSI SQL:2008 Nonreserved Words" on page 200.
Asterisk (*) above and to the right of a word	This word is new for this release.

ABORT ABORTSESSION ABS[AR] ACCESS_LOCK ACCOUNT ACOS ACOSH ADD[AN] ADD_MONTHS ADMIN[AN] AFTER[AN] AGGREGATE ALL[AR] ALTER[AR] AMP AND[AR] ANSIDATE ANY[AR] ARGLPAREN AS[AR] ASC[AN] ASIN ASINH AT[AR] ATAN ATAN2 ATANH ATOMIC[AR] AUTHORIZATION[AR] AVE AVERAGE AVG[AR]

BEFORE[AN] BEGIN[AR] BETWEEN[AR] BIGINT[AR] BINARY[AR] BLOB[AR] BOTH[AR] BT BUT BY[AR] BYTE BYTEINT BYTES

CALL[AR] CASE[AR] CASE_N CASESPECIFIC CAST[AR] CD CHAR[AR] CHAR_LENGTH[AR] CHAR2HEXINT CHARACTER[AR] CHARACTER_LENGTH[AR] CHARACTERS[AN] CHARS CHECK[AR] CHECKPOINT CLASS CLOB[AR]

CLOSE[AR] CLUSTER CM COALLESCE[AR] COLLATION[AN] COLLECT[AR]
COLUMN[AR] COMMENT COMMIT[AR] COMPRESS CONNECT[*AR]
CONSTRAINT[AR] CONSTRUCTOR[AN] CONSUME CONTAINS[AN] CONTINUE[AN]
CONVERT_TABLE_HEADER CORR[AR] COS COSH COUNT[AR] COVAR_POP[AR]
COVAR_SAMP[AR] CREATE[AR] CROSS[AR] CS CSUM CT CTCONTROL[*]
CUBE[AR] CURRENT[AR] CURRENT_DATE[AR] CURRENT_ROLE[*AR]
CURRENT_TIME[AR] CURRENT_TIMESTAMP[AR] CURRENT_USER[*AR] CURSOR[AR]
CV CYCLE[AR]

DATABASE DATABLOCKSIZE DATE[AR] DATEFORM DAY[AR] DEALLOCATE[AR]
DEC[AR] DECIMAL[AR] DECLARE[AR] DEFAULT[AR] DEFERRED[AN] DEGREES
DEL DELETE[AR] DESC[AN] DETERMINISTIC[AR] DIAGNOSTIC DISABLED
DISTINCT[AR] DO[AR] DOMAIN[AN] DOUBLE[AR] DROP[AR] DUAL DUMP
DYNAMIC[AR]

EACH[AR] ECHO ELSE[AR] ELSEIF[AR] ENABLED END[AR] EQ EQUALS[AN]
ERROR ERRORFILES ERRORTABLES ESCAPE[AR] ET EXCEPT[AR] EXEC[AR]
EXECUTE[AR] EXISTS[AR] EXIT[AN] EXP[AR] EXPAND[*] EXPANDING[*]
EXPLAIN EXTERNAL[AR] EXTRACT[AR]

FALLBACK FASTEXPORT FETCH[AR] FIRST[AN] FLOAT[AR] FOR[AR]
FOREIGN[AR] FORMAT FOUND[AN] FREESPACE FROM[AR] FULL[AR]
FUNCTION[AR]

GE GENERATED[AN] GET[*AR] GIVE GRANT[AR] GRAPHIC
GROUP[AR] GROUPING[AR] GT

HANDLER[AR] HASH HASHAMP HASHBAKAMP HASHBUCKET HASHROW
HAVING[AR] HELP HOUR[AR]

IDENTITY[AR] IF[AR] IMMEDIATE[AN] IN[AR] INCONSISTENT INDEX
INITIATE INNER[AR] INOUT[AR] INPUT[AN] INS INSERT[AR] INSTANCE[AN]
INSTEAD[AN] INT[AR] INTEGER[AR] INTEGERDATE INTERSECT[AR]
INTERVAL[AR] INTO[AR] IS[AR] ITERATE[AR]

JAR[AR] JOIN[AR] JOURNAL

KEY[AN] KURTOSIS

LANGUAGE[AR] LARGE[AR] LE LEADING[AR] LEAVE[AR] LEFT[AR]
LIKE[AR]LIMIT LN[AR] LOADING LOCAL[AR] LOCATOR[AN] LOCK LOCKING
LOG LOGGING LOGON LONG LOOP[AR] LOWER[AR] LT

MACRO MAP[AN] MAVG MAX[AR] MAXIMUM MCHARACTERS MDIFF
MERGE[AR] METHOD[AR] MIN[AR] MINDEX MINIMUM MINUS
MINUTE[AR] MLINREG MLOAD MOD[AR] MODE MODIFIES[AR] MODIFY
MONITOR MONRESOURCE MONSESSION MONTH[AR] MSUBSTR
MSUM MULTISET[AR]

NAMED NATURAL[AR] NE NEW[AR] NEW_TABLE NEXT[AN] NO[AR] NONE[AR]
NOT[AR] NOWAIT NULL[AR] NULLIF[AR] NULLIFZERO NUMERIC[AR]

OBJECT[AN] OBJECTS OCTET_LENGTH[AR] OF[AR] OFF OLD[AR]
OLD_TABLE ON[AR] ONLY[AR] OPEN[AR] OPTION[AN] OR[AR] ORDER[AR]
ORDERING[AN] OUT[AR] OUTER[AR] OVER[AR] OVERLAPS[AR] OVERRIDE

PARAMETER[AR] PASSWORD PERCENT PERCENT_RANK[AR] PERM
PERMANENT POSITION[AR] PRECISION[AR] PREPARE[AR]
PRESERVE[AN] PRIMARY[AR] PRIVILEGES[AN] PROCEDURE[AR] PROFILE
PROTECTION PUBLIC[AN]

QUALIFIED QUALIFY QUANTILE QUEUE

RADIANS RANDOM RANGE_N RANK[AR] READS[AR] REAL[AR]
RECURSIVE[AR] REFERENCES[AR] REFERENCING[AR] REGR_AVGX[AR]
REGR_AVGY[AR] REGR_COUNT[AR] REGR_INTERCEPT[AR] REGR_R2[AR]
REGR_SLOPE[AR] REGR_SXX[AR] REGR_SXY[AR] REGR_SYY[AR] RELATIVE[AN]
RELEASE[AR] RENAME REPEAT[AR] REPLACE REPLCONTROL REPLICATION
REQUEST RESIGNAL[*AR] RESTART[AN] RESTORE RESULT[AR] RESUME
RET RETRIEVE RETURN[AR] RETURNS[AR] REVALIDATE REVOKE[AR]
RIGHT[AR] RIGHTS ROLE[AN] ROLLBACK[AR] ROLLFORWARD ROLLUP[AR]
ROW[AR] ROW_NUMBER[AR] ROWID ROWS[AR]

SAMPLE SAMPLEID SCROLL[AR] SECOND[AR] SEL SELECT[AR] SESSION[AN]
SET[AR] SETRESRATE SETS[AN] SETSESSRATE SHOW SIGNAL[*AR] SIN
SINH SKEW SMALLINT[AR] SOME[AR] SOUNDEX SPECIFIC[AR] SPOOL
SQL[AR] SQLEXCEPTION[AR] SQLTEXT SQLWARNING[AR] SQRT[AR] SS
START[AR] STARTUP STATEMENT[AN] STATISTICS STDDEV_POP[AR]

STDDEV_SAMP[AR] STEPINFO STRING_CS SUBSCRIBER SUBSTR
SUBSTRING[AR] SUM[AR] SUMMARY SUSPEND

TABLE[AR] TAN TANH TBL_CS TEMPORARY[AN] TERMINATE THEN[AR]
THRESHOLD TIME[AR] TIMESTAMP[AR] TIMEZONE_HOUR[AR]
TIMEZONE_MINUTE[AR] TITLE TO[AR] TOP TRACE TRAILING[AR]
TRANSACTION[AN] TRANSFORM[AN] TRANSLATE[AR] TRANSLATE_CHK
TRIGGER[AR] TRIM[AR] TYPE[AN]

UC UDTCASTAS UDTCASTLPAREN UDTMETHOD UDTTYPE UDTUSAGE
UESCAPE[AR] UNDEFINED UNDO[AN] UNION[AR] UNIQUE[AR] UNTIL[AR]
UNTIL_CHANGED[*] UPD UPDATE[AR] UPPER[AR] UPPERCASE USER[AR]
USING[AR]

VALUE[AR] VALUES[AR] VAR_POP[AR] VAR_SAMP[AR] VARBYTE VARCHAR[AR]
VARGRAPHIC VARIANT_TYPE[*] VARYING[AR] VIEW[AN] VOLATILE

WHEN[AR] WHERE[AR] WHILE[AR] WIDTH_BUCKET[AR] WITH[AR] WITHOUT[AR]
WORK[AN]

XMLPLAN[*]

YEAR[AR]

ZEROIFNULL ZONE[AN]

Teradata Nonreserved Words

Teradata Database nonreserved keywords are permitted as identifiers but are discouraged because of possible confusion that may result.

The following table explains the notation that appears in the list of nonreserved words.

Notation	Explanation
AR above and to the right of a word	This word is also an ANSI SQL:2008 reserved word. For a list of ANSI SQL:2008 reserved words, see "ANSI SQL:2008 Reserved Words" on page 198.
AN above and to the right of a word	This word is also an ANSI SQL:2008 nonreserved word. For a list of ANSI SQL:2008 nonreserved words, see "ANSI SQL:2008 Nonreserved Words" on page 200.

Notation	Explanation
Plus sign (+) above and to the right of a word	This word must always follow the AS keyword if it appears as an alias name in the select list of a SELECT statement.
Asterisk (*) above and to the right of a word	This word is new for this release.

ACCESS AG ALLOCATION ALLPARAMS ALWAYS[AN] ANALYSIS APPLNAME[*] ARCHIVE ASCII ASSIGNMENT[AN] ATTR ATTRIBUTE[AN] ATTRIBUTES[AN] ATTRS

C[AN] CALLED[AR] CALLER CHANGERATE CHARACTERISTICS[AN] CHARSET_COLL CHECKSUM CLASS_ORIGIN[*AN] CLIENT COLUMNS COLUMNSPERINDEX COLUMNSPERJOININDEX COMMAND_FUNCTION[*AN] COMMAND_FUNCTION_CODE[*AN] COMPARABLE[AN] COMPARISON COMPILE CONDITION[*AR] CONDITION_IDENTIFIER[*AN] CONDITION_NUMBER[*AN] COST COSTS CPP CPUTIME CPUTIMENORM CREATOR[*]

DATA[AN] DBC DEBUG DEFINER[AN] DEMOGRAPHICS DIAGNOSTICS[*AN] DENIALS DIGITS DOWN[*] DR

EBCDIC ELAPSEDSEC ELAPSEDTIME ENCRYPT ERRORS EXCEPTION[*] EXCL EXCLUSIVE EXPIRE

FINAL[AN] FOLLOWING[AN]

G[AN] GLOBAL[AR] GLOP[*]

HIGH HOST

IFP INCREMENT[AN] INDEXESPERTABLE INDEXMAINTMODE INIT INSTANTIABLE[AN] INTERFACE[AN] INTERNAL INVOKER[AN] IOCOUNT ISOLATION[AN]

JAVA[AN] JIS_COLL

K[AN] KANJI1 KANJISJIS KBYTE KBYTES KEEP KILOBYTES

LAST[AN] LATIN LDIFF[*+] LEVEL[AN] LOCKEDUSEREXPIRE LOW

M[AN] MATCHED[AN] MAXCHAR MAXLOGONATTEMPTS MAXVALUE[AN] MEDIUM MEETS[*+] MEMBER[* AR] MESSAGE_LENGTH[* AN] MESSAGE_TEXT[* AN] MINCHAR MINVALUE[AN] MODIFIED MORE[* AN] MULTINATIONAL

NAME[AN] NODDLTEXT[*] NONOPTCOST NONOPTINIT NUMBER[* AN]

OA OLD_NEW_TABLE[*] ONLINE OPTIONS[AN] ORDERED_ANALYTIC OVERLAYS OWNER[*]

P_INTERSECT[*+] P_NORMALIZE[*+] PARAMID PARTITION[AR] PARTITIONED PARTITION#L1 PARTITION#L2 PARTITION#L3 PARTITION#L4 PARTITION#L5 PARTITION#L6 PARTITION#L7 PARTITION#L8 PARTITION#L9 PARTITION#L10 PARTITION#L11 PARTITION#L12 PARTITION#L13 PARTITION#L14 PARTITION#L15 PERIOD[*] PRECEDES[*+] PRECEDING[AN] PRINT PRIOR[* AN] PROTECTED

QUERY QUERY_BAND

RANDOMIZED RANGE[AR] RANGE#L1 RANGE#L2 RANGE#L3 RANGE#L4 RANGE#L5 RANGE#L6 RANGE#L7 RANGE#L8 RANGE#L9 RANGE#L10 RANGE#L11 RANGE#L12 RANGE#L13 RANGE#L14 RANGE#L15 RDIFF[*+] READ[AN] RECALC REPLACEMENT RESET[*] RESTRICTWORDS RETAIN RETURNED_SQLSTATE[* AN] REUSE ROW_COUNT[* AN] RU RULES[*] RULESET[*]

SAMPLES SEARCHSPACE SECURITY[AN] SEED SELF[AN] SERIALIZABLE[AN] SHARE SOURCE[AN] SPECCHAR SPL SQLDATA[AN] SQLSTATE[AR] SR STAT STATS STYLE[AN] SUBCLASS_ORIGIN[* AN] SUCCEEDS[*+] SUMMARYONLY SYSTEM[AR] SYSTEMTEST

TARGET TD_GENERAL TD_INTERNAL TEXT THROUGH[*] TIES[AN] TPA TRANSACTION_ACTIVE[* AN]

UNBOUNDED[AN] UNCOMMITTED[AN] UNICODE UNKNOWN[AR] USE

WARNING WRITE[AN]

XML[*]

Teradata Future Reserved Words

The words listed here are reserved for future Teradata Database use and cannot be used as identifiers.

The following table explains the notation that appears in the list of future reserved words.

Notation	Explanation
AR above and to the right of a word	This word is also an ANSI SQL:2008 reserved word. For a list of ANSI SQL:2008 reserved words, see "ANSI SQL:2008 Reserved Words" on page 198.
AN above and to the right of a word	This word is also an ANSI SQL:2008 nonreserved word. For a list of ANSI SQL:2008 nonreserved words, see "ANSI SQL:2008 Nonreserved Words" on page 200.
Asterisk (*) above and to the right of a word	This word is new for this release.

```
ALIAS

DESCRIPTOR[AN]

GO[AN]    GOTO[AN]

INDICATOR[AR]

PRIVATE

WAIT
```

ANSI SQL:2008 Reserved Words

The words listed here are ANSI SQL:2008 reserved words.

The following table explains the notation that appears in the list of reserved words.

Notation	Explanation
TR above and to the right of a word	This word is also a Teradata reserved word and cannot be used as an identifier. For a complete list of Teradata reserved words, see "Teradata Reserved Words" on page 191.
TN above and to the right of a word	This word is also a Teradata nonreserved word. Although the word is permitted as an identifier, it is discouraged because of possible confusion that may result. For a list of Teradata nonreserved words, see "Teradata Nonreserved Words" on page 194.
TF above and to the right of a word	This word is also a Teradata future reserved word and cannot be used as an identifier. For a complete list of Teradata future reserved words, see "Teradata Future Reserved Words" on page 197.

Note: Words in the following list that do have special notation are permitted as identifiers, but are discouraged because of possible confusion that may result.

ABS[TR] ALL[TR] ALLOCATE ALTER[TR] AND[TR] ANY[TR] ARE ARRAY ARRAY_AGG AS[TR] ASENSITIVE ASYMMETRIC AT[TR] ATOMIC[TR] AUTHORIZATION[TR] AVG[TR]

BEGIN[TR] BETWEEN[TR] BIGINT[TR] BINARY[TR] BLOB[TR] BOOLEAN BOTH[TR] BY[TR]

CALL[TR] CALLED[TN] CARDINALITY CASCADED CASE[TR] CAST[TR] CEIL CEILING CHAR[TR] CHAR_LENGTH[TR] CHARACTER[TR] CHARACTER_LENGTH[TR] CHECK[TR] CLOB[TR] CLOSE[TR] COALESCE[TR] COLLATE COLLECT[TR] COLUMN[TR] COMMIT[TR] CONDITION[TN] CONNECT[TR] CONSTRAINT[TR] CONVERT CORR[TR] CORRESPONDING COUNT[TR] COVAR_POP[TR] COVAR_SAMP[TR] CREATE[TR] CROSS[TR] CUBE[TR] CUME_DIST CURRENT[TR] CURRENT_CATALOG CURRENT_DATE[TR] CURRENT_DEFAULT_TRANSFORM_GROUP CURRENT_PATH CURRENT_ROLE[TR] CURRENT_SCHEMA CURRENT_TIME[TR] CURRENT_TIMESTAMP[TR] CURRENT_TRANSFORM_GROUP_FOR_TYPE CURRENT_USER[TR] CURSOR[TR] CYCLE[TR]

DATE[TR] DAY[TR] DEALLOCATE[TR] DEC[TR] DECIMAL[TR] DECLARE[TR] DEFAULT[TR] DELETE[TR] DENSE_RANK DEREF DESCRIBE

DETERMINISTIC[TR] DISCONNECT DISTINCT[TR] DO[TR] DOUBLE[TR]
DROP[TR] DYNAMIC[TR]

EACH[TR] ELEMENT ELSE[TR] ELSEIF[TR] END[TR] END-EXEC ESCAPE[TR]
EVERY EXCEPT[TR] EXEC[TR] EXECUTE[TR] EXISTS[TR] EXP[TR]
EXTERNAL[TR] EXTRACT[TR]

FALSE FETCH[TR] FILTER FLOAT[TR] FLOOR FOR[TR] FOREIGN[TR] FREE
FROM[TR] FULL[TR] FUNCTION[TR] FUSION

GET[TR] GLOBAL[TN] GRANT[TR] GROUP[TR] GROUPING[TR]

HANDLER[TR] HAVING[TR] HOLD HOUR[TR]

IDENTITY[TR] IF[TR] IN[TR] INDICATOR[TF] INNER[TR] INOUT[TR]
INSENSITIVE INSERT[TR] INT[TR] INTEGER[TR] INTERSECT[TR]
INTERSECTION INTERVAL[TR] INTO[TR] IS[TR] ITERATE[TR]

JAR[TR] JOIN[TR]

LANGUAGE[TR] LARGE[TR] LATERAL LEADING[TR] LEAVE[TR] LEFT[TR] LIKE[TR]
LIKE_REGEX LN[TR] LOCAL[TR] LOCALTIME LOCALTIMESTAMP LOOP[TR]
LOWER[TR]

MATCH MAX[TR] MEMBER[TN] MERGE[TR] METHOD[TR] MIN[TR] MINUTE[TR]
MOD[TR] MODIFIES[TR] MODULE MONTH[TR] MULTISET[TR]

NATIONAL NATURAL[TR] NCHAR NCLOB NEW[TR] NO[TR] NONE[TR]
NORMALIZE NOT[TR] NULL[TR] NULLIF[TR] NUMERIC[TR]

OCTET_LENGTH[TR] OCCURRENCES_REGEX OF[TR] OLD[TR] ON[TR] ONLY[TR]
OPEN[TR] OR[TR] ORDER[TR] OUT[TR] OUTER[TR] OVER[TR] OVERLAPS[TR]
OVERLAY

PARAMETER[TR] PARTITION[TN] PERCENT_RANK[TR] PERCENTILE_CONT
PERCENTILE_DISC POSITION[TR] POSITION_REGEX POWER PRECISION[TR]
PREPARE[TR] PRIMARY[TR] PROCEDURE[TR]

RANGE[TN] RANK[TR] READS[TR] REAL[TR] RECURSIVE[TR] REF
REFERENCES[TR] REFERENCING[TR] REGR_AVGX[TR] REGR_AVGY[TR]

REGR_COUNT[TR] REGR_INTERCEPT[TR] REGR_R2[TR] REGR_SLOPE[TR]
REGR_SXX[TR] REGR_SXY[TR] REGR_SYY[TR] RELEASE[TR] REPEAT[TR]
RESIGNAL[TR] RESULT[TR] RETURN[TR] RETURNS[TR] REVOKE[TR] RIGHT[TR]
ROLLBACK[TR] ROLLUP[TR] ROW[TR] ROW_NUMBER[TR] ROWS[TR]

SAVEPOINT SCOPE SCROLL[TR] SEARCH SECOND[TR] SELECT[TR]
SENSITIVE SESSION_USER SET[TR] SIGNAL[TR] SIMILAR SMALLINT[TR]
SOME[TR] SPECIFIC[TR] SPECIFICTYPE SQL[TR] SQLEXCEPTION[TR]
SQLSTATE[TN] SQLWARNING[TR] SQRT[TR] START[TR] STATIC STDDEV_POP[TR]
STDDEV_SAMP[TR] SUBMULTISET SUBSTRING[TR] SUBSTRING_REGEX SUM[TR]
SYMMETRIC SYSTEM[TN] SYSTEM_USER

TABLE[TR] TABLESAMPLE THEN[TR] TIME[TR] TIMESTAMP[TR]
TIMEZONE_HOUR[TR] TIMEZONE_MINUTE[TR] TO[TR] TRAILING[TR] TRANSLATE[TR]
TRANSLATE_REGEX TRANSLATION TREAT TRIGGER[TR] TRUNCATE
TRIM[TR] TRUE

UESCAPE[TR] UNION[TR] UNIQUE[TR] UNKNOWN[TN] UNNEST UNTIL[TR]
UPDATE[TR] UPPER[TR] USER[TR] USING[TR]

VALUE[TR] VALUES[TR] VAR_POP[TR] VAR_SAMP[TR] VARBINARY VARCHAR[TR]
VARYING[TR]

WHEN[TR] WHENEVER WHERE[TR] WHILE[TR] WIDTH_BUCKET[TR] WINDOW
WITH[TR] WITHIN WITHOUT[TR]

YEAR[TR]

ANSI SQL:2008 Nonreserved Words

The words listed here are ANSI SQL:2008 nonreserved words.

The following table explains the notation that appears in the list of nonreserved words.

Notation	Explanation
TR above and to the right of a word	This word is also a Teradata reserved word and cannot be used as an identifier. For a complete list of Teradata reserved words, see “Teradata Reserved Words” on page 191.

Notation	Explanation
TN above and to the right of a word	This word is also a Teradata nonreserved word. Although the word is permitted as an identifier, it is discouraged because of possible confusion that may result. For a list of Teradata nonreserved words, see "Teradata Nonreserved Words" on page 194.
TF above and to the right of a word	This word is also a Teradata future reserved word and cannot be used as an identifier. For a complete list of Teradata future reserved words, see "Teradata Future Reserved Words" on page 197.

Note: Words in the following list that do have special notation are permitted as identifiers, but are discouraged because of possible confusion that may result.

A ABSOLUTE ACTION ADA ADD[TR] ADMIN[TR] AFTER[TR] ALWAYS[TN]
ASC[TR] ASSERTION ASSIGNMENT[TN] ATTRIBUTE[TN] ATTRIBUTES[TN]

BEFORE[TR] BERNOULLI BREADTH

C[TN] CASCADE CATALOG CATALOG_NAME CHAIN
CHARACTER_SET_CATALOG CHARACTER_SET_NAME CHARACTER_SET_SCHEMA
CHARACTERISTICS[TN] CHARACTERS[TR] CLASS_ORIGIN[TN] COBOL
COLLATION[TR] COLLATION_CATALOG COLLATION_NAME COLLATION_SCHEMA
COLUMN_NAME COMMAND_FUNCTION[TN] COMMAND_FUNCTION_CODE[TN]
COMMITTED COMPARABLE[TN] CONDITION_IDENTIFIER[TN]
CONDITION_NUMBER[TN] CONNECTION CONNECTION_NAME
CONSTRAINT_CATALOG CONSTRAINT_NAME CONSTRAINT_SCHEMA
CONSTRAINTS CONSTRUCTOR[TR] CONTAINS[TR] CONTINUE[TR] CURSOR_NAME

DATA[TN] DATETIME_INTERVAL_CODE DATETIME_INTERVAL_PRECISION
DEFAULTS DEFERRABLE DEFERRED[TR] DEFINED DEFINER[TN] DEGREE
DEPTH DERIVED DESC[TR] DESCRIPTOR[TF] DIAGNOSTICS[TN] DISPATCH
DOMAIN[TR] DYNAMIC_FUNCTION DYNAMIC_FUNCTION_CODE

EQUALS[TR] EXCLUDE EXCLUDING EXIT[TR]

FINAL[TN] FIRST[TR] FLAG FOLLOWING[TN] FORTRAN FOUND[TR]

G[TN] GENERAL GENERATED[TR] GO[TF] GOTO[TF] GRANTED

HIERARCHY

IMMEDIATE[TR] IMPLEMENTATION INCLUDING INCREMENT[TN] INITIALLY INPUT[TR] INSTANCE[TR] INSTANTIABLE[TN] INSTEAD[TR] INTERFACE[TN] INVOKER[TN] ISOLATION[TN]

JAVA[TN]

K[TN] KEY[TR] KEY_MEMBER KEY_TYPE

LAST[TN] LENGTH LEVEL[TN] LOCATOR[TR]

M[TN] MAP[TR] MATCHED[TN] MAXVALUE[TN] MESSAGE_LENGTH[TN] MESSAGE_OCTET_LENGTH MESSAGE_TEXT[TN] MINVALUE[TN] MORE[TN] MUMPS

NAME[TN] NAMES NESTING NEXT[TR] NFC NFD NFKC NFKD NORMALIZED NULLABLE NULLS NUMBER[TN]

OBJECT[TR] OCTETS OPTION[TR] OPTIONS[TN] ORDERING[TR] ORDINALITY OTHERS OUTPUT OVERRIDING

P PAD PARAMETER_MODE PARAMETER_NAME PARAMETER_ORDINAL_POSITION PARAMETER_SPECIFIC_CATALOG PARAMETER_SPECIFIC_NAME PARAMETER_SPECIFIC_SCHEMA PARTIAL PASCAL PATH PLACING PLI PRECEDING[TN] PRESERVE[TR] PRIOR[TN] PRIVILEGES[TR] PUBLIC[TR]

READ[TN] RELATIVE[TR] REPEATABLE RESTART[TR] RESTRICT RETURNED_CARDINALITY RETURNED_LENGTH RETURNED_OCTET_LENGTH RETURNED_SQLSTATE[TN] ROLE[TR] ROUTINE ROUTINE_CATALOG ROUTINE_NAME ROUTINE_SCHEMA ROW_COUNT[TN]

SCALE SCHEMA SCHEMA_NAME SCOPE_CATALOG SCOPE_NAME SCOPE_SCHEMA SECTION SECURITY[TN] SELF[TN] SEQUENCE SERIALIZABLE[TN] SERVER_NAME SESSION[TR] SETS[TR] SIMPLE SIZE SOURCE[TN] SPACE SPECIFIC_NAME SQLDATA[TN] STACKED STATE STATEMENT[TR] STRUCTURE STYLE[TN] SUBCLASS_ORIGIN[TN]

T TABLE_NAME TEMPORARY[TR] TIES[TR] TOP_LEVEL_COUNT TRANSACTION[TR] TRANSACTION_ACTIVE[TN] TRANSACTIONS_COMMITTED TRANSACTIONS_ROLLED_BACK TRANSFORM[TR] TRANSFORMS TRIGGER_CATALOG TRIGGER_NAME TRIGGER_SCHEMA TYPE[TR]

UNBOUNDED[TN] UNCOMMITTED[TN] UNDER UNDO[TR] UNNAMED USAGE
USER_DEFINED_TYPE_CATALOG USER_DEFINED_TYPE_CODE
USER_DEFINED_TYPE_NAME USER_DEFINED_TYPE_SCHEMA

VIEW[TR]

WORK[TR] WRITE[TR]

ZONE[TR]

APPENDIX C Teradata Database Limits

This appendix provides information on the limis that apply to Teradata Database systems, databases, and sessions.

System Limits

The system specifications in the following tables apply to an entire Teradata Database *configuration*.

Miscellaneous System Limits

Parameter	Value
Maximum number of combined databases and users.	4.2×10^9
Maximum number of database objects per system lifetime.[a]	1,073,741,824
Maximum data format descriptor size.	30 characters
Maximum size for a thin table header.	64 KB
Maximum size for a fat table header.	1 MB
Maximum size of table header cache.	8×10^6 bytes
Maximum size of a response spool file row.	64 KB
Maximum number of change logs that can be specified for a system at any point in time.	1,000,000
Maximum aggregate number of locks that can be placed for the online archive logging of tables, databases, or both per request.	25,000[b]
Maximum number of 64KB parse tree segments allocated for parsing requests.	12,000

a. A *database object* is any object whose definition is recorded in *DBC.TVM*.

   Because the system does not reuse *DBC.TVM* IDs, this means that a maximum of 1,073,741,824 such objects can be created over the lifetime of any given system. At the rate of creating one new database object per minute, it would take 2,042 years to use 1,073,741,824 unique IDs.

b. The maximum number of locks that can be placed per LOGGING ONLINE ARCHIVE ON request is fixed at 25,000 and cannot be altered.

Message Limits

Parameter	Value
Maximum number of CLIv2 parcels per message.	256
Maximum message size.	~ 65,000 bytes This limit applies to messages to and from client systems and to some internal Teradata Database messages.
Maximum error message text size in a failure parcel.	255 bytes

Storage Limits

Parameter	Value
Total data capacity.	• Expressed as a base 10 value: 1.39 TB/AMP (1.39×10^{12} bytes/AMP) • Expressed as a base 2 value: 1.26 TB/AMP (1.26×10^{12} bytes/AMP)
Maximum number of sectors per data block.	255[a]
Minimum data block size.	9,216 bytes (9 KB - 18 sectors)
Maximum data block size.	130,560 bytes (127.5 KB - 255 sectors)
Maximum number of data blocks that can be merged per data block merge.	8
Maximum merge block ratio	60% of the maximum multirow block size for a table.
Data block header size.	Depends on several factors. **FOR a data block that is …** / **The data block header size is this many bytes …** new or has been updated: 72 on a system and has not been updated: 40

a. The increase in data block header size from 36 or 40 bytes to 64 bytes increases the size of roughly 6 percent of the data blocks by one sector (see the limits for “Data block header size” in this table).

Gateway and vproc Limits

Parameter	Value
Maximum number of sessions per PE.	120
Maximum number of gateways per node.	Multiple.[a] The exact number depends on whether the gateways belong to different host groups and listen on different IP addresses.
Maximum number of sessions per gateway.	Tunable.[b] 1,200 maximum certified.
Maximum number of PEs per system.	1,024
Maximum number of AMPs per system.	16,383[c] More generally, the maximum number of AMPs per system depends on the number of PEs in the configuration. The following equation provides the most general solution: 16,384 – number_of_PEs
Maximum number of AMP and PE vprocs, in any combination, per node.	128
Maximum number of AMP and PE vprocs, in any combination, per system.	16,384
Maximum number of external routine protected mode platform tasks per PE or AMP.	20[d]
Maximum number of external routine secure mode platform tasks per PE or AMP.	20[c]
Maximum number of locks that can be placed at a time per AMP.	27,000[e]
Maximum size of the lock table per AMP.	2 MB[f]
Maximum size of the queue table FIFO runtime cache per PE.	• 100 queue table entries • 1 MB
Maximum number of SELECT AND CONSUME statements in a delayed state per PE.	24
Amount of private disk swap space required per protected or secure mode server for C/C++ external routines per PE or AMP vproc.	256 KB

Parameter	Value
Amount of private disk swap space required per protected or secure mode server for Java external routines per node.	30 MB

a. This is true because the gateway runs in its own vproc on each node. See *Utilities* for details.

b. See *Utilities* for details.

c. This value is derived by subtracting 1 from the maximum total of PE and AMP vprocs per system (because each system must have at least one PE), which is 16,384.
This is obviously not a practical configuration.

d. The valid range is 0 to 20, inclusive. The limit is 20 platforms for *each* platform type, not 20 combined for both. See *Utilities* for details.

e. The number of locks that can be placed at a time per AMP is fixed at 27,000 and cannot be altered.

f. The AMP lock table size is fixed at 2 MB and cannot be altered.

Hash Bucket Limits

Parameter	Value
Number of hash buckets per system.[a]	The number is user-selectable per system. • 65,536 • 1,048,576 Bucket numbers range from 0 to the system maximum.
Size of a hash bucket	The size depends on the number of hash buckets on the system. IF the system has this number of hash buckets … / THEN the size of a hash bucket is this many bits … 65,536 / 16 1,048,576 / 20 You set the default hash bucket size for your system using the DBS Control utility (see *Utilities* for details).

a. This value is user-selectable. See the topic “Hash Maps” in Chapter 9 of *Database Design* for details.

Interval Histogram Limits

Parameter	Value
Number of hash values	4.2×10^9
Maximum number of interval histograms per index or column set[a]	200
Maximum number of equal-height intervals per interval histogram[b]	200
Maximum number of loner intervals per interval histogram[b]	199
Maximum number of loners per high-biased interval[b]	2
Maximum number of values represented per high-biased interval[b]	2
Maximum number of loners per interval histogram[b, b]	398

a. The system wide maximum number of interval histograms is set using the MaxStatsInterval flag of the DBS Control record or your cost profile.

   For descriptions of the other parameters listed, see *SQL Request and Transaction Processing* for details.

b. The maximum number of loners per interval histogram is determined by subtracting 2 from the maximum number of interval histograms set for your system using the MaxStatsInterval flag.

Database Limits

The database specifications in the following tables apply to a single Teradata database. The values presented are for their respective parameters individually *and not in combination*.

Table and View Limits

Parameter	Value
Maximum number of journal tables per database.	1
Maximum number of error tables per base data table.	1
Maximum number of columns per base data table or view.	2,048
Maximum number of columns per error table.	2,061[d]
Maximum number of UDT columns per base data table.	~1,600[e, f, g]
Maximum number of LOB columns per base data table.	32[h]
Maximum number of columns created over the life of a base data table.	2,560
Maximum number of rows per base data table.	Limited only by disk capacity.
Maximum number of bytes per table header per AMP.[a]	1 megabyte
Maximum number of characters per SQL index constraint.	16,000

Parameter	Value
Maximum row size.	65,536 bytes
Maximum size of the queue table FIFO runtime cache per table.	2,211 row entries
Maximum logical row size.[b]	67,106,816,000 bytes[i]
Maximum non-LOB column size.	• 65,522 bytes (NPPI table)[j] • 65 520 bytes (PPI table)[k]
Maximum number of compressed values per base table column.	255 plus nulls[l]
Maximum number of primary indexes per table.	1[q]
Minimum number of primary indexes per table.	1[m]
Maximum number of columns per primary index.	64
Maximum number of combined partitions for a partitioned primary index.	65,535
Maximum number of partitioning levels or partitioning expressions for a multilevel partitioned primary index[c]	15
Minimum number of partitions per partitioning level for a multilevel partitioned primary index.	2
Maximum number of table-level constraints per base data table.	100
Maximum number of referential integrity constraints per base data table.	64
Maximum number of reference indexes per base data table.	64[n]
Maximum number of columns per foreign and parent keys.	64
Maximum number of characters per string constant.	31,000
Minimum PERM space required by a materialized global temporary table for its table header.	512 bytes × number of AMPs

a. A table header that is large enough to require more than ~64,000 bytes uses multiple 64Kbyte rows per AMP. A table header that requires 64,000 or fewer bytes does not use the additional rows that are required to contain a fat table header.

   The *maximum* size for a table header is 1 megabyte.

b. In this case, a logical row is defined as a base table row plus the sum of the bytes stored in a LOB subtable for that row.

c. Other limits can further restrict the number of levels for a specific partitioning.

d. 2,048 data table columns plus 13 error table columns.

e. The absolute limit is 2,048, and the realizable number varies as a function of the number of other features declared for a table that occupy table header space.

f. The figure of 1,600 UDT columns assumes a *fat* table header.

g. This limit is true whether the UDT is a distinct or a structured type.

h. This includes both predefined type LOB columns and UDT LOB columns.
A LOB UDT column counts as one LOB column even if the UDT is a structured type that has multiple LOB attributes.

i. This value is derived by multiplying the maximum number of LOB columns per base table (32) times the maximum size of a LOB column (2,097,088,000 8-bit bytes). Remember that each LOB column consumes 39 bytes of Object ID from the base table, so 1 248 of those 67,106,816,000 bytes cannot be used for data.

j. Based on subtracting the minimum row overhead value for an NPPI table row (14 bytes) from the system-defined maximum row length (65,536 bytes).

k. Based on subtracting the minimum row overhead value for a PPI table row (16 bytes) from the system-defined maximum row length (65,536 bytes).

l. Nulls are always compressed by default.

m. This is not true for global temporary trace tables. These tables are not hashed, so they do not have a primary index.

n. There are 128 Reference Indexes in a table header, 64 from a parent table to child tables and 64 from child tables to a parent table.

Large Object and Related Limits

Parameter	Value
Maximum BLOB object size.	2,097,088,000 8-bit bytes
Maximum CLOB object size.	• 2,097,088,000 single-byte characters • 1,048,544,000 double-byte characters
Maximum size of the file name passed to the AS DEFERRED BY NAME option in a USING request modifier.	VARCHAR(1024)

User-Defined Data Type Limits

Parameter	Value
Maximum structured UDT size.[a]	• 65,521 bytes (NPPI table) • 65,519 bytes (PPI table)
Maximum number of UDT columns per base data table.	~1,600[b, c, d]
Maximum database, user, base table, view, macro, index, trigger, stored procedure, UDF, UDM, UDT, replication group name, constraint, or column name size.	30 bytes
Maximum number of attributes that can be specified for a structured UDT per CREATE TYPE or ALTER TYPE statement.	300 - 512[e]
Maximum number of attributes that can be defined for a structured UDT.	~4,000[f]
Maximum number of levels of nesting of attributes that can be specified for a structured UDT.	512
Maximum number of methods associated with a UDT.	~500[g]
Maximum number of input parameters with a UDT data type of VARIANT_TYPE that can be declared for a UDF definition.	8

a. Based on a table having a 1 byte (BYTEINT) primary index. Because a UDT column cannot be part of any index definition, there must be at least one non-UDT column in the table for its primary index. Row header overhead consumes 14 bytes in an NPPI table and 16 bytes in a PPI table, so the maximum structured UDT size is derived by subtracting 15 bytes (for an NPPI table) or 17 bytes (for a PPI table) from the row maximum of 65,536 bytes.

b. The absolute limit is 2,048, and the realizable number varies as a function of the number of other features declared for a table that occupy table header space.

c. The figure of 1,600 UDT columns assumes a FAT table header.

d. This limit is true whether the UDT is a distinct or a structured type.

e. The maximum is platform-dependent, not absolute.

f. While you can specify no more than 300 to 512 attributes for a structured UDT per CREATE TYPE or ALTER TYPE statement, you can submit any number of ALTER TYPE statements with the ADD ATTRIBUTE option specified as necessary to add additional attributes to the type up to the upper limit of approximately 4,000.

g. There is no absolute limit on the number of methods that can be associated with a given UDT. Methods can have a variable number of parameters, and the number of parameters directly affects the limit, which is due to Parser memory restrictions.
There is a workaround for this issue. See the documentation for "ALTER TYPE" in *SQL Data Definition Language* for details.

Macro, UDF, SQL Stored Procedure, and External Routine Limits

Parameter	Value
Maximum number of parameters specified in a macro.	2,048
Maximum expanded text size for macros and views.	2 Mbytes
Maximum number of open cursors per stored procedure.	15
Maximum number of result sets a stored procedure can return	15[b]
Maximum number of columns returned by a dynamic result table function.	2,048[c]
Maximum number of dynamic SQL statements per stored procedure	15
Maximum length of a dynamic SQL statement in a stored procedure	65,536 bytes[d]
Maximum number of parameters specified in a UDF defined without dynamic UDT parameters.	128
Maximum number of parameters specified in a UDF defined with dynamic UDT parameters.	1,144[e]
Maximum number of parameters specified in a UDM.	128
Maximum number of parameters specified in an SQL stored procedure.	256
Maximum number of parameters specified in an external stored procedure written in C or C++.	256
Maximum number of parameters specified in an external stored procedure written in Java.	255
Maximum length of external name string for an external routine.[a]	1,000 characters
Maximum package path length for an external routine.	256 characters
Maximum number of nested CALL statements in a stored procedure.	15
Maximum SQL text size in a stored procedure.	64 kilobytes

Parameter	Value
Maximum number of Statement Areas per SQL stored procedure diagnostics area.	1
Maximum number of Condition Areas per SQL stored procedure diagnostics area.	16

a. An external routine is the portion of a UDF, external stored procedure, or method that is written in C, C++, or Java (only external stored procedures can be written in Java). This is the code that defines the semantics for the UDF, procedure, or method.

b. The valid range is from 0 to 15. The default is 0.

c. The valid range is from 1 to 2,048. There is no default.

d. This includes its SQL text, the USING data (if any), and the CLIv2 parcel overhead.

e. 120 standard parameters plus an additional 1,024 (8*128 = 1,024) dynamic UDT parameters.

Query and Workload Analysis Limits

Parameter	Value
Maximum size of the Index Wizard workload cache.	187 megabytes[b]
Maximum number of columns and indexes on which statistics can be recollected for a table or join index subtable.	512
Maximum number of secondary, hash, and join indexes, in any combination, on which statistics can be collected and maintained at one time. This limit is independent of the number of pseudoindexes on which statistics can be collected and maintained.	32
Maximum number of pseudoindexes[a] on which multicolumn statistics can be collected and maintained at one time. This limit is independent of the number of indexes on which statistics can be collected and maintained.	32
Maximum number of sets of multicolumn statistics that can be collected on a table if single-column PARTITION statistics are *not* collected on the table.	32
Maximum number of sets of multicolumn statistics that can be collected on a table if single-column PARTITION statistics *are* collected on the table.	31
Maximum size of SQL query text overflow stored in QCD table *QryRelX* that can be read by the Teradata Index Wizard.	1 megabyte

a. A pseudoindex is a file structure that allows you to collect statistics on a composite, or multicolumn, column set in the same way you collect statistics on a composite index.

b. The default is 48 Mbytes and the minimum is 32 Mbytes.

Secondary, Hash, and Join Index Limits

Parameter	Value
Minimum number of secondary, hash, and join indexes, in any combination, per base data table.	0[c]
Maximum number of secondary[a], hash, and join indexes, in any combination, per base data table.	32
Maximum number of columns per secondary index.	64
Maximum number of columns referenced per single table in a hash or join index.	64
Maximum number of columns referenced in the fixed part of a compressed join index.[b]	64
Maximum number of columns referenced in the repeating part of a compressed join index.	64
Maximum number of columns in an uncompressed join index.	2,048
Maximum number of columns in a compressed join index.	128

a. Each composite NUSI defined with an ORDER BY clause counts as 2 consecutive indexes in this calculation.

b. There are two very different types of user-visible compression in the system.

   When describing compression of hash and join indexes, compression refers to a logical row compression in which multiple sets of nonrepeating column values are appended to a single set of repeating column values. This allows the system to store the repeating value set only once, while any nonrepeating column values are stored as logical segmental extensions of the base repeating set.

   When describing compression of column values, compression refers to the storage of those values one time only in the table header, not in the row itself, and pointing to them by means of an array of presence bits in the row header. The method is called Dictionary Indexing, and it can be viewed as a variation of Run-Length Encoding (see “Value Compression” in *Database Design* for additional information).

c. The only index required for any Teradata Database table is a primary index.

SQL Request and Response Limits

Parameter	Value
Maximum SQL request size.	1 megabyte (Includes SQL statement text, USING data, and parcel overhead)
Maximum number of entries in an IN list.	unlimited[a]
Maximum SQL title size.	60 characters
Maximum SQL activity count size.	4,294,967,295 rows[b]
Maximum number of tables and single-table views that can be joined per query block.	128[c]

Parameter	Value
Maximum number of partitions for a hash join operation.	50
Maximum number of subquery nesting levels per query.	64
Maximum number of tables or single-table views that can be referenced per subquery.	128[c]
Maximum number of fields in a USING row descriptor.	2,547
Maximum number of open cursors per embedded SQL program	16
Maximum SQL response size.	1 megabyte
Maximum number of columns per DML statement ORDER BY clause.	16

a. There is no fixed limit on the number of entries in an IN list; however, other limits such as the maximum SQL text size, place a request-specific upper bound on this number.

b. Derived from 2^{32} - 1.

c. This limit is controlled by the MaxJoinTables DBS Control and Cost Profile parameter flags.

Session Limits

The session specifications in the following table apply to a single *session*:

<table>
<tr><th>Parameter</th><th>Value</th></tr>
<tr><td>Maximum number of sessions per PE.</td><td>120</td></tr>
<tr><td>Maximum number of sessions per gateway.</td><td>Tunable.[a]
1,200 maximum certified.</td></tr>
<tr><td>Maximum number of sessions per vproc/host group.</td><td>Depends on the operating system.
<table>
<tr><th>FOR this operating system …</th><th>The maximum number of sessions per …</th></tr>
<tr><td>• Windows
• Linux</td><td>vproc is 2,147,483,647.
The valid range is 1 - 2,147,483,647.
The default is 600.</td></tr>
<tr><td>UNIX MP-RAS</td><td>host group depends on the availability of Teradata Database resources.</td></tr>
</table></td></tr>
<tr><td>Active request result spool files.</td><td>16</td></tr>
<tr><td>Maximum number of parallel steps per request.
Parallel steps can be used to process a request submitted within a transaction (which can be either explicit or implicit).</td><td>20 steps</td></tr>
<tr><td>Maximum number of channels required for various parallel step operations.
The value per request is determined as follows:</td><td></td></tr>
<tr><td>• Primary index request with an equality constraint.</td><td>0 channels</td></tr>
<tr><td>• Requests that do not involve row distribution.</td><td>2 channels</td></tr>
<tr><td>• Requests that involves redistribution of rows to other AMPs, such as a join or an INSERT … SELECT operation.</td><td>4 channels</td></tr>
<tr><td>Maximum number of materialized global temporary tables per session.</td><td>2,000</td></tr>
</table>

Parameter	Value
Maximum number of volatile tables per session.	1,000
Maximum number of SQL stored procedure Diagnostic Areas per session.	1

a. See *Utilities* for details.

APPENDIX D ANSI SQL Compliance

This appendix describes the ANSI SQL standard, Teradata compliance with the ANSI SQL standard, and terminology differences between ANSI SQL and Teradata SQL.

ANSI SQL Standard

Introduction

The American National Standards Institute (ANSI) defines a version of SQL that all vendors of relational database management systems support to a greater or lesser degree.

The complete ANSI/ISO SQL:2008 standard is defined across nine individual volumes.

The following table lists each of the individual component standards of the complete ANSI SQL:2008 standard. This standard cancels and replaces the ANSI SQL:2003 standard.

Title	Description
ANSI/ISO/IEC 9075-1:2008, Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)	Defines the conceptual framework for the entire standard. The subject matter covered is similar to *SQL Fundamentals*.
ANSI/ISO/IEC 9075-2:2008, Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)	Defines the data structures of, and basic operations on, SQL data, including the definitions of the statements, clauses, and operators of the SQL language. The subject matter covered is similar to the following Teradata manuals: • *Data Dictionary* • *SQL Request and Transaction Processing* (Chapter 9) • *SQL Data Types and Literals* • *SQL Data Definition Language* • *SQL Functions, Operators, Expressions, and Predicates* • *SQL Data Manipulation Language* • *SQL Stored Procedures and Embedded SQL* (Material about embedded SQL)

Title	Description
ANSI/ISO/IEC 9075-3:2008, Information technology — Database languages — SQL — Part 3: Call-Level Interface (SQL/CLI)	Defines the structures and procedures used to execute SQL statements from a client application using function calls. The subject matter covered is similar to *ODBC Driver for Teradata User Guide*.
ANSI/ISO/IEC 9075-4:2008, Information technology — Database languages — SQL — Part 4: Persistent Stored Modules (SQL/PSM)	Defines the syntax and semantics for declaring and maintaining persistent routines on the database server. The subject matter covered is similar to the following Teradata manuals: • *SQL External Routine Programming* • *SQL Stored Procedures and Embedded SQL* (Material about stored procedure control statements) • *SQL Data Definition Language* (Material about creating and maintaining user-defined functions, methods, and external stored procedures) • *SQL Data Manipulation Language* (CALL statement)
ANSI/ISO/IEC 9075-9:2008, Information technology — Database languages — SQL — Part 9: Management of External Data (SQL/MED)	Defines SQL language extensions for supporting external data using foreign data wrappers and datalink types.
ANSI/ISO/IEC 9075-10:2008, Information technology — Database languages — SQL — Part 10: Object Language Bindings (SQL/OLB)	Defines SQL language extensions to support embedded SQL for Java applications.
ANSI/ISO/IEC 9075-11:2008, Information technology — Database languages — SQL — Part 11: Information and Definition Schemas (SQL/Schemata)	"…provide[s] a data model to support the Information Schema and to assist understanding. An SQL-implementation need do no more than simulate the existence of the Definition Schema, as viewed through the Information Schema views. The specification does not imply that an SQL implementation shall provide the functionality in the manner described in the Definition Schema."
ANSI/ISO/IEC 9075-13:2008, Information technology — Database languages — SQL — Part 13: SQL Routines and Types using the Java Programming Language (SQL/JRT)	Defines the call-level interface API for Java SQL function calls. The subject matter covered is similar to the following Teradata manuals: • *Teradata JDBC Driver User Guide* • *SQL External Routine Programming* (Material about Java external routines) • *SQL Data Definition Language* (Material about Java external routines)

Title	Description
ANSI/ISO/IEC 9075-14:2008, Information technology — Database languages — SQL — Part 14: XML-Related Specifications (SQL/XML)	Defines how the SQL language manipulates XML text. **Note:** This standard cancels and replaces the 2006 version of this standard.

Final versions of the standards are available for purchase from the International Organization for Standardization (ISO) at http://www.iso.org.

You can download the complete set of working drafts of the ANSI SQL:2008 standard at no charge from the following URL: http://www.wiscorp.com/SQLStandards.html. For nearly all applications, the working drafts are identical to the final standards.

Motivation Behind an SQL Standard

Teradata, like most vendors of relational database management systems, had its own dialect of the SQL language for many years prior to the development of the SQL standard.

You might ask several questions like the following:

- Why should there be an industry-wide SQL standard?
- Why should any vendor with an entrenched user base consider modifying its SQL dialect to conform with the ANSI SQL standard?

Advantages of Having an SQL Standard

National and international standards abound in the computer industry. As anyone who has worked in the industry for any length of time knows, standardization offers both advantages and disadvantages both to users and to vendors.

The principal advantages of having an SQL standard are the following:

- Open systems

  The overwhelming trend in computer technology has been toward open systems with publicly defined standards to facilitate third party and end user access and development using the standardized products.

  The ANSI SQL standard provides an open definition for the SQL language.
- Less training for transfer and new employees

  A programmer trained in ANSI-standard SQL can move from one SQL programming environment to another with no need to learn a new SQL dialect. When a core dialect of the language is the lingua franca for SQL programmers, the need for retraining is significantly reduced.
- Application portability

  When there is a standardized public definition for a programming language, users can rest assured that any applications they develop to the specifications of that standard are portable to any environment that supports the same standard.

This is an extremely important budgetary consideration for any large scale end user application development project.

- Definition and manipulation of heterogeneous databases is facilitated

  Many user data centers support multiple merchant databases across different platforms. A standard language for communicating with relational databases, irrespective of the vendor offering the database management software, is an important factor in reducing the overhead of maintaining such an environment.
- Intersystem communication is facilitated

  An enterprise commonly exchanges applications and data among different merchant databases.

  Common examples of this follow.

  - Two-phase commit transactions where rows are written to multiple databases simultaneously.
  - Bulk data import and export between different vendor databases.

  These operations are made much cleaner and simpler when there is no need to translate data types, database definitions, and other component definitions between source and target databases.

Teradata Compliance With the ANSI Standard

Conformance to a standard presents problems for any vendor that produces an evolved product and supports a large user base.

Teradata, in its historical development, has produced any number of innovative SQL language elements that do not conform to the ANSI SQL standard, a standard that did not exist when those features were conceived. The existing Teradata user base had invested substantial time, effort, and capital into developing applications using that Teradata SQL dialect.

At the same time, new customers demand that vendors conform to open standards for everything from chip sets to operating systems to application programming interfaces.

Meeting these divergent requirements presents a challenge that Teradata SQL solves by following the multipronged policy outlined in the following table.

WHEN ...	THEN ...
a new feature or feature enhancement is added to Teradata SQL	that feature conforms to the ANSI SQL standard.
the difference between the Teradata SQL dialect and the ANSI SQL standard for a language feature is slight	the ANSI SQL is added to Teradata Database feature as an option.

<table>
<tr><th>WHEN ...</th><th>THEN ...</th></tr>
<tr><td>the difference between the Teradata SQL dialect and the ANSI SQL standard for a language feature is significant</td><td>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 in the following ways:
Persistently
Use the SessionMode field of the DBS Control Record to define session mode characteristics.
For a session
Use the BTEQ .SET SESSION TRANSACTION command to control transaction semantics.
Use the BTEQ .SET SESSION SQLFLAG command to control use of the SQL Flagger.
Use the SQL statement SET SESSION DATEFORM to control how data typed as DATE is handled.</td></tr>
<tr><td>a new feature or feature enhancement is added to Teradata SQL and that feature is not defined by the ANSI SQL standard</td><td>that feature is designed using the following criteria:<table><tr><th>IF other vendors ...</th><th>THEN Teradata designs the new feature ...</th></tr><tr><td>offer a similar feature or feature extension</td><td>to broadly comply with other solutions, but consolidates the best ideas from all and, where necessary, creates its own, cleaner solution.</td></tr><tr><td>do not offer a similar feature or feature extension</td><td>as cleanly and generically as possible with an eye toward creating a language element that will not be subject to major revisions to comply with future updates to the ANSI SQL standard.
in a way that offers the most power to users without violating any of the basic tenets of the ANSI SQL standard.</td></tr></table></td></tr>
</table>

To find out if a specific Teradata SQL statement, option, data type, or function conforms to the ANSI SQL standard, see Appendix E: “What is New in Teradata SQL.”

Third Party Books on the ANSI SQL Standard

There are few third party books on the ANSI SQL standard, and none have yet been written about the SQL:2003 or SQL:2008 standards. The following books on various aspects of earlier ANSI SQL standards were all either written or co-written by Jim Melton, who is the editor of the ANSI SQL standard:

- Jim Melton and Alan R. Simon, *SQL:1999 Understanding Relational Language Components*, San Francisco, CA: Morgan Kaufmann, 2002.

  Provides a thorough overview of the basic components of the SQL language including statements, expressions, security, triggers, and constraints.

- Jim Melton, *Advanced SQL:1999 Understanding Object-Relational and Other Advanced Features*, San Francisco, CA: Morgan Kaufmann, 2003.

  Covers object-relational extensions to the basic SQL language including user-defined types, user-defined procedures, user-defined methods, OLAP, and SQL-related aspects of Java.
- Jim Melton, *SQL's Stored Procedures: A Complete Guide to SQL/PSM*, San Francisco, CA: Morgan Kaufmann, 1998.

  As the title says, this book provides comprehensive coverage of SQL stored procedures as of the SQL/PSM-96 standard.
- Jim Melton and Andrew Eisenberg, *Understanding SQL and Java Together: A Guide to SQLJ, JDBC, and Related Technologies*, San Francisco, CA: Morgan Kaufmann, 2000.

  Provides basic coverage of the Java language as it relates to SQL, JDBC, and other SQL-related aspects of Java.

The following book offers excellent, albeit often critical, coverage of the ANSI SQL-92 standard:

- C.J. Date with Hugh Darwen, *A Guide to the SQL Standard* (4th ed.), Reading, MA: Addison-Wesley, 1997.

Terminology Differences Between ANSI SQL and Teradata

The ANSI SQL standard and Teradata occasionally use different terminology. The following table lists the more important variances.

ANSI	Teradata
Base table	Table[1]
Binding style	Not defined, but implicitly includes the following: • Interactive SQL • Embedded SQL • ODBC • CLIv2
Authorization ID	User ID
Catalog	Dictionary
CLI	ODBC[2]
Direct SQL	Interactive SQL
Domain	Not defined
External routine function	User-defined function (UDF)

ANSI	Teradata
Module	Not defined
Persistent stored module	Stored procedure
Schema	User Database
SQL database	Relational database
Viewed table	View
Not defined	Explicit transaction[3]
Not defined	CLIv2[4]
Not defined	Macro[5]

Note:

1) In the ANSI SQL standard, the term table has the following definitions:
   - A base table
   - A viewed table (view)
   - A derived table
2) ANSI CLI is not exactly equivalent to ODBC, but the ANSI standard is heavily based on the ODBC definition.
3) ANSI transactions are always implicit, beginning with an executable SQL statement and ending with either a COMMIT or a ROLLBACK statement.
4) Teradata CLIv2 is an implementation-defined binding style.
5) 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.

SQL Flagger

Function

The SQL Flagger, when enabled, reports the use of non-standard SQL. The SQL Flagger always permits statements flagged as non-entry-level or noncompliant ANSI SQL to execute. Its task is not to enforce the standard, but rather to return a warning message to the requestor noting the noncompliance.

The analysis includes syntax checking as well as some dictionary lookup, particularly the implicit assignment and comparison of different data types (where ANSI requires use of the CAST function to convert the types explicitly) as well as some semantic checks.

The SQL Flagger does not check or detect every condition for noncompliance; thus, a statement that is not flagged does not necessarily mean it is compliant.

Enabling and Disabling the SQL Flagger

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
	See *Basic Teradata Query Reference* for more detail on using BTEQ commands.	
Preprocessor2	SQLFLAGGER(ENTRY)	to entry-level ANSI
	SQLFLAGGER(NONE)	off
	See *Teradata Preprocessor2 for Embedded SQL Programmer Guide* for details on setting Preprocessor options.	
CLI	set lang_conformance = '2' set lang_conformance to '2'	to entry-level ANSI
	set lang_conformance = 'N'	off
	See *Teradata Call-Level Interface Version 2 Reference for Channel-Attached Systems* and *Teradata Call-Level Interface Version 2 Reference for Network-Attached Systems* for details on setting the conformance field.	

Differences Between Teradata and ANSI SQL

For a complete list of SQL features in this release, see Appendix E: "What is New in Teradata SQL". The list identifies which features are ANSI SQL compliant and which features are Teradata extensions.

The list of features includes SQL statements and options, functions and operators, data types and literals.

APPENDIX E What is New in Teradata SQL

This appendix details the differences in SQL between this release and previous releases.

The intent of this appendix is to provide a way to readily identify new SQL in this release and previous releases of Teradata Database. It is not meant as a Teradata SQL reference.

Notation Conventions

The following table describes the conventions used in this appendix.

This notation ...	Means ...
UPPERCASE	a keyword
italics	a variable, such as a column or table name
[*n*]	that the use of *n* is optional
| *n* |	that option *n* is described separately in this appendix
/	that the item that follows provides an alternative

Statements and Modifiers

The table that follows lists SQL statements and modifiers for this version and previous versions of Teradata Database.

The following type codes appear in the Compliance column.

A ANSI SQL:2008 compliant

T Teradata extension

A, T ANSI SQL:2008 compliant with Teradata extensions

Statement	Compliance	13.0	12.0	V2R6.2
ABORT	T	X	X	X
FROM *option*	T	X	X	X
WHERE *condition*	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
ALTER FUNCTION ALTER SPECIFIC FUNCTION	T	X	X	X
EXECUTE PROTECTED/ EXECUTE NOT PROTECTED	T	X	X	X
COMPILE/ COMPILE ONLY	T	X	X	X
ALTER METHOD ALTER CONSTRUCTOR METHOD ALTER INSTANCE METHOD ALTER SPECIFIC METHOD	T	X	X	X
EXECUTE PROTECTED/ EXECUTE NOT PROTECTED/ COMPILE/ COMPILE ONLY	T	X	X	X
ALTER PROCEDURE (external form)	T	X	X	X
LANGUAGE C/ LANGUAGE CPP/	T	X	X	X
LANGUAGE JAVA	T	X	X	
COMPILE [ONLY]/ EXECUTE [NOT] PROTECTED	T	X	X	X
ALTER PROCEDURE (SQL form)	T	X	X	X
COMPILE	T	X	X	X
WITH PRINT/ WITH NO PRINT	T	X	X	X
WITH SPL/ WITH NO SPL	T	X	X	X
WITH WARNING/ WITH NO WARNING	T	X	X	X
ALTER REPLICATION GROUP	T	X	X	X
ADD *table_name*	T	X	X	X
DROP *table_name*	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
ALTER TABLE	A, T	X	X	X
ADD *column_name* |Data Type| |Data Type Attributes|	A	X	X	X
ADD *column_name* |Column Storage Attributes|	T	X	X	X
ADD *column_name* NO COMPRESS	T	X	X	X
ADD *column_name* |Column Constraint Attributes|	T	X	X	X
ADD *column_name* |Table Constraint Attributes|	T	X	X	X
ADD |Table Constraint Attributes|	A	X	X	X
ADD *column_name* NULL	T	X	X	X
AFTER JOURNAL/ NO AFTER JOURNAL/ DUAL AFTER JOURNAL/ LOCAL AFTER JOURNAL/ NOT LOCAL AFTER JOURNAL	T	X	X	X
BEFORE JOURNAL/ JOURNAL/ NO BEFORE JOURNAL/ DUAL BEFORE JOURNAL	T	X	X	X
DATABLOCKSIZE IMMEDIATE/ MINIMUM DATABLOCKSIZE/ MAXIMUM DATABLOCKSIZE/ DEFAULT DATABLOCKSIZE	T	X	X	X
CHECKSUM = DEFAULT/ CHECKSUM = NONE/ CHECKSUM = LOW/ CHECKSUM = MEDIUM/ CHECKSUM = HIGH/ CHECKSUM = ALL	T	X	X	X
DROP *column_name*	A	X	X	X
DROP CHECK/ DROP *column_name* CHECK/ DROP CONSTRAINT *name* CHECK	T	X	X	X
DROP CONSTRAINT	T	X	X	X
DROP FOREIGN KEY REFERENCES	T	X	X	X

Statement		Compliance	13.0	12.0	V2R6.2
ALTER TABLE, *continued*					
	WITH CHECK OPTION/ WITH NO CHECK OPTION	T	X	X	X
	DROP INCONSISTENT REFERENCES	T	X	X	X
	FALLBACK PROTECTION/ NO FALLBACK PROTECTION	T	X	X	X
	FREESPACE/ DEFAULT FREESPACE	T	X	X	X
	[NO] LOG	T	X	X	X
	MODIFY CHECK/ MODIFY *column_name* CHECK/ MODIFY CONSTRAINT *name* CHECK	T	X	X	X
	MODIFY [[NOT] UNIQUE] PRIMARY INDEX *index* [(*column*)]/ MODIFY [[NOT] UNIQUE] PRIMARY INDEX NOT NAMED [(*column*)]	T	X	X	X
	NOT PARTITIONED/ PARTITION BY *expression*/	T	X	X	X
	PARTITION BY (*expression* [… , *expression*])/		X	X	
	DROP RANGE WHERE *expression* [ADD RANGE *ranges*]/ DROP RANGE *ranges* [ADD RANGE *ranges*]/ ADD RANGE *ranges*/		X	X	X
	DROP RANGE[#L*n*] WHERE *expression* [ADD RANGE[#L*n*] *ranges*]/ DROP RANGE[#L*n*] *ranges* [ADD RANGE[#L*n*] *ranges*]/ ADD RANGE[#L*n*] *ranges*		X	X	
	WITH DELETE/ WITH INSERT [INTO] *table_name*	T	X	X	X
	ON COMMIT DELETE ROWS/ ON COMMIT PRESERVE ROWS	T	X	X	X
	RENAME *column_name*	T	X	X	X
	REVALIDATE PRIMARY INDEX/ REVALIDATE PRIMARY INDEX WITH DELETE/ REVALIDATE PRIMARY INDEX WITH INSERT [INTO] *table_name*	T	X	X	X
	WITH JOURNAL TABLE	T	X	X	X
	SET DOWN/ RESET DOWN	T	X		
ALTER TRIGGER		T	X	X	X
	ENABLED/ DISABLED/ TIMESTAMP	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
ALTER TYPE	A,T	X	X	X
ADD ATTRIBUTE/ DROP ATTRIBUTE/ ADD METHOD/ ADD INSTANCE METHOD/ ADD CONSTRUCTOR METHOD/ ADD SPECIFIC METHOD/ DROP METHOD/ DROP INSTANCE METHOD/ DROP CONSTRUCTOR METHOD/ DROP SPECIFIC METHOD	A,T	X	X	X
BEGIN DECLARE SECTION	A	X	X	X
BEGIN LOGGING	T	X	X	X
DENIALS	T	X	X	X
WITH TEXT	T	X	X	X
FIRST/ LAST/ FIRST AND LAST/ EACH	T	X	X	X
BY *database_name*	T	X	X	X
ON ALL/ ON *operation*/ ON GRANT	T	X	X	X
ON DATABASE/ ON USER/ ON TABLE/ ON VIEW/ ON MACRO/ ON PROCEDURE/ ON FUNCTION/ ON TYPE	T	X	X	X
BEGIN QUERY LOGGING	T	X	X	X
WITH NONE/	T	X		
WITH ALL/ WITH EXPLAIN/ WITH OBJECTS/ WITH SQL/ WITH STEPINFO/	T	X	X	X
WITH XMLPLAN	T	X		
LIMIT SQLTEXT [=*n*] [AND …]/ LIMIT SUMMARY = *n1*, *n2*, *n3* [AND …]/ LIMIT THRESHOLD [=*n*] [AND …]/ LIMIT MAXCPU [=*n*] [AND …]	T	X	X	X

Statement		Compliance	13.0	12.0	V2R6.2
BEGIN QUERY LOGGING, *continued*					
	ON ALL ON ALL ACCOUNT = ('*account_ID*' [… , '*account_ID*'])/ ON *user_name* [… , *user_name*]/ ON *user_name* ACCOUNT = ('*account_ID*' [… , '*account_ID*'])/	T	X	X	X
	ON APPLNAME = ('*app_name*' [… , '*app_name*'])	T	X		
BEGIN TRANSACTION/ BT		T	X	X	X
CALL		A	X	X	X
CHECKPOINT		T	X	X	X
	NAMED *checkpoint*	T	X	X	X
	INTO *host_variable_name*	T	X	X	X
	[INDICATOR] :*host_indicator_name*	T	X	X	X
CLOSE		A	X	X	X
COLLECT DEMOGRAPHICS		T	X	X	X
	FOR *table_name*/ FOR (*table_name* [… ,*table_name*])	T	X	X	X
	ALL/ WITH NO INDEX	T	X	X	X
COLLECT STATISTICS/ COLLECT STATS/ COLLECT STAT (QCD form)		T	X	X	X
	PERCENT	T	X	X	X
	SET QUERY *query_ID*	T	X	X	X
	SAMPLEID *statistics_ID*	T	X	X	X
	UPDATE MODIFIED	T	X	X	X
	INDEX (*column_list*)/ INDEX *index_name*/ COLUMN (*column_list*)/ COLUMN *column_name*	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
COLLECT STATISTICS/ COLLECT STATS/ COLLECT STAT (optimizer form)	T	X	X	X
USING SAMPLE	T	X	X	X
[ON] VOLATILE *table_name*/	T	X		
[ON] TEMPORARY *table_name*/ [ON] *table_name*/ [ON] *join_index_name*/ [ON] *hash_index_name*	T	X	X	X
INDEX (*column_list*)/ INDEX *index_name*/ COLUMN (*column_list*)/ COLUMN *column_name*	T	X	X	X
FROM [TEMPORARY] *table_name*/ FROM [TEMPORARY] *table_name* [COLUMN] (*column_list*)/ FROM [VOLATILE] *table_name*/ FROM [VOLATILE] *table_name* [COLUMN] (*column_list*)	T	X		
COLLECT STATISTICS/ COLLECT STATS/ COLLECT STAT (optimizer form, CREATE INDEX-style syntax)	T	X	X	X
USING SAMPLE	T	X	X	X
[UNIQUE] INDEX [*index_name*] [ALL] (*column_list*) [ORDER BY [VALUES]] (*column_name*)/ [UNIQUE] INDEX [*index_name*] [ALL] (*column_list*) [ORDER BY [HASH]] (*column_name*)/ COLUMN *column_name*/ COLUMN (*column_list*)	T	X	X	X
ON VOLATILE *table_name*/	T	X		
ON TEMPORARY *table_name*/ ON *table_name*/ ON *hash_index_name*/ ON *join_index_name*	T	X	X	X

Statement		Compliance	13.0	12.0	V2R6.2
COLLECT STATISTICS (optimizer form, CREATE INDEX style), *continued*					
	FROM [TEMPORARY] *table_name*/ FROM [TEMPORARY] *table_name* COLUMN *column_name*/ FROM [TEMPORARY] *table_name* COLUMN PARTITION/ FROM [TEMPORARY] *table_name* COLUMN (*column_list*)/ FROM [VOLATILE] *table_name*/ FROM [VOLATILE] *table_name* COLUMN *column_name*/ FROM [VOLATILE] *table_name* COLUMN PARTITION/ FROM [VOLATILE] *table_name* COLUMN (*column_list*)/ FROM *join_index_name*/ FROM *join_index_name* COLUMN *column_name*/ FROM *join_index_name* COLUMN PARTITION/ FROM *join_index_name* COLUMN (*column_list*)/ FROM *hash_index_name*/ FROM *hash_index_name* COLUMN *column_name*/ FROM *hash_index_name* COLUMN PARTITION/ FROM *hash_index_name* COLUMN (*column_list*)	T	X		
COMMENT		T	X	X	X
	[ON] COLUMN *object_name*/ [ON] DATABASE *object_name*/ [ON] FUNCTION *object_name*/ [ON] MACRO *object_name*/ [ON] PROCEDURE *object_name*/ [ON] TABLE *object_name*/ [ON] TRIGGER *object_name*/ [ON] USER *object_name*/ [ON] VIEW *object_name*/ [ON] PROFILE *object_name*/ [ON] ROLE *object_name*/ [ON] GROUP *group_name*/ [ON] METHOD *object_name*/ [ON] TYPE *object_name*	T	X	X	X
	AS '*comment*'/ IS '*comment*'	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
COMMENT (embedded SQL)	T	X	X	X
[ON] COLUMN *object_reference/* [ON] DATABASE *object_reference/* [ON] FUNCTION *object_name/* [ON] MACRO *object_reference/* [ON] PROCEDURE *object_reference/* [ON] TABLE *object_reference/* [ON] TRIGGER *object_reference/* [ON] USER *object_reference/* [ON] VIEW *object_reference/* [ON] PROFILE *object_name/* [ON] ROLE *object_name/* [ON] GROUP *group_name*	T	X	X	X
INTO *host_variable_name*	T	X	X	X
[INDICATOR] :*host_indicator_name*	T	X	X	X
COMMIT	A, T	X	X	X
WORK	A	X	X	X
RELEASE	T	X	X	X
CONNECT (embedded SQL)	T	X	X	X
IDENTIFIED BY *passwordvar/* IDENTIFIED BY :*passwordvar*	T	X	X	X
AS *connection_name/* AS :*namevar*	T	X	X	X
CREATE AUTHORIZATION	T	X	X	X
[AS] DEFINER [DEFAULT]/ [AS] INVOKER	T	X	X	X
DOMAIN '*domain_name*'	T	X	X	X
CREATE CAST	A	X	X	X
WITH SPECIFIC METHOD *specific_method_name/* WITH METHOD *method_name/* WITH INSTANCE METHOD *method_name/* WITH SPECIFIC FUNCTION *specific_function_name/* WITH FUNCTION *function_name*	A	X	X	X
AS ASSIGNMENT	A	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
CREATE DATABASE	T	X	X	X
PERMANENT = *n* [BYTES]	T	X	X	X
SPOOL = *n* [BYTES]	T	X	X	X
TEMPORARY = *n* [BYTES]	T	X	X	X
ACCOUNT	T	X	X	X
FALLBACK [PROTECTION]/ NO FALLBACK [PROTECTION]	T	X	X	X
[BEFORE] JOURNAL/ NO [BEFORE] JOURNAL/ DUAL [BEFORE] JOURNAL	T	X	X	X
[NO] AFTER JOURNAL/ DUAL AFTER JOURNAL/ [NOT] LOCAL AFTER JOURNAL	T	X	X	X
DEFAULT JOURNAL TABLE	T	X	X	X
CREATE ERROR TABLE	T	X	X	
CREATE FUNCTION	A, T	X	X	X
(*parameter_name data_type* […, *parameter_name data_type*])/ (*parameter_name* VARIANT_TYPE […, *parameter_name data_type*])/ (*parameter_name data_type* […, *parameter_name* VARIANT_TYPE])/ (*parameter_name* VARIANT_TYPE […, *parameter_name* VARIANT_TYPE])	A T	X X	X	X
RETURNS *data_type* [CAST FROM *data_type*]	A	X	X	X
LANGUAGE C/ LANGUAGE CPP/ LANGUAGE JAVA	A A	X X	X	X
NO SQL	A	X	X	X
SPECIFIC [*database_name.*] *function_name*	A	X	X	X
CLASS AGGREGATE/ CLASS AG	T	X	X	X
PARAMETER STYLE SQL/ PARAMETER STYLE TD_GENERAL/ PARAMETER STYLE JAVA	A A	X X	X	X
DETERMINISTIC/ NOT DETERMINISTIC	A	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
CREATE FUNCTION, *continued*				
CALLED ON NULL INPUT/ RETURNS NULL ON NULL INPUT	A	X	X	X
USING GLOP SET *GLOP_set_name*	T	X		
EXTERNAL/ EXTERNAL NAME *function_name*/ EXTERNAL NAME *function_name* PARAMETER STYLE SQL/ EXTERNAL NAME *function_name* PARAMETER STYLE TD_GENERAL/ EXTERNAL PARAMETER STYLE SQL/ EXTERNAL PARAMETER STYLE TD_GENERAL/ EXTERNAL NAME '[F *delimiter function_name*] [D] [SI *delimiter name delimiter include_name*] [CI *delimiter name delimiter include_name*] [SL *delimiter library_name*] [SO *delimiter name delimiter object_name*] [CO *delimiter name delimiter object_name*] [SP *delimiter package_name*] [SS *delimiter name delimiter source_name*] [CS *delimiter name delimiter source_name*]'/	A	X	X	X
EXTERNAL NAME '*jar_id*:*class_name.method_name*' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*class_name.method_name*(*Java_data_type* … [, *Java_data_type*]) returns *Java_data_type*' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*package_name.*[…*package_name.*]*class_name.method_name*' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*package_name.*[…*package_name.*]*class_name.method_name* (*Java_data_type* … [, *Java_data_type*]) returns *Java_data_type*' [PARAMETER STYLE JAVA]	A	X		
EXTERNAL SECURITY DEFINER/ EXTERNAL SECURITY DEFINER *authorization_name*/ EXTERNAL SECURITY INVOKER	A	X	X	X
CREATE FUNCTION (table function form)	T	X	X	X
(*parameter_name data_type* […, *parameter_name data_type*])/	A	X	X	X
(*parameter_name* VARIANT_TYPE […, *parameter_name data_type*])/ (*parameter_name data_type* […, *parameter_name* VARIANT_TYPE])/ (*parameter_name* VARIANT_TYPE […, *parameter_name* VARIANT_TYPE])	T	X		

Statement	Compliance	13.0	12.0	V2R6.2
CREATE FUNCTION (table function form), *continued*				
RETURNS TABLE (*column_name data_type* [… , *column_name data_type*])/	T	X	X	X
RETURNS TABLE VARYING COLUMNS (*max_output_columns*)		X	X	
LANGUAGE C/ LANGUAGE CPP/	T	X	X	X
LANGUAGE JAVA	T	X		
NO SQL	T	X	X	X
SPECIFIC [*database_name.*] *function_name*	T	X	X	X
PARAMETER STYLE SQL/	T	X	X	X
PARAMETER STYLE JAVA	T	X		
[NOT] DETERMINISTIC	T	X	X	X
CALLED ON NULL INPUT/ RETURNS NULL ON NULL INPUT	T	X	X	X
USING GLOP SET *GLOP_set_name*	T	X		
EXTERNAL [PARAMETER STYLE SQL]/ EXTERNAL NAME *function_name* [PARAMETER STYLE SQL]/ EXTERNAL NAME '[F *delimiter function_name*] [D] [SI *delimiter name delimiter include_name*] [CI *delimiter name delimiter include_name*] [SL *delimiter library_name*] [SO *delimiter name delimiter object_name*] [CO *delimiter name delimiter object_name*] [SP *delimiter package_name*] [SS *delimiter name delimiter source_name*] [CS *delimiter name delimiter source_name*]'/	T	X	X	X
EXTERNAL NAME '*jar_id*:*class_name.method_name*' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*class_name.method_name*(*Java_data_type* … [, *Java_data_type*])' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*package_name.*[…*package_name.*]*class_name.method_name*' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*package_name.*[…*package_name.*]*class_name.method_name* (*Java_data_type* … [, *Java_data_type*])' [PARAMETER STYLE JAVA]	T	X		
EXTERNAL SECURITY DEFINER [*authorization_name*]/ EXTERNAL SECURITY INVOKER	T	X	X	X
CREATE GLOP SET	T	X		

Statement	Compliance	13.0	12.0	V2R6.2
CREATE HASH INDEX	T	X	X	X
[NO] FALLBACK PROTECTION	T	X	X	X
ORDER BY VALUES/ ORDER BY HASH	T	X	X	X
CHECKSUM = DEFAULT/ CHECKSUM = NONE/ CHECKSUM = LOW/ CHECKSUM = MEDIUM/ CHECKSUM = HIGH/ CHECKSUM = ALL	T	X	X	X
CREATE INDEX CREATE UNIQUE INDEX	T	X	X	X
ALL	T	X	X	X
ORDER BY VALUES/ ORDER BY HASH	T	X	X	X
TEMPORARY	T	X	X	X
CREATE JOIN INDEX	T	X	X	X
[NO] FALLBACK PROTECTION	T	X	X	X
CHECKSUM = DEFAULT/ CHECKSUM = NONE/ CHECKSUM = LOW/ CHECKSUM = MEDIUM/ CHECKSUM = HIGH/ CHECKSUM = ALL	T	X	X	X
ROWID	T	X	X	X
EXTRACT YEAR FROM/ EXTRACT MONTH FROM	T	X	X	X
SUM *numeric_expression*	T	X	X	X
COUNT *column_expression*	T	X	X	X
FROM *table_name*/ FROM *table_name correlation_name*/ FROM *table_name* AS *correlation_name*	T	X	X	X
FROM (*joined_table*)	T	X	X	X
FROM *table* JOIN *table*/ FROM *table* INNER JOIN *table*/ FROM *table* LEFT JOIN *table*/ FROM *table* LEFT OUTER JOIN *table*/ FROM *table* RIGHT JOIN *table*/ FROM *table* RIGHT OUTER JOIN *table*	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
CREATE JOIN INDEX, *continued*				
|WHERE statement modifier|	A	X	X	X
|GROUP BY statement modifier|	T	X	X	X
|ORDER BY statement modifier|	A, T	X	X	X
INDEX [*index_name*] [ALL] (*column_list*)/ INDEX [*index_name*] [ALL] (*column_list*) ORDER BY HASH [(*column_name*)]/ INDEX [*index_name*] [ALL] (*column_list*) ORDER BY VALUES [(*column_name*)]/ UNIQUE INDEX [*index_name*] (*column_list*)/ PRIMARY INDEX [*index_name*] (*column_list*)/	T	X	X	X
PRIMARY INDEX [*index_name*] (*column_list*) PARTITION BY *expression*/	T	X	X	X
PRIMARY INDEX [*index_name*] (*column_list*) PARTITION BY (*expression* […, *expression*])	T	X	X	
CREATE MACRO/ CM	T	X	X	X
CREATE METHOD CREATE INSTANCE METHOD CREATE CONSTRUCTOR METHOD	A	X	X	X
(*parameter_name data_type* […, *parameter_name data_type*])	A	X	X	X
RETURNS *data_type* [CAST FROM *data_type*]	A	X	X	X
USING GLOP SET *GLOP_set_name*	T	X		
EXTERNAL/ EXTERNAL NAME *method_name*/ EXTERNAL NAME '[F *delimiter function_entry_name*] [D] [SI *delimiter name delimiter include_name*] [CI *delimiter name delimiter include_name*] [SL *delimiter library_name*] [SO *delimiter name delimiter object_name*] [CO *delimiter name delimiter object_name*] [SP *delimiter package_name*] [SS *delimiter name delimiter source_name*] [CS *delimiter name delimiter source_name*]'	A	X	X	X
EXTERNAL SECURITY DEFINER/ EXTERNAL SECURITY DEFINER *authorization_name*/ EXTERNAL SECURITY INVOKER	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
CREATE ORDERING	A	X	X	X
MAP WITH [SPECIFIC] METHOD *method_name*/ MAP WITH INSTANCE METHOD *method_name*/ MAP WITH [SPECIFIC] FUNCTION *function_name*	A	X	X	X
CREATE PROCEDURE (external stored procedure form)	A	X	X	X
LANGUAGE C/ LANGUAGE CPP/	A	X	X	X
LANGUAGE JAVA	A	X	X	
NO SQL/	A	X	X	X
CONTAINS SQL/ READS SQL DATA/ MODIFIES SQL DATA	A	X	X	
DYNAMIC RESULT SETS *number_of_sets*	A	X	X	
PARAMETER STYLE SQL/ PARAMETER STYLE TD_GENERAL/	A	X	X	X
PARAMETER STYLE JAVA	A	X	X	
USING GLOP SET *GLOP_set_name*	T	X		
EXTERNAL/ EXTERNAL NAME *procedure_name*/ EXTERNAL NAME *procedure_name* PARAMETER STYLE SQL/ EXTERNAL NAME *procedure_name* PARAMETER STYLE TD_GENERAL/ EXTERNAL PARAMETER STYLE SQL/ EXTERNAL PARAMETER STYLE TD_GENERAL/ EXTERNAL NAME '[F *delimiter function_entry_name*] [D] [SI *delimiter name delimiter include_name*] [CI *delimiter name delimiter include_name*] [SL *delimiter library_name*] [SO *delimiter name delimiter object_name*] [CO *delimiter name delimiter object_name*] [SS *delimiter name delimiter source_name*] [CS *delimiter name delimiter source_name*] [SP *delimiter package_name*] [SP *delimiter* CLI]'/	A	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
CREATE PROCEDURE, *continued*				
EXTERNAL NAME '*jar_id*:*class_name.method_name*' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*class_name.method_name*(*Java_data_type* … [, *Java_data_type*])' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*package_name.*[…*package_name.*]*class_name.method_name*' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*package_name.*[…*package_name.*]*class_name.method_name* (*Java_data_type* … [, *Java_data_type*]) [PARAMETER STYLE JAVA]'	A	X	X	
SQL SECURITY OWNER/ SQL SECURITY CREATOR/ SQL SECURITY DEFINER/ SQL SECURITY INVOKER	T	X		
EXTERNAL SECURITY DEFINER/ EXTERNAL SECURITY DEFINER *authorization_name*/ EXTERNAL SECURITY INVOKER	A	X	X	X
CREATE PROCEDURE (stored procedure form)	A, T	X	X	X
parameter_name data_type/ IN *parameter_name data_type*/ OUT *parameter_name data_type*/ INOUT *parameter_name data_type*	A	X	X	X
DYNAMIC RESULT SETS *number_of_sets*	A	X	X	
CONTAINS SQL/ READS SQL DATA/ MODIFIES SQL DATA	A	X	X	
SQL SECURITY OWNER/ SQL SECURITY CREATOR/ SQL SECURITY DEFINER/ SQL SECURITY INVOKER	T	X		
DECLARE *variable-name data-type* [DEFAULT *literal*]/ DECLARE *variable-name data-type* [DEFAULT NULL]	A	X	X	X
DECLARE *condition_name* CONDITION [FOR *sqlstate_value*]	A	X		

Statement	Compliance	13.0	12.0	V2R6.2
CREATE PROCEDURE, *continued*				
DECLARE *cursor_name* [SCROLL] CURSOR/ DECLARE *cursor_name* [NO SCROLL] CURSOR	A	X	X	X
WITHOUT RETURN/ WITH RETURN ONLY [TO CALLER]/ WITH RETURN ONLY TO CLIENT/	T	X	X	
WITH RETURN [TO CALLER]/ WITH RETURN TO CLIENT	T	X		
FOR *cursor_specification* FOR READ ONLY/ FOR *cursor_specification* FOR UPDATE/	A	X	X	X
FOR *statement_name*	T	X	X	
DECLARE CONTINUE HANDLER FOR DECLARE EXIT HANDLER FOR	A	X	X	X
SQLSTATE [VALUE] *sqlstate*/ SQLEXCEPTION/ SQLWARNING/ NOT FOUND/	A	X	X	X
condition_name	A	X		
SET *assignment_target* = *assignment_source*	A	X	X	X
IF *expression* THEN *statement* [ELSEIF *expression* THEN *statement*] [ELSE *statement*] END IF	A	X	X	X
CASE *operand1* WHEN *operand2* THEN *statement* [ELSE *statement*] END CASE	A	X	X	X
CASE WHEN *expression* THEN *statement* [ELSE *statement*] END CASE	A	X	X	X
ITERATE *label_name*	A	X	X	X
LEAVE *label_name*	A	X	X	X
SQL_statement	A	X	X	X
CALL *procedure_name*	A	X	X	X
PREPARE *statement_name* FROM *statement_variable*	A	X	X	
OPEN *cursor_name*	A	X	X	X
OPEN *cursor_name* USING *using_argument* [… , *using_argument*]	T	X	X	
CLOSE *cursor_name*	A	X	X	X

Statement		Compliance	13.0	12.0	V2R6.2
CREATE PROCEDURE, *continued*					
	FETCH [[NEXT] FROM] *cursor_name* INTO *local_variable_name* [… , *local_variable_name*]/ FETCH [[FIRST] FROM] *cursor_name* INTO *local_variable_name* [… , *local_variable_name*]/ FETCH [[NEXT] FROM] *cursor_name* INTO *parameter_reference* [… , *parameter_reference*]/ FETCH [[FIRST] FROM] *cursor_name* INTO *parameter_reference* [… , *parameter_reference*]	A	X	X	X
	WHILE *expression* DO *statement* END WHILE	A	X	X	X
	LOOP *statement* END LOOP	A	X	X	X
	FOR *for_loop_variable* AS [*cursor_name* CURSOR FOR] SELECT *expression* [AS *correlation_name*] FROM *table_name* [WHERE *clause*] [SELECT *clause*] DO *statement_list* END FOR	A	X	X	X
	REPEAT *statement_list* UNTIL *conditional_expression* END REPEAT	A	X	X	X
	GET DIAGNOSTICS *SQL-diagnostics-information*	A	X		
	RESIGNAL [*signal_value*] [*set_signal_information*]	A	X		
	SIGNAL *signal_value* [*set_signal_information*]	A	X		
CREATE PROFILE		T	X	X	X
	ACCOUNT = '*account_id*'/ ACCOUNT = ('*account_id*' [… ,'*account_id*'])/ ACCOUNT = NULL	T	X	X	X
	DEFAULT DATABASE = *database_name*/ DEFAULT DATABASE = NULL	T	X	X	X
	COST PROFILE = *cost_profile_name*/ COST PROFILE = NULL	T	X	X	
	SPOOL = *n* [BYTES]/ SPOOL = NULL	T	X	X	X
	TEMPORARY = *n* [BYTES]/ TEMPORARY = NULL	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
CREATE PROFILE, *continued*				
PASSWORD [ATTRIBUTES] = NULL/	T	X	X	X
PASSWORD [ATTRIBUTES] = ([EXPIRE = *n*/ EXPIRE = NULL] [[,] MINCHAR = *n*/ MINCHAR = NULL] [[,] MAXCHAR = *n*/ MAXCHAR = NULL] [[,] DIGITS = *n*/ DIGITS = NULL] [[,] SPECCHAR = *c*/ SPECCHAR = NULL] [[,] MAXLOGONATTEMPTS = *n*/ MAXLOGONATTEMPTS = NULL] [[,] LOCKEDUSEREXPIRE = *n*/ LOCKEDUSEREXPIRE = NULL] [[,] REUSE = *n*/ REUSE = NULL]				
[[,] RESTRICTWORDS = *c*/ RESTRICTWORDS = NULL])	T	X	X	
CREATE REPLICATION GROUP	T	X	X	X
CREATE REPLICATION RULESET	T	X		
CREATE ROLE	A	X	X	X
CREATE TABLE/ CT	A, T	X	X	X
SET/ MULTISET	T	X	X	X
GLOBAL TEMPORARY	A	X	X	X
GLOBAL TEMPORARY TRACE	T	X	X	X
VOLATILE	T	X	X	X
QUEUE	T	X	X	X
[NO] FALLBACK [PROTECTION]	T	X	X	X
WITH JOURNAL TABLE = *name*	T	X	X	X
[NO] LOG	T	X	X	X
[BEFORE] JOURNAL/ NO [BEFORE] JOURNAL/ DUAL [BEFORE] JOURNAL/	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
CREATE TABLE, *continued*				
[NO] AFTER JOURNAL/ DUAL AFTER JOURNAL/ [NOT] LOCAL JOURNAL	T	X	X	X
FREESPACE = *integer* PERCENT	T	X	X	X
DATABLOCKSIZE = *integer*/ DATABLOCKSIZE = *integer* BYTES/ DATABLOCKSIZE = *integer* KBYTES/ DATABLOCKSIZE = *integer* KILOBYTES	T	X	X	X
MINIMUM DATABLOCKSIZE/ MAXIMUM DATABLOCKSIZE	T	X	X	X
CHECKSUM = DEFAULT/ CHECKSUM = NONE/ CHECKSUM = LOW/ CHECKSUM = MEDIUM/ CHECKSUM = HIGH/ CHECKSUM = ALL	T	X	X	X
REPLICATION GROUP *group_name*	T	X		
column_name |Data Type| |Data Type Attributes|	A	X	X	X
column_name |Data Type| |Column Storage Attributes|	T	X	X	X
column_name |Data Type| |Column Constraint Attributes|	A	X	X	X
GENERATED ALWAYS AS IDENTITY/ GENERATED BY DEFAULT AS IDENTITY	A	X	X	X
|Column Constraint Attributes|	T	X	X	X
|Table Constraint Attributes|	T	X	X	X
[UNIQUE] INDEX [*name*] [ALL] (*column_name* […, *column_name*])	T	X	X	X
[UNIQUE] PRIMARY INDEX [*name*] (*column_name* […, *column_name*])/	T	X	X	X
[UNIQUE] PRIMARY INDEX [*name*] (*column_name* […, *column_name*]) PARTITION BY *expression*/	T	X	X	X
[UNIQUE] PRIMARY INDEX [*name*] (*column_name* […, *column_name*]) PARTITION BY (*expression* […, *expression*])	T	X	X	
NO PRIMARY INDEX	T	X		

Statement	Compliance	13.0	12.0	V2R6.2
CREATE TABLE, *continued*				
INDEX [*name*] [ALL] (*column_name* […, *column_name*]) ORDER BY VALUES (*name*)/ INDEX [*name*] [ALL] (*column_name* […, *column_name*]) ORDER BY HASH (*name*)	T	X	X	X
ON COMMIT DELETE ROWS/ ON COMMIT PRESERVE ROWS	A	X	X	X
AS *source_table_name* WITH [NO] DATA/	A	X	X	X
AS *source_table_name* WITH [NO] DATA AND [NO] STATISTICS/ AS *source_table_name* WITH [NO] DATA AND [NO] STATS/ AS *source_table_name* WITH [NO] DATA AND [NO] STAT/	T	X	X	X
AS (*query_expression*) WITH [NO] DATA/	A	X	X	X
AS (*query_expression*) WITH [NO] DATA AND [NO] STATISTICS/ AS (*query_expression*) WITH [NO] DATA AND [NO] STATS/ AS (*query_expression*) WITH [NO] DATA AND [NO] STAT	T	X	X	X
CREATE TRANSFORM	A	X	X	X
TO SQL WITH [SPECIFIC] METHOD *method_name*/ TO SQL WITH INSTANCE METHOD *method_name*/ TO SQL WITH [SPECIFIC] FUNCTION *function_name*	A	X	X	X
FROM SQL WITH [SPECIFIC] METHOD *method_name*/ FROM SQL WITH INSTANCE METHOD *method_name*/ FROM SQL WITH [SPECIFIC] FUNCTION *function_name*	A	X	X	X
CREATE TRIGGER	A, T	X	X	X
ENABLED/ DISABLED	T	X	X	X
BEFORE/ AFTER	A	X	X	X
INSERT ON *table_name* [ORDER *integer*]/ DELETE ON *table_name* [ORDER *integer*]/ UPDATE [OF (*column_list*)] ON *table_name* [ORDER *integer*]	A	X	X	X
REFERENCING OLD_TABLE [AS] *identifier* [NEW_TABLE [AS] *identifier*]/	T	X	X	X
REFERENCING OLD_NEW_TABLE [AS] *transition_table_name* (*old_transition_variable_name*, *new_transition_variable_name*)/	T	X		
REFERENCING OLD [AS] *identifier* [NEW [AS] *identifier*]/ REFERENCING OLD TABLE [AS] *identifier* [NEW TABLE [AS] *identifier*]/ REFERENCING OLD [ROW] [AS] *identifier* [NEW [ROW] [AS] *identifier*]	A	X	X	X

Statement		Compliance	13.0	12.0	V2R6.2
CREATE TRIGGER, *continued*					
	FOR EACH ROW/ FOR EACH STATEMENT	A	X	X	X
	WHEN (*search_condition*)	A	X	X	X
	(*SQL_proc_statement* ;)/ *SQL_proc_statement* / BEGIN ATOMIC (*SQL_proc_statement*;) END/ BEGIN ATOMIC *SQL_proc_statement* ; END	A,T	X	X	X
CREATE TYPE (distinct form)		A, T	X	X	X
	METHOD [SYSUDTLIB.]*method_name*/ INSTANCE METHOD [SYSUDTLIB.]*method_name*	A, T	X	X	X
	RETURNS *predefined_data_type* [AS LOCATOR]/ RETURNS *predefined_data_type* [AS LOCATOR] CAST FROM *predefined_data_type* [AS LOCATOR]/ RETURNS *predefined_data_type* CAST FROM [SYSUDTLIB.]*UDT_name* [AS LOCATOR]/ RETURNS [SYSUDTLIB.]*UDT_name* [AS LOCATOR]/ RETURNS [SYSUDTLIB.]*UDT_name* [AS LOCATOR] CAST FROM *predefined_data_type* [AS LOCATOR]/ RETURNS [SYSUDTLIB.]*UDT_name* CAST FROM [SYSUDTLIB.]*UDT_name* [AS LOCATOR]	A, T	X	X	X
	LANGUAGE C/ LANGUAGE CPP	A	X	X	X
	NO SQL	A	X	X	X
	SPECIFIC [SYSUDTLIB.] *specific_method_name*	A, T	X	X	X
	SELF AS RESULT	A	X	X	X
	PARAMETER STYLE SQL/ PARAMETER STYLE TD_GENERAL	A	X	X	X
	[NOT] DETERMINISTIC	A	X	X	X
	CALLED ON NULL INPUT/ RETURNS NULL ON NULL INPUT	A	X	X	X
CREATE TYPE (structured form)		A, T	X	X	X
	INSTANTIABLE	A	X	X	X
	METHOD [SYSUDTLIB.]*method_name*/ INSTANCE METHOD [SYSUDTLIB.]*method_name* CONSTRUCTOR METHOD [SYSUDTLIB.]*method_name*	A, T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
CREATE TYPE (structured form), *continued*				
RETURNS *predefined_data_type* [AS LOCATOR]/ RETURNS *predefined_data_type* [AS LOCATOR] CAST FROM *predefined_data_type* [AS LOCATOR]/ RETURNS *predefined_data_type* CAST FROM [SYSUDTLIB.]*UDT_name* [AS LOCATOR]/ RETURNS [SYSUDTLIB.]*UDT_name*/ RETURNS [SYSUDTLIB.]*UDT_name* AS LOCATOR/ RETURNS [SYSUDTLIB.]*UDT_name* [AS LOCATOR] CAST FROM *predefined_data_type* [AS LOCATOR]/ RETURNS [SYSUDTLIB.]*UDT_name* CAST FROM [SYSUDTLIB.]*UDT_name* [AS LOCATOR]	A, T	X	X	X
LANGUAGE C/ LANGUAGE CPP	A	X	X	X
NO SQL	A	X	X	X
SPECIFIC [SYSUDTLIB.] *specific_method_name*	A, T	X	X	X
SELF AS RESULT	A	X	X	X
PARAMETER STYLE SQL/ PARAMETER STYLE TD_GENERAL	A	X	X	X
DETERMINISTIC/ NOT DETERMINISTIC	A	X	X	X
CALLED ON NULL INPUT/ RETURNS NULL ON NULL INPUT	A	X	X	X
CREATE USER	T	X	X	X
FROM *database_name*	T	X	X	X
PERMANENT = *number* [BYTES]/ PERM = *number* [BYTES]	T	X	X	X
PASSWORD = *password*/ PASSWORD = NULL	T	X	X	X
STARTUP = '*string*;'	T	X	X	X
TEMPORARY = *n* [bytes]	T	X	X	X
SPOOL = *n* [BYTES]	T	X	X	X
DEFAULT DATABASE = *database_name*	T	X	X	X
COLLATION = *collation_sequence*	T	X	X	X
ACCOUNT = '*acct_ID*'/ ACCOUNT = ('*acct_ID*' [… ,'*acct_ID*'])	T	X	X	X
[NO] FALLBACK [PROTECTION]	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
CREATE USER, *continued*				
[BEFORE] JOURNAL/ NO [BEFORE] JOURNAL/ DUAL [BEFORE] JOURNAL	T	X	X	X
[NO] AFTER JOURNAL/ DUAL AFTER JOURNAL/ [NOT] LOCAL AFTER JOURNAL	T	X	X	X
DEFAULT JOURNAL TABLE = *table_name*	T	X	X	X
TIME ZONE = LOCAL/ TIME ZONE = [*sign*] *quotestring*/ TIME ZONE = NULL	T	X	X	X
DATEFORM = INTEGERDATE/ DATEFORM = ANSIDATE	T	X	X	X
DEFAULT CHARACTER SET *data_type*	T	X	X	X
DEFAULT ROLE = *role_name*/ DEFAULT ROLE = NONE/ DEFAULT ROLE = NULL/ DEFAULT ROLE = ALL	T	X	X	X
PROFILE = *profile_name*/ PROFILE = NULL	T	X	X	X
CREATE VIEW	A, T	X	X	X
(*column_name* [… , *column_name*])	A	X	X	X
AS [|LOCKING statement modifier|] *query_expression*	A, T	X	X	X
WITH CHECK OPTION	A	X	X	X
CREATE RECURSIVE VIEW	A	X	X	X
(*column_name* [… , *column_name*])	A	X	X	X
AS *(seed_statement* [UNION ALL *recursive_statement*)] [… [UNION ALL *seed_statement*] [… UNION ALL *recursive_statement*])	A	X	X	X
DATABASE	T	X	X	X
DECLARE CURSOR (selection form)	A, T	X	X	X
FOR SELECT	A	X	X	X
FOR COMMENT/ FOR EXPLAIN/ FOR HELP/ FOR SHOW	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
DECLARE CURSOR (request form)	A	X	X	X
FOR '*request_specification*'	A	X	X	X
DECLARE CURSOR (macro form)	T	X	X	X
FOR EXEC *macro_name*	T	X	X	X
DECLARE CURSOR (dynamic SQL form)	A	X	X	X
FOR *statement_name*	A	X	X	X
DECLARE STATEMENT	T	X	X	X
DECLARE TABLE	T	X	X	X
DELETE (basic/searched form)/ DEL	A, T	X	X	X
[FROM] *table_name*	A	X	X	X
[AS] *alias_name*	A	X	X	X
WHERE *condition*	A	X	X	X
ALL	T	X	X	X
DELETE (implied join condition form)/ DEL	A, T	X	X	X
delete_table_name	T	X	X	X
[FROM] *table_name* [… ,[FROM] *table_name*]	T	X	X	X
[AS] *alias_name*	A	X	X	X
WHERE *condition*	A	X	X	X
ALL	T	X	X	X
DELETE (positioned form)/ DEL	A	X	X	X
FROM *table_name*	A	X	X	X
WHERE CURRENT OF *cursor_name*	A	X	X	X
DELETE DATABASE DELETE USER	T	X	X	X
ALL	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
DESCRIBE	T	X	X	X
INTO *descriptor_area*	T	X	X	X
USING NAMES/ USING ANY/ USING BOTH/ USING LABELS	T	X	X	X
FOR STATEMENT *statement_number*/ FOR STATEMENT [:] *num_var*	T	X	X	X
DIAGNOSTIC "*validate index*"	T	X	X	X
ON/ NOT ON	T	X	X	X
DIAGNOSTIC COSTPRINT	T	X	X	
DIAGNOSTIC DUMP COSTS	T	X	X	
DIAGNOSTIC DUMP SAMPLES	T	X	X	X
DIAGNOSTIC HELP COSTS	T	X	X	
DIAGNOSTIC HELP PROFILE	T	X	X	
DIAGNOSTIC HELP SAMPLES	T	X	X	X
DIAGNOSTIC SET COSTS	T	X	X	
DIAGNOSTIC SET PROFILE	T	X	X	
DIAGNOSTIC SET SAMPLES	T	X	X	X
ON/ NOT ON	T	X	X	X
FOR SESSION/ FOR SYSTEM	T	X	X	X
DROP AUTHORIZATION	T	X	X	X
DROP CAST	A	X	X	X
DROP DATABASE DROP USER	T	X	X	X
DROP ERROR TABLE	T	X	X	
DROP FUNCTION DROP SPECIFIC FUNCTION	A	X	X	X
DROP GLOP SET	T	X		
DROP HASH INDEX	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
DROP INDEX	T	X	X	X
TEMPORARY	T	X	X	X
ORDER BY (*column_name*)/ ORDER BY VALUES (*column_name*)/ ORDER BY HASH (*column_name*)	T	X	X	X
DROP JOIN INDEX	T	X	X	X
DROP MACRO	T	X	X	X
DROP ORDERING	A	X	X	X
DROP PROCEDURE	A	X	X	X
DROP PROFILE	T	X	X	X
DROP REPLICATION GROUP	T	X	X	X
DROP REPLICATION RULESET	T	X		
DROP ROLE	A	X	X	X
DROP STATISTICS/ DROP STATS/ DROP STAT (optimizer form)	T	X	X	X
[FOR] [UNIQUE] INDEX *index_name*/ [FOR] [UNIQUE] INDEX [*index_name*] (*col_name*) [ORDER BY *col_name*]/ [FOR] [UNIQUE] INDEX [*index_name*] (*col_name*) [ORDER BY VALUES (*col_name*)]/ [FOR] [UNIQUE] INDEX [*index_name*] (*col_name*) [ORDER BY HASH (*col_name*)]/	T	X	X	X
[FOR] COLUMN *column_name*/ [FOR] COLUMN (*column_name* [… , *column_name*])/				
[FOR] COLUMN (*column_name* [… , *column_name*], PARTITION [… , *column_name*])/ [FOR] COLUMN (PARTITION [… , *column_name*])/ [FOR] COLUMN PARTITION	T	X	X	X
ON	T	X	X	X
TEMPORARY	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
DROP STATISTICS/ DROP STATS/ DROP STAT (QCD form)	T	X	X	X
INDEX (*column_name* [… , *column_name*])/ INDEX *index_name*/	T	X	X	X
COLUMN (*column_name* [… ,*column_name*])/ COLUMN *column_name*/				
COLUMN (*column_name [… , column_name]*, PARTITION [… , *column_name*])/ COLUMN (PARTITION [… , *column_name*])/ COLUMN PARTITION	T	X	X	X
DROP TABLE	A, T	X	X	X
TEMPORARY	A	X	X	X
ALL	A	X	X	X
OVERRIDE	A	X	X	X
DROP TRANSFORM	A	X	X	X
DROP TRIGGER	T	X	X	X
DROP TYPE	A	X	X	X
DROP VIEW	T	X	X	X
DUMP EXPLAIN	T	X	X	X
AS *query_plan_name*	T	X	X	X
LIMIT/ LIMIT SQL/ LIMIT SQL = *n*	T	X	X	X
CHECK STATISTICS	T	X	X	X
ECHO	T	X	X	X
END DECLARE SECTION	T	X	X	X
END-EXEC	A	X	X	X
END LOGGING	T	X	X	X
DENIALS	T	X	X	X
WITH TEXT	T	X	X	X
ALL/ *operation*/ GRANT	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
END LOGGING, *continued*				
BY *database_name*	T	X	X	X
ON DATABASE *name*/ ON FUNCTION/ ON MACRO *name*/ ON PROCEDURE *name*/ ON TABLE *name*/ ON TRIGGER *name*/ ON USER *name*/ ON VIEW *name*	T	X	X	X
END QUERY LOGGING	T	X	X	X
ON ALL/ ON ALL ACCOUNT = ('*account_ID*' [… ,'*account_ID*'])/ ON *user_name* [… , *user_name*])/ ON *user_name* ACCOUNT = ('*account_ID*' [… , '*account_ID*'])/	T	X	X	X
ON ALL RULES/ ON APPLNAME = ('*app_name*' [… , '*app_name*'])	T	X		
END TRANSACTION/ ET	T	X	X	X
EXECUTE *macro_name*/ EXEC *macro_name*	T	X	X	X
EXECUTE *statement_name*	A	X	X	X
USING [:] *host_variable_name*	A	X	X	X
[INDICATOR] :*host_indicator_name*	A	X	X	X
USING DESCRIPTOR [:] *descriptor_area*	A	X	X	X
EXECUTE IMMEDIATE	A	X	X	X
FETCH	A	X	X	X
INTO [:] *host_variable_name*	A	X	X	X
[INDICATOR] :*host_indicator_name*	A	X	X	X
USING DESCRIPTOR [:] *descriptor_area*	A	X	X	X
GET CRASH (embedded SQL)	T	X	X	X
GIVE	T	X	X	X
database_name TO *recipient_name*/ *user_name* TO *recipient_name*	T	X	X	X

Statement		Compliance	13.0	12.0	V2R6.2
GRANT		A, T	X	X	X
	ALL/	A	X	X	X
	ALL PRIVILEGES/ ALL BUT	T	X	X	X
	DELETE/ EXECUTE/ INSERT/ REFERENCES/ REFERENCES (*column_list*)/ SELECT/ TRIGGER/ UPDATE/ UPDATE (*column_list*)/	A	X	X	X
	INSERT (*column_list*)/ SELECT (*column_list*)/	A	X		
	ALTER/ AUTHORIZATION/ CHECKPOINT/ CREATE/ DROP/ DUMP/ INDEX/ RESTORE/ REPLCONTROL/ UDTMETHOD/ UDTTYPE/ UDTUSAGE/	T	X	X	X
	CREATE OWNER PROCEDURE/ CTCONTROL/ GLOP/ SHOW/ STATISTICS	T	X		
	ON *database_name*/ ON *database_name.object_name*/ ON *object_name*/ ON PROCEDURE *identifier*/ ON SPECIFIC FUNCTION *specific_function_name*/ ON FUNCTION *function_name*/	A	X	X	X
	ON TYPE *UDT_name*/ ON TYPE SYSUDTLIB.*UDT_name*	A	X	X	X
	TO *user_name*/ TO ALL *user_name*/	T	X	X	X
	TO PUBLIC	A	X	X	X
	WITH GRANT OPTION	A	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
GRANT CONNECT THROUGH	T	X		
TO *app_user_name* [... , *app_user_name*] WITH ROLE *role_name* [... , *role_name*]/ TO PERMANENT *perm_user_name* [... , *perm_user_name*] WITHOUT ROLE/ TO PERMANENT *perm_user_name* [... , *perm_user_name*] WITH ROLE *role_name* [... , *role_name*]	T	X		
GRANT LOGON	T	X	X	X
ON *host_id*/ ON ALL	T	X	X	X
AS DEFAULT/ TO *database_name*/ FROM *database_name*	T	X	X	X
WITH NULL PASSWORD	T	X	X	X
GRANT MONITOR/ GRANT *monitor_privilege*	T	X	X	X
PRIVILEGES/ BUT NOT *monitor_privilege*	T	X	X	X
TO [ALL] *user_name*/ TO PUBLIC	T	X	X	X
WITH GRANT OPTION	T	X	X	X
GRANT ROLE	A	X	X	X
WITH ADMIN OPTION	A	X	X	X
HELP	T	X	X	X
CAST [*database_name.*] *UDT_name*/ CAST [*database_name.*] *UDT_name* SOURCE/ CAST [*database_name.*] *UDT_name* TARGET	T	X	X	X
COLUMN *column_name* FROM *table_name*/ COLUMN * FROM *table_name*/ COLUMN *table_name.column_name*/ COLUMN *table_name.**/ COLUMN *expression*/	T	X	X	X
COLUMN *column_name* FROM ERROR TABLE FOR *table_name*	T	X	X	
CONSTRAINT [*database_name.*] *table_name.name*	T	X	X	X
DATABASE *database_name*	T	X	X	X
ERROR TABLE FOR *data_table_name*	T	X	X	
FUNCTION *function_name* [(*data_type* [… , *data_type*])]/ SPECIFIC FUNCTION *specific_function_name*	T	X	X	X

Statement		Compliance	13.0	12.0	V2R6.2
HELP, *continued*					
	HASH INDEX *hash_index_name*	T	X	X	X
	[TEMPORARY] INDEX *table_name* [(*column_name*)]/ [TEMPORARY] INDEX *join_index_name* [(*column_name*)]	T	X	X	X
	JOIN INDEX *join_index_name*	T	X	X	X
	MACRO *macro_name*	T	X	X	X
	METHOD [*database_name.*] *method_name*/ INSTANCE METHOD [*database_name.*] *method_name*/ CONSTRUCTOR METHOD [*database_name.*] *method_name*/ SPECIFIC METHOD [*database_name.*] *specific_method_name*	T	X	X	X
	PROCEDURE [*database_name.*] *procedure_name*/ PROCEDURE [*database_name.*] *procedure_name* ATTRIBUTES/ PROCEDURE [*database_name.*] *procedure_name* ATTR/ PROCEDURE [*database_name.*] *procedure_name* ATTRS	T	X	X	X
	REPLICATION GROUP	T	X	X	X
	SESSION	T	X	X	X
	TABLE *table_name*/ TABLE *join_index_name*	T	X	X	X
	TRANSFORM [*database_name.*] *UDT_name*	T	X	X	X
	TRIGGER [*database_name.*] *trigger_name*/ TRIGGER [*database_name.*] *table_name*	T	X	X	X
	TYPE [*database_name.*] *UDT_name*/ TYPE [*database_name.*] *UDT_name* ATTRIBUTE/ TYPE [*database_name.*] *UDT_name* METHOD	T	X	X	X
	USER *user_name*	T	X	X	X
	VIEW *view_name*	T	X	X	X
	VOLATILE TABLE	T	X	X	X
HELP STATISTICS/ HELP STATS/ HELP STAT (optimizer form)		T	X	X	X
	INDEX (*column_name* [… , *column_name*])/ INDEX *index_name*/ COLUMN (*column_name* [… ,*column_name*])/ COLUMN *column_name*/ COLUMN (*column_name [… , column_name]*, PARTITION [… , *column_name*])/ COLUMN (PARTITION [… , *column_name*])/ COLUMN PARTITION	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
HELP STATISTICS/ HELP STATS/ HELP STAT (QCD form)	T	X	X	X
INDEX (*column_name* [… , *column_name*])/ INDEX *index_name*/ COLUMN (*column_name* [… ,*column_name*])/ COLUMN *column_name*/ COLUMN (*column_name [… , column_name]*, PARTITION [… , *column_name*])/ COLUMN (PARTITION [… , *column_name*])/ COLUMN PARTITION	T	X	X	X
FOR QUERY *query_ID*	T	X	X	X
SAMPLEID *statistics_ID*	T	X	X	X
UPDATE MODIFIED	T	X	X	X
INCLUDE	A	X	X	X
INCLUDE SQLCA	T	X	X	X
INCLUDE SQLDA	T	X	X	X
INITIATE INDEX ANALYSIS	T	X	X	X
ON *table_name* [… , *table_name*]	T	X	X	X
SET IndexesPerTable = *value* [, SearchSpace = *value*] [, ChangeRate = *value*] [, ColumnsPerIndex = *value*] [, ColumnsPerJoinIndex = *value*] [, IndexMaintMode = *value*]	T	X	X	X
KEEP INDEX	T	X	X	X
USE MODIFIED STATISTICS/ USE MODIFIED STATS/ USE MODIFIED STAT	T	X	X	X
WITH INDEX TYPE *number* [… , *number*]/ WITH NO INDEX TYPE *number* [… , *number*]	T	X	X	X
CHECKPOINT *checkpoint_trigger*/	T	X	X	X
TIME LIMIT = *elapsed_time*/	T	X	X	
CHECKPOINT *checkpoint_trigger* TIME LIMIT = *elapsed_time*	T	X		
INITIATE PARTITION ANALYSIS	T	X	X	
ON *table_name* [… , *table_name*]	T	X	X	
TIME LIMIT = *elapsed_time*	T	X	X	

Statement	Compliance	13.0	12.0	V2R6.2
INSERT/ INS	A T	X	X	X
[VALUES] (*expression* [… , *expression*])	A	X	X	X
(*column_name* [… , *column_name*]) VALUES (*expression* [… , *expression*])	A	X	X	X
[(*column_name* [… , *column_name*])] *subquery*	A	X	X	X
LOGGING [ALL] ERRORS/ LOGGING [ALL] ERRORS WITH NO LIMIT/ LOGGING [ALL] ERRORS WITH LIMIT OF *error_limit*	T	X	X	
DEFAULT VALUES	A	X	X	X
INSERT EXPLAIN	T	X	X	X
WITH STATISTICS [AND DEMOGRAPHICS]/ WITH NO STATISTICS [AND DEMOGRAPHICS]	T	X	X	X
USING SAMPLE *percentage*/ USING SAMPLE *percentage* PERCENT	T	X	X	X
FOR *table_name* [… , *table_name*]	T	X	X	X
AS *query_plan_name*	T	X	X	X
LIMIT/ LIMIT SQL/ LIMIT SQL = *n*	T	X	X	X
FOR *frequency*	T	X	X	X
CHECK STATISTICS	T	X	X	X
IN XML/ IN XML COMPRESS/ IN XML NODDLTEXT/ IN XML COMPRESS NODDLTEXT	T	X		
LOGGING ONLINE ARCHIVE OFF	T	X	X	
OVERRIDE	T	X	X	
LOGGING ONLINE ARCHIVE ON	T	X	X	
LOGOFF (embedded SQL)	T	X	X	X
CURRENT/ ALL/ *connection_name*/ :*host_variable_name*	T	X	X	X
LOGON (embedded SQL)	T	X	X	X
AS *connection_name*/ AS :*namevar*	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
MERGE	A	X	X	X
INTO	A	X	X	X
AS *correlation_name*	A	X	X	X
VALUES *using_expression*/ (*subquery*)	A	X	X	X
ON *match_condition*	A	X	X	X
WHEN MATCHED THEN UPDATE SET	A	X	X	X
WHEN NOT MATCHED THEN INSERT	A	X	X	X
LOGGING [ALL] ERRORS/ LOGGING [ALL] ERRORS WITH NO LIMIT/ LOGGING [ALL] ERRORS WITH LIMIT OF *error_limit*	T	X	X	
MODIFY DATABASE	T	X	X	X
PERMANENT = *number* [BYTES]/ PERM = *number* [BYTES]	T	X	X	X
TEMPORARY = *number* [bytes]	T	X	X	X
SPOOL = *number* [BYTES]	T	X	X	X
ACCOUNT = '*account_ID*'	T	X	X	X
[NO] FALLBACK [PROTECTION]	T	X	X	X
[BEFORE] JOURNAL/ NO [BEFORE] JOURNAL/ DUAL [BEFORE] JOURNAL	T	X	X	X
[NO] AFTER JOURNAL/ DUAL AFTER JOURNAL/ [NOT] LOCAL AFTER JOURNAL	T	X	X	X
DEFAULT JOURNAL TABLE = *table_name*	T	X	X	X
DROP DEFAULT JOURNAL TABLE [= *table_name*]	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
MODIFY PROFILE	T	X	X	X
ACCOUNT = '*account_id*'/ ACCOUNT = ('*account_id*' [… ,'*account_id*'])/ ACCOUNT = NULL	T	X	X	X
COST PROFILE = *cost_profile_name*/ COST PROFILE = NULL	T	X	X	
DEFAULT DATABASE = *database_name*/ DEFAULT DATABASE = NULL	T	X	X	X
SPOOL = *n* [BYTES]/ SPOOL = NULL	T	X	X	X
TEMPORARY = *n* [BYTES]/ TEMPORARY = NULL	T	X	X	X
PASSWORD [ATTRIBUTES] = NULL/	T	X	X	X
PASSWORD [ATTRIBUTES] = ([EXPIRE = *n*/ EXPIRE = NULL] [[,] MINCHAR = *n*/ MINCHAR = NULL] [[,] MAXCHAR = *n*/ MAXCHAR = NULL] [[,] DIGITS = *n*/ DIGITS = NULL] [[,] SPECCHAR = *c*/ SPECCHAR = NULL] [[,] MAXLOGONATTEMPTS = *n*/ MAXLOGONATTEMPTS = NULL] [[,] LOCKEDUSEREXPIRE = *n*/ LOCKEDUSEREXPIRE = NULL] [[,] REUSE = *n*/ REUSE = NULL]				
[[,] RESTRICTWORDS = *c*/ RESTRICTWORDS = NULL])	T	X	X	

Statement	Compliance	13.0	12.0	V2R6.2
MODIFY USER	T	X	X	X
PERMANENT = *number* [BYTES]/ PERM = *number* [BYTES]	T	X	X	X
PASSWORD = *password* [FOR USER]	T	X	X	X
STARTUP = '*string*;'/ STARTUP = NULL	T	X	X	X
RELEASE PASSWORD LOCK	T	X	X	X
TEMPORARY = *n* [bytes]	T	X	X	X
SPOOL = *n* [BYTES]	T	X	X	X
ACCOUNT = '*acct_ID*' ACCOUNT = ('*acct_ID*' [… ,'*acct_ID*'])	T	X	X	X
DEFAULT DATABASE = *database_name*	T	X	X	X
COLLATION = *collation_sequence*	T	X	X	X
[NO] FALLBACK [PROTECTION]	T	X	X	X
[BEFORE] JOURNAL/ NO [BEFORE] JOURNAL/ DUAL [BEFORE] JOURNAL	T	X	X	X
[NO] AFTER JOURNAL/ DUAL AFTER JOURNAL/ [NOT] LOCAL AFTER JOURNAL	T	X	X	X
DEFAULT JOURNAL TABLE = *table_name*	T	X	X	X
DROP DEFAULT JOURNAL TABLE [= *table_name*]	T	X	X	X
TIME ZONE = LOCAL/ TIME ZONE = [*sign*] *quotestring*/ TIME ZONE = NULL	T	X	X	X
DATEFORM = INTEGERDATE/ DATEFORM = ANSIDATE	T	X	X	X
DEFAULT CHARACTER SET *data_type*	T	X	X	X
DEFAULT ROLE	T	X	X	X
PROFILE	T	X	X	X
OPEN	A	X	X	X
USING [:] *host_variable_name*	A	X	X	X
[INDICATOR] :*host_indicator_name*	A	X	X	X
USING DESCRIPTOR [:] *descriptor_area*	A	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
POSITION	A	X	X	X
TO NEXT/ TO [STATEMENT] *statement_number*/ TO [STATEMENT] [:] *numvar*	A	X	X	X
PREPARE	A	X	X	X
INTO [:] *descriptor_area*	A	X	X	X
USING NAMES/ USING ANY/ USING BOTH/ USING LABELS	A	X	X	X
FOR STATEMENT *statement_number*/ FOR STATEMENT [:] *numvar*	A	X	X	X
FROM *statement_string*/ FROM [:] *statement_string_var*	A	X	X	X
RENAME FUNCTION	T	X	X	X
RENAME MACRO	T	X	X	X
RENAME PROCEDURE	T	X	X	X
RENAME TABLE	T	X	X	X
RENAME TRIGGER	T	X	X	X
RENAME VIEW	T	X	X	X
REPLACE CAST	T	X	X	X
WITH [SPECIFIC] METHOD *method_name*/ WITH INSTANCE METHOD *method_name*/ WITH [SPECIFIC] FUNCTION *function_name*	T	X	X	X
AS ASSIGNMENT	T	X	X	X
REPLACE FUNCTION	T	X	X	X
(*parameter_name data_type* […, *parameter_name data_type*])/ (*parameter_name* VARIANT_TYPE […, *parameter_name data_type*])/ (*parameter_name data_type* […, *parameter_name* VARIANT_TYPE])/ (*parameter_name* VARIANT_TYPE […, *parameter_name* VARIANT_TYPE])	A T	X X	X	X
RETURNS *data_type*/ RETURNS *data_type* CAST FROM *data_type*	A	X	X	X
LANGUAGE C/ LANGUAGE CPP/ LANGUAGE JAVA	A A	X X	X	X
NO SQL	A	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
REPLACE FUNCTION, *continued*				
SPECIFIC [*database_name.*] *function_name*	A	X	X	X
CLASS AGGREGATE/ CLASS AG	T	X	X	X
PARAMETER STYLE SQL/ PARAMETER STYLE TD_GENERAL/	A	X	X	X
PARAMETER STYLE JAVA	A	X		
[NOT] DETERMINISTIC	A	X	X	X
CALLED ON NULL INPUT/ RETURNS NULL ON NULL INPUT	A	X	X	X
USING GLOP SET *GLOP_set_name*	T	X		
EXTERNAL/ EXTERNAL NAME *function_name* [PARAMETER STYLE SQL]/ EXTERNAL NAME *function_name* [PARAMETER STYLE TD_GENERAL]/ EXTERNAL PARAMETER STYLE SQL/ EXTERNAL PARAMETER STYLE TD_GENERAL/	A	X	X	X
EXTERNAL NAME '[F *delimiter function_name*] [D] [SI *delimiter name delimiter include_name*] [CI *delimiter name delimiter include_name*] [SL *delimiter library_name*] [SO *delimiter name delimiter object_name*] [CO *delimiter name delimiter object_name*] [SP *delimiter package_name*] [SS *delimiter name delimiter source_name*] [CS *delimiter name delimiter source_name*]'/	A	X	X	X
EXTERNAL NAME '*jar_id*:*class_name.method_name*' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*class_name.method_name*(*Java_data_type* … [, *Java_data_type*]) returns *Java_data_type*' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*package_name.*[…*package_name.*]*class_name.method_name*' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*package_name.*[…*package_name.*]*class_name.method_name* (*Java_data_type* … [, *Java_data_type*]) returns *Java_data_type*' [PARAMETER STYLE JAVA]	A	X		
EXTERNAL SECURITY DEFINER/ EXTERNAL SECURITY DEFINER *authorization_name*/ EXTERNAL SECURITY INVOKER	A	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
REPLACE FUNCTION (table function form)	T	X	X	X
(*parameter_name data_type* […, *parameter_name data_type*])/	A	X	X	X
(*parameter_name* VARIANT_TYPE […, *parameter_name data_type*])/ (*parameter_name data_type* […, *parameter_name* VARIANT_TYPE])/ (*parameter_name* VARIANT_TYPE […, *parameter_name* VARIANT_TYPE])	T	X		
RETURNS TABLE (*column_name data_type* [… , *column_name data_type*])/	T	X	X	X
RETURNS TABLE VARYING COLUMNS (*max_output_columns*)		X	X	
LANGUAGE C/ LANGUAGE CPP/	T	X	X	X
LANGUAGE JAVA	T	X		
NO SQL	T	X	X	X
SPECIFIC [*database_name.*] *function_name*	T	X	X	X
PARAMETER STYLE SQL/	T	X	X	X
PARAMETER STYLE JAVA	T	X		
[NOT] DETERMINISTIC	T	X	X	X
CALLED ON NULL INPUT/ RETURNS NULL ON NULL INPUT	T	X	X	X
USING GLOP SET *GLOP_set_name*	T	X		
EXTERNAL/ EXTERNAL NAME *function_name* [PARAMETER STYLE SQL]/ EXTERNAL PARAMETER STYLE SQL/	T	X	X	X
EXTERNAL NAME '[F *delimiter function_name*] [D] [SI *delimiter name delimiter include_name*] [CI *delimiter name delimiter include_name*] [SL *delimiter library_name*] [SO *delimiter name delimiter object_name*] [CO *delimiter name delimiter object_name*] [SP *delimiter package_name*] [SS *delimiter name delimiter source_name*] [CS *delimiter name delimiter source_name*]'/	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
REPLACE FUNCTION (table function form), *continued*				
EXTERNAL NAME '*jar_id*:*class_name.method_name*' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*class_name.method_name*(*Java_data_type* … [, *Java_data_type*])' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*package_name.*[…*package_name.*]*class_name.method_name*' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*package_name.*[…*package_name.*]*class_name.method_name* (*Java_data_type* … [, *Java_data_type*]) [PARAMETER STYLE JAVA]'	T	X		
EXTERNAL SECURITY DEFINER/ EXTERNAL SECURITY DEFINER *authorization_name*/ EXTERNAL SECURITY INVOKER	T	X	X	X
REPLACE MACRO	T	X	X	X
REPLACE METHOD REPLACE CONSTRUCTOR METHOD REPLACE INSTANCE METHOD REPLACE SPECIFIC METHOD	T	X	X	X
(*parameter_name data_type* […, *parameter_name data_type*])	T	X	X	X
USING GLOP SET *GLOP_set_name*	T	X		
EXTERNAL/ EXTERNAL NAME *method_name*/ EXTERNAL NAME '[F *delimiter function_entry_name*] [D] [SI *delimiter name delimiter include_name*] [CI *delimiter name delimiter include_name*] [SL *delimiter library_name*] [SO *delimiter name delimiter object_name*] [CO *delimiter name delimiter object_name*] [SP *delimiter package_name*] [SS *delimiter name delimiter source_name*] [CS *delimiter name delimiter source_name*]'	T	X	X	X
EXTERNAL SECURITY DEFINER/ EXTERNAL SECURITY DEFINER *authorization_name*/ EXTERNAL SECURITY INVOKER	T	X	X	X
REPLACE ORDERING	A	X	X	X
MAP WITH SPECIFIC METHOD *specific_method_name*/ MAP WITH METHOD *method_name*/ MAP WITH INSTANCE METHOD *method_name*/ MAP WITH SPECIFIC FUNCTION *specific_function_name*/ MAP WITH FUNCTION *function_name*	A	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
REPLACE PROCEDURE (external stored procedure form)	A	X	X	X
parameter_name data_type/ IN *parameter_name data_type*/ OUT *parameter_name data_type*/ INOUT *parameter_name data_type*	A	X	X	X
LANGUAGE C/ LANGUAGE CPP/	A	X	X	X
LANGUAGE JAVA	A	X	X	
NO SQL/	A	X	X	X
CONTAINS SQL/ READS SQL DATA/ MODIFIES SQL DATA	A	X	X	
DYNAMIC RESULT SETS *number_of_sets*	A	X	X	
PARAMETER STYLE SQL/ PARAMETER STYLE TD_GENERAL/	A	X	X	X
PARAMETER STYLE JAVA	A	X	X	
USING GLOP SET *GLOP_set_name*	T	X		
EXTERNAL/ EXTERNAL NAME *procedure_name* [PARAMETER STYLE SQL]/ EXTERNAL NAME *procedure_name* [PARAMETER STYLE TD_GENERAL]/ EXTERNAL PARAMETER STYLE SQL/ EXTERNAL PARAMETER STYLE TD_GENERAL/	A	X	X	X
EXTERNAL NAME '[F *delimiter function_entry_name*] [D] [SI *delimiter name delimiter include_name*] [CI *delimiter name delimiter include_name*] [SL *delimiter library_name*] [SO *delimiter name delimiter object_name*] [CO *delimiter name delimiter object_name*] [SS *delimiter name delimiter source_name*] [CS *delimiter name delimiter source_name*] [SP *delimiter package_name*] [SP *delimiter* CLI]'/	A	X	X	

Statement	Compliance	13.0	12.0	V2R6.2
REPLACE PROCEDURE (external stored procedure form), *continued*				
EXTERNAL NAME '*jar_id*:*class_name.method_name*' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*class_name.method_name*(*Java_data_type* … [, *Java_data_type*])' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*package_name.*[…*package_name.*]*class_name.method_name*' [PARAMETER STYLE JAVA]/ EXTERNAL NAME '*jar_id*:*package_name.*[…*package_name.*]*class_name.method_name* (*Java_data_type* … [, *Java_data_type*])' [PARAMETER STYLE JAVA]	A	X	X	
SQL SECURITY OWNER/ SQL SECURITY CREATOR/ SQL SECURITY DEFINER/ SQL SECURITY INVOKER	T	X		
EXTERNAL SECURITY DEFINER/ EXTERNAL SECURITY DEFINER *authorization_name*/ EXTERNAL SECURITY INVOKER	A	X	X	X
REPLACE PROCEDURE (stored procedure form)	T	X	X	X
parameter_name data_type/ IN *parameter_name data_type*/ OUT *parameter_name data_type*/ INOUT *parameter_name data_type*	T	X	X	X
DYNAMIC RESULT SETS *number_of_sets*	T	X	X	
CONTAINS SQL/ READS SQL DATA/ MODIFIES SQL DATA	T	X	X	
SQL SECURITY OWNER/ SQL SECURITY CREATOR/ SQL SECURITY DEFINER/ SQL SECURITY INVOKER	T	X		
DECLARE *variable-name data-type* [DEFAULT *literal*]/ DECLARE *variable-name data-type* [DEFAULT NULL]	T	X	X	X
DECLARE *condition_name* CONDITION [FOR *sqlstate_value*]	A	X		

Statement	Compliance	13.0	12.0	V2R6.2
REPLACE PROCEDURE (stored procedure form), *continued*				
DECLARE *cursor_name* [SCROLL] CURSOR/ DECLARE *cursor_name* [NO SCROLL] CURSOR	A	X	X	X
WITHOUT RETURN/ WITH RETURN ONLY [TO CALLER]/ WITH RETURN ONLY TO CLIENT/	T	X	X	
WITH RETURN [TO CALLER]/ WITH RETURN TO CLIENT	T	X		
FOR *cursor_specification* FOR READ ONLY/ FOR *cursor_specification* FOR UPDATE/	A	X	X	X
FOR *statement_name*	T	X	X	
DECLARE CONTINUE HANDLER FOR DECLARE EXIT HANDLER FOR	A	X	X	X
SQLSTATE [VALUE] *sqlstate*/ SQLEXCEPTION/ SQLWARNING/ NOT FOUND/	A	X	X	X
condition_name	A	X		
SET *assignment_target* = *assignment_source*	T	X	X	X
IF *expression* THEN *statement* [ELSEIF *expression* THEN *statement*] [ELSE *statement*] END IF	T	X	X	X
CASE *operand1* WHEN *operand2* THEN *statement* [ELSE *statement*] END CASE	T	X	X	X
CASE WHEN *expression* THEN *statement* [ELSE *statement*] END CASE	T	X	X	X
ITERATE *label_name*	T	X	X	X
LEAVE *label_name*	T	X	X	X
SQL_statement	T	X	X	X
CALL *procedure_name*	T	X	X	X
PREPARE *statement_name* FROM *statement_variable*	T	X	X	
OPEN *cursor_name*/	T	X	X	X
OPEN *cursor_name* USING *using_argument* [… , *using_argument*]	T	X	X	
CLOSE *cursor_name*	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
REPLACE PROCEDURE (stored procedure form), *continued*				
FETCH [[NEXT] FROM] *cursor_name* INTO *local_variable_name* [… , *local_variable_name*]/ FETCH [[FIRST] FROM] *cursor_name* INTO *local_variable_name* [… , *local_variable_name*]/ FETCH [[NEXT] FROM] *cursor_name* INTO *parameter_reference* [… , *parameter_reference*]/ FETCH [[FIRST] FROM] *cursor_name* INTO *parameter_reference* [… , *parameter_reference*]	T	X	X	X
WHILE *expression* DO *statement* END WHILE	T	X	X	X
LOOP *statement* END LOOP	T	X	X	X
FOR *for_loop_variable* AS [*cursor_name* CURSOR FOR] SELECT *column_name* [AS *correlation_name*] FROM *table_name* [WHERE *clause*] [SELECT *clause*] DO *statement_list* END FOR/ FOR *for_loop_variable* AS [*cursor_name* CURSOR FOR] SELECT *expression* [AS *correlation_name*] FROM *table_name* [WHERE *clause*] [SELECT *clause*] DO *statement_list* END FOR	T	X	X	X
REPEAT *statement_list* UNTIL *conditional_expression* END REPEAT	T	X	X	X
GET DIAGNOSTICS *SQL-diagnostics-information*	A	X		
RESIGNAL [*signal_value*] [*set_signal_information*]	A	X		
SIGNAL *signal_value* [*set_signal_information*]	A	X		
REPLACE REPLICATION RULESET	T	X		
REPLACE TRANSFORM	T	X	X	X
TO SQL WITH [SPECIFIC] METHOD *method_name*/ TO SQL WITH INSTANCE METHOD *method_name*/ TO SQL WITH [SPECIFIC] FUNCTION *function_name*	T	X	X	X
FROM SQL WITH [SPECIFIC] METHOD *method_name*/ FROM SQL WITH INSTANCE METHOD *method_name*/ FROM SQL WITH [SPECIFIC] FUNCTION *function_name*	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
REPLACE TRIGGER	T	X	X	X
ENABLED/ DISABLED	T	X	X	X
BEFORE/ AFTER	T	X	X	X
INSERT/ DELETE/ UPDATE [OF (*column_list*)]	T	X	X	X
ORDER *integer*	T	X	X	X
REFERENCING OLD_TABLE [AS] *identifier* [NEW_TABLE [AS] *identifier*]/ REFERENCING OLD [AS] *identifier* [NEW [AS] *identifier*]/ REFERENCING OLD TABLE [AS] *identifier* [NEW TABLE [AS] *identifier*]/ REFERENCING OLD [ROW] [AS] *identifier* [NEW [ROW] [AS] *identifier*]/	T	X	X	X
REFERENCING OLD_NEW_TABLE [AS] *transition_table_name* (*old_transition_variable_name*, *new_transition_variable_name*)	T	X		
FOR EACH ROW/ FOR EACH STATEMENT	T	X	X	X
WHEN (*search_condition*)	T	X	X	X
(*SQL_proc_statement* ;)/ *SQL_proc_statement* / BEGIN ATOMIC (*SQL_proc_statement*;) END/ BEGIN ATOMIC *SQL_proc_statement* ; END	T	X	X	X
REPLACE VIEW	A, T	X	X	X
(*column_name* [… , *column_name*])	T	X	X	X
AS [|LOCKING statement modifier|] *query_expression*	A, T	X	X	X
WITH CHECK OPTION	A	X	X	X
RESTART INDEX ANALYSIS	T	X	X	X
CHECKPOINT *checkpoint_trigger*/ TIME LIMIT = *elapsed_time*/ CHECKPOINT *checkpoint_trigger* TIME LIMIT = *elapsed_time*	T	X		

Statement	Compliance	13.0	12.0	V2R6.2
REVOKE	A, T	X	X	X
GRANT OPTION FOR	A	X	X	X
ALL/ ALL PRIVILEGES/ ALL BUT *operation*	A	X	X	X
DELETE/ EXECUTE/ INSERT/ SELECT/ REFERENCES/ REFERENCES (*column_list*)/ UPDATE/ UPDATE (*column_list*)/	A	X	X	X
INSERT (*column_list*)/ SELECT (*column_list*)/	A	X		
ALTER/ CHECKPOINT/ CREATE/ DROP/ DUMP/ INDEX/ RESTORE/ REPLCONTROL/ UDTMETHOD/ UDTTYPE/ UDTUSAGE/	T	X	X	X
CREATE OWNER PROCEDURE/ CTCONTROL/ GLOP/ SHOW/ STATISTICS	T	X		
ON *database_name*/ ON [*database_name.*]*object_name*/ ON PROCEDURE *procedure_name*/ ON [SPECIFIC] FUNCTION *function_name*/ ON TYPE [SYSUDTLIB.]*UDT_name*	A	X	X	X
TO [ALL] *user_name*/ TO PUBLIC/ FROM [ALL] *user_name*/ FROM PUBLIC	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
REVOKE CONNECT THROUGH	T	X		
FROM *app_user_name* [... , *app_user_name*]/ FROM PERMANENT *perm_user_name* [... , *perm_user_name*]/ TO *app_user_name* [... , *app_user_name*]/ TO PERMANENT *perm_user_name* [... , *perm_user_name*]	T	X		
WITH ROLE *role_name* [... , *role_name*]	T	X		
REVOKE LOGON	T	X	X	X
ON *host_id*/ ON ALL	T	X	X	X
AS DEFAULT/ TO *database_name*/ FROM *database_name*	T	X	X	X
REVOKE MONITOR/ REVOKE *monitor_privilege*	T	X	X	X
GRANT OPTION FOR	T	X	X	X
PRIVILEGES/ BUT NOT *monitor_privilege*	T	X	X	X
TO [ALL] *user_name*/ TO PUBLIC/ FROM [ALL] *user_name*/ FROM PUBLIC	T	X	X	X
REVOKE ROLE	A	X	X	X
ADMIN OPTION FOR	A	X	X	X
REWIND	T	X	X	X
ROLLBACK	A, T	X	X	X
WORK	A	X	X	X
WORK RELEASE	T	X	X	X
'*abort_message*'	T	X	X	X
FROM_clause	T	X	X	X
WHERE_clause	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
SELECT/ SEL	A, T	X	X	X
|WITH [RECURSIVE] statement modifier|	A	X	X	X
DISTINCT/ ALL	A	X	X	X
TOP *integer* [WITH TIES]/ TOP *integer* PERCENT [WITH TIES]/ TOP *decimal* [WITH TIES]/ TOP *decimal* PERCENT [WITH TIES]	T	X	X	X
*/ *expression*/ *expression* [AS] *alias_name*/ *table_name*.*/	A	X	X	X
*.ALL/ *table_name*.*.ALL/ *column_name*.ALL	T	X	X	X
SAMPLEID	T	X	X	X
FROM *table_name*/ FROM *table_name* [AS] *alias_name*/ FROM *join_table_name* JOIN *joined_table* ON *search_condition*/ FROM *join_table_name* INNER JOIN *joined_table* ON *search_condition*/ FROM *join_table_name* LEFT JOIN *joined_table* ON *search_condition*/ FROM *join_table_name* LEFT OUTER JOIN *joined_table* ON *search_condition*/ FROM *join_table_name* RIGHT JOIN *joined_table* ON *search_condition*/ FROM *join_table_name* RIGHT OUTER JOIN *joined_table* ON *search_condition*/ FROM *join_table_name* FULL JOIN *joined_table* ON *search_condition*/ FROM *join_table_name* FULL OUTER JOIN *joined_table* ON *search_condition*/ FROM *join_table_name* CROSS JOIN/ FROM (*subquery*) [AS] *derived_table_name*/ FROM (*subquery*) [AS] *derived_table_name* (*column_name*)/	A	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
SELECT, *continued*				
FROM TABLE (*function_name*([*expression* [… , *expression*]])) [AS] *derived_table_name*/ FROM TABLE (*function_name*([*expression* [… , *expression*]])) [AS] *derived_table_name* (*column_name* [… , *column_name*])/	T	X	X	X
FROM TABLE (*function_name*([*expression* [… , *expression*]]) RETURNS *table_name*) [AS] *derived_table_name* [(*column_name* [… , *column_name*])/ FROM TABLE (*function_name*([*expression* [… , *expression*]]) RETURNS (*column_name* [… , *column_name*])) [AS] *derived_table_name* [(*column_name* [… , *column_name*])]	T	X	X	
FROM TABLE (*function_name*([*expression* [… , *expression*]]) [RETURNS *table_name*] [HASH BY *column_name*] [LOCAL ORDER BY *column_name*]) [AS] *derived_table_name* [(*column_name* [… , *column_name*])/ FROM TABLE (*function_name*([*expression* [… , *expression*]]) [RETURNS (*column_name* [… , *column_name*])] [HASH BY *column_name*] [LOCAL ORDER BY *column_name*]) [AS] *derived_table_name* [(*column_name* [… , *column_name*])]	T	X		
|WHERE statement modifier|	A	X	X	X
|GROUP BY statement modifier|	A, T	X	X	X
|HAVING statement modifier|	A	X	X	X
|SAMPLE statement modifier|	T	X	X	X
|QUALIFY statement modifier|	T	X	X	X
|ORDER BY statement modifier|	A, T	X	X	X
|WITH statement modifier|	T	X	X	X
SELECT AND CONSUME TOP 1	T	X	X	X
FROM *queue_table_name*	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
SELECT … INTO/ SEL … INTO	A, T	X	X	X
DISTINCT/ ALL	A	X	X	X
AND CONSUME TOP 1	T	X	X	X
expression/ *expression* [AS] *alias_name*	A	X	X	X
FROM *table_name*/ FROM *table_name* [AS] *alias_name*/ FROM *join_table_name* [INNER] JOIN *joined_table* ON *search_condition*/ FROM *join_table_name* LEFT [OUTER] JOIN *joined_table* ON *search_condition*/ FROM *join_table_name* RIGHT [OUTER] JOIN *joined_table* ON *search_condition*/ FROM *join_table_name* FULL [OUTER] JOIN *joined_table* ON *search_condition*/ FROM *join_table_name* CROSS JOIN/ FROM (*subquery*) [AS] *derived_table_name*/ FROM (*subquery*) [AS] *derived_table_name* (*column_name*)	A	X	X	X
|WHERE statement modifier|	A	X	X	X
SET BUFFERSIZE (embedded SQL)	T	X	X	X
SET CHARSET (embedded SQL)	T	X	X	X
SET CONNECTION (embedded SQL)	T	X	X	X
SET CRASH (embedded SQL)	T	X	X	X
WAIT_NOTELL/ NOWAIT_TELL	T	X	X	X
SET QUERY_BAND	T	X	X	
'*pair_name=pair_value*; [… *pair_name=pair_value*;]'/ NONE	T	X	X	
UPDATE	T	X		
FOR SESSION/ FOR TRANSACTION	T	X	X	
SET ROLE	A, T	X	X	X
role_name/ NONE/	A	X	X	X
NULL/ ALL/ EXTERNAL	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
SET SESSION ACCOUNT/ SS ACCOUNT	T	X	X	X
FOR SESSION/ FOR REQUEST	T	X	X	X
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL/ SS CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL	A	X	X	X
RU/ READ UNCOMMITTED/ SR/ SERIALIZABLE	A	X	X	X
SET SESSION COLLATION/ SS COLLATION	T	X	X	X
SET SESSION DATABASE/ SS DATABASE	T	X	X	X
SET SESSION DATEFORM/ SS DATEFORM	T	X	X	X
ANSIDATE/ INTEGERDATE	T	X	X	X
SET SESSION FUNCTION TRACE/ SS FUNCTION TRACE	T	X	X	X
OFF/ USING *mask* FOR TABLE *table_name*/ USING *mask* FOR TRACE TABLE *table_name*	T	X	X	X
SET SESSION OVERRIDE REPLICATION/ SS OVERRIDE REPLICATION	T	X	X	X
OFF/ ON	T	X	X	X
SET SESSION SUBSCRIBER/	T	X		
OFF/ ON	T	X		
SET TIME ZONE	T	X	X	X
LOCAL/ INTERVAL *offset* HOUR TO MINUTE/ USER	T	X	X	X
SHOW	T	X	X	X
QUALIFIED	T	X	X	X
SHOW CAST	T	X	X	X
SHOW ERROR TABLE	T	X	X	

Statement	Compliance	13.0	12.0	V2R6.2
SHOW FUNCTION SHOW SPECIFIC FUNCTION	T	X	X	X
SHOW HASH INDEX	T	X	X	X
SHOW JOIN INDEX	T	X	X	X
SHOW MACRO	T	X	X	X
SHOW METHOD SHOW CONSTRUCTOR METHOD SHOW INSTANCE METHOD SHOW SPECIFIC METHOD	T	X	X	X
SHOW PROCEDURE	T	X	X	X
SHOW QUERY LOGGING	T	X		
SHOW REPLICATION GROUP	T	X	X	X
SHOW [TEMPORARY] TABLE	T	X	X	X
SHOW TRIGGER	T	X	X	X
SHOW TYPE	T	X	X	X
SHOW VIEW	T	X	X	X
TEST	T	X	X	X
async_statement_identifier/ :*namevar*	T	X	X	X
COMPLETION	T	X	X	X
UPDATE/ UPD (searched form)	A, T	X	X	X
table_name	A	X	X	X
[AS] *alias_name*/ FROM *table_name* [[AS] *alias_name*] [… , *table_name* [[AS] *alias_name*]]	A, T	X	X	X
SET *column_name=expression* [… , *column_name=expression*]/ SET *column_name=expression* [… , *column_name=expression*] [… , *column_name.mutator_name=expression*]/ SET *column_name.mutator_name=expression* [… , *column_name.mutator_name=expression*] [… , *column_name=expression*]	A	X	X	X
ALL	T	X	X	X
|WHERE statement modifier|	A	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
UPDATE/ UPD (positioned form)	A	X	X	X
table_name [*alias_name*]	A	X	X	X
SET *column_name=expression* [… , *column_name=expression*]	A	X	X	X
WHERE CURRENT OF *cursor_name*	A	X	X	X
UPDATE/ UPD (upsert form)	T	X	X	X
table_name_1	T	X	X	X
SET *column_name=expression* [… , *column_name=expression*]/	T	X	X	X
SET *column_name=expression* [… , *column_name=expression*] [… , *column_name.mutator_name=expression*]/	T	X	X	X
SET *column_name.mutator_name=expression* [… , *column_name.mutator_name=expression*] [… , *column_name=expression*]				
|WHERE statement modifier|	T	X	X	X
ELSE INSERT [INTO] *table_name_2*/ ELSE INS [INTO] *table_name_2*	T	X	X	X
[(*column_name* [… , *column_name*])] VALUES (*expression*)/ DEFAULT VALUES	T	X	X	X
WAIT	T	X	X	X
async_statement_identifier COMPLETION/ ALL COMPLETION/ ANY COMPLETION INTO [:] *stmtvar*, [:] *sessvar*	T	X	X	X
WHENEVER	A, T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
Request Modifiers				
EXPLAIN	T	X	X	X
IN XML/ IN XML COMPRESS/ IN XML NODDLTEXT/ IN XML COMPRESS NODDLTEXT	T	X		
USING	T	X	X	X
AS LOCATOR/ AS DEFERRED/	T	X	X	X
AS DEFERRED BY NAME	T	X		
Statement Modifiers				
ASYNC	T	X	X	X
EXEC SQL	A	X	X	X
GROUP BY clause	A, T	X	X	X
CUBE/ GROUPING SETS/ ROLLUP	A	X	X	X
HAVING clause	A	X	X	X
LOCKING/ LOCK	T	X	X	X
DATABASE *database_name*/ TABLE *table_name*/ VIEW *view_name*/ ROW	T	X	X	X
FOR/ IN	T	X	X	X
ACCESS/ EXCLUSIVE/ EXCL/ SHARE/ WRITE/ CHECKSUM/ READ/ READ OVERRIDE	T	X	X	X

Statement	Compliance	13.0	12.0	V2R6.2
Statement Modifiers, *continued*				
LOCKING, *continued*				
MODE	T	X	X	X
NOWAIT	T	X	X	X
ORDER BY clause	A, T	X	X	X
expression	T	X	X	X
column_name/ *column_position*	A	X	X	X
ASC/ DESC	A	X	X	X
QUALIFY clause	T	X	X	X
SAMPLE clause	T	X	X	X
WITH REPLACEMENT	T	X	X	X
RANDOMIZED ALLOCATION	T	X	X	X
WHERE clause	A	X	X	X
WITH clause	T	X	X	X
expression_1	T	X	X	X
BY *expression_2*	T	X	X	X
ASC/ DESC	T	X	X	X
WITH [RECURSIVE] clause	A	X	X	X
(*column_name* [… , *column_name*])	A	X	X	X
AS *(seed_statement* [UNION ALL *recursive_statement*)] [… [UNION ALL *seed_statement*] [… UNION ALL *recursive_statement*])	A	X	X	X

Data Types and Literals

The following list contains all SQL data types and literals for this release and previous releases of Teradata Database.

The following type codes appear in the Compliance column:

A	ANSI SQL:2008 compliant
T	Teradata extension
A, T	ANSI SQL:2008 compliant with Teradata extensions
P	Partially ANSI SQL:2008 compliant
M	ISO SQL/MM Spatial compliant
M(P)	Partially ISO SQL/MM Spatial compliant

Data Type / Literal	Compliance	13.0	12.0	V2R6.2
Data Types				
BIGINT	A	X	X	X
BINARY LARGE OBJECT, BLOB	A	X	X	X
BYTE	T	X	X	X
BYTEINT	T	X	X	X
CHAR, CHARACTER	A	X	X	X
CHAR VARYING, CHARACTER VARYING	A	X	X	X
CHARACTER LARGE OBJECT, CLOB	A	X	X	X
DATE	A, T	X	X	X
DEC, DECIMAL	A	X	X	X
DOUBLE PRECISION	A	X	X	X
FLOAT	A	X	X	X
GRAPHIC	T	X	X	X
INT, INTEGER	A	X	X	X
INTERVAL DAY	A	X	X	X
INTERVAL DAY TO HOUR	A	X	X	X
INTERVAL DAY TO MINUTE	A	X	X	X
INTERVAL DAY TO SECOND	A	X	X	X
INTERVAL HOUR	A	X	X	X

Data Type / Literal	Compliance	13.0	12.0	V2R6.2
Data Types, *continued*				
INTERVAL HOUR TO MINUTE	A	X	X	X
INTERVAL HOUR TO SECOND	A	X	X	X
INTERVAL MINUTE	A	X	X	X
INTERVAL MINUTE TO SECOND	A	X	X	X
INTERVAL MONTH	A	X	X	X
INTERVAL SECOND	A	X	X	X
INTERVAL YEAR	A	X	X	X
INTERVAL YEAR TO MONTH	A	X	X	X
LONG VARCHAR	T	X	X	X
LONG VARGRAPHIC	T	X	X	X
MBR	T	X		
NUMERIC	A	X	X	X
PERIOD(DATE)	T	X		
PERIOD(TIME [(*n*)] [WITH TIME ZONE])	T	X		
PERIOD(TIMESTAMP [(*n*)] [WITH TIME ZONE])	T	X		
REAL	A	X	X	X
SMALLINT	A	X	X	X
ST_Geometry	M(P)	X		
TIME	P	X	X	X
TIME WITH TIMEZONE	A	X	X	X
TIMESTAMP	P	X	X	X
TIMESTAMP WITH TIMEZONE	A	X	X	X
user-defined type (UDT)	A	X	X	X
VARBYTE	T	X	X	X
VARCHAR	A	X	X	X
VARGRAPHIC	T	X	X	X
Literals				
Character data	A	X	X	X
DATE	A	X	X	X

Data Type / Literal	Compliance	13.0	12.0	V2R6.2
Literals, *continued*				
Decimal	A	X	X	X
Floating point	A	X	X	X
Graphic	T	X	X	X
Hexadecimal	T	X	X	X
Integer	A	X	X	X
Interval	A	X	X	X
PERIOD	T	X		
TIME	A	X	X	X
TIMESTAMP	A	X	X	X
Unicode character string	A, T	X	X	
Unicode delimited identifier	A, T	X	X	
Data Type Attributes				
AS output format phrase	A	X	X	X
CASESPECIFIC/NOT CASESPECIFIC phrase/ CS/NOT CS phrase	T	X	X	X
CHARACTER SET	A	X	X	X
CHECK table constraint attribute	A	X	X	X
COMPRESS/ COMPRESS NULL/ COMPRESS *string*/ COMPRESS *value* column storage attribute	T	X	X	X
COMPRESS (*value_list*) column storage attribute	T	X	X	X
CONSTRAINT/ CONSTRAINT CHECK/ CONSTRAINT PRIMARY KEY/ CONSTRAINT REFERENCES/ CONSTRAINT UNIQUE column constraint attribute	T	X	X	X
DEFAULT *constant_value*/ DEFAULT DATE *quotestring*/ DEFAULT INTERVAL *quotestring*/ DEFAULT TIME *quotestring*/ DEFAULT TIMESTAMP *quotestring* default value control phrase	A	X	X	X

Data Type / Literal	Compliance	13.0	12.0	V2R6.2
Data Type Attributes, *continued*				
FOREIGN KEY table constraint attribute	A	X	X	X
FORMAT output format phrase	T	X	X	X
NAMED output format phrase	T	X	X	X
NOT NULL default value control phrase	A	X	X	X
PRIMARY KEY table constraint attribute	A	X	X	X
REFERENCES table constraint attribute	A	X	X	X
TITLE output format phrase	T	X	X	X
UC, UPPERCASE phrase	T	X	X	X
UNIQUE table constraint attribute	A	X	X	X
WITH CHECK OPTION/ WITH NO CHECK OPTION column constraint attribute	T	X	X	X
WITH DEFAULT default value control phrase	T	X	X	X

Functions, Operators, Expressions, and Predicates

The following list contains all SQL functions, operators, expressions, and predicates for this release and previous releases of Teradata Database.

The following type codes appear in the Compliance column:

A ANSI SQL:2008 compliant
A(P) Partially ANSI SQL:2008 compliant
T Teradata extension
A, T ANSI SQL:2008 compliant with Teradata extensions

Function / Operator / Expression	Compliance	13.0	12.0	V2R6.2
- (subtract)	A	X	X	X
- (unary minus)	A	X	X	X
* (multiply)	A	X	X	X
** (exponentiate)	T	X	X	X

Function / Operator / Expression		Compliance	13.0	12.0	V2R6.2
/ (divide)		A	X	X	X
^= (inequality)		T	X	X	X
+ (add)		A	X	X	X
+ (unary plus)		A	X	X	X
< (less than)		A	X	X	X
<= (less than or equal)		A	X	X	X
<> (inequality)		A	X	X	X
= (equality)		A	X	X	X
> (greater than)		A	X	X	X
>= (greater than or equal)		A	X	X	X
ABS		T	X	X	X
ACCOUNT		T	X	X	X
ACOS		T	X	X	X
ACOSH		T	X	X	X
ADD_MONTHS		T	X	X	X
ALL		A	X	X	X
AND		A	X	X	X
ANY		A	X	X	X
ASIN		T	X	X	X
ASINH		T	X	X	X
ATAN		T	X	X	X
ATAN2		T	X	X	X
ATANH		T	X	X	X
AVE/ AVERAGE/ AVG		T A	X X	X X	X X
	OVER	A	X	X	X
	PARTITION BY *value_expression*	A	X	X	X
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	X
	ROWS *window_frame_extent*	A	X	X	X

Function / Operator / Expression	Compliance	13.0	12.0	V2R6.2
BEGIN	T	X		
BETWEEN NOT BETWEEN	A	X	X	X
BYTE/ BYTES	T	X	X	X
CASE	A	X	X	X
CASE_N	T	X	X	X
CAST	A, T	X	X	X
CHAR/ CHARACTERS/ CHARS	T	X	X	X
CHAR_LENGTH/ CHARACTER_LENGTH	A	X	X	X
CHAR2HEXINT	T	X	X	X
COALESCE	A	X	X	X
CONTAINS	T	X		
CORR	A	X	X	X
OVER	A	X	X	
PARTITION BY *value_expression*	A	X	X	
ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	
ROWS *window_frame_extent*	A	X	X	
COS	T	X	X	X
COSH	T	X	X	X
COUNT	A	X	X	X
OVER	A	X	X	X
PARTITION BY *value_expression*	A	X	X	X
ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	X
ROWS *window_frame_extent*	A	X	X	X

Function / Operator / Expression		Compliance	13.0	12.0	V2R6.2
COVAR_POP		A	X	X	X
	OVER	A	X	X	
	PARTITION BY *value_expression*	A	X	X	
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	
	ROWS *window_frame_extent*	A	X	X	
COVAR_SAMP		A	X	X	X
	OVER	A	X	X	
	PARTITION BY *value_expression*	A	X	X	
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	
	ROWS *window_frame_extent*	A	X	X	
CSUM		T	X	X	X
CURRENT_DATE		A	X	X	X
CURRENT_ROLE		A	X		
CURRENT_TIME		A	X	X	X
CURRENT_TIMESTAMP		A	X	X	X
CURRENT_USER		A	X		
DATABASE		T	X	X	X
DATE		T	X	X	X
DEFAULT		A, T	X	X	X
DEGREES		T	X	X	
EQ		T	X	X	X
END		T	X		
EXCEPT		A, T	X	X	X
	ALL	T	X	X	X
EXISTS NOT EXISTS		A	X	X	X
EXP		T	X	X	X
EXTRACT		A(P)	X	X	X

Function / Operator / Expression	Compliance	13.0	12.0	V2R6.2
FORMAT	T	X	X	X
GE	T	X	X	X
GROUPING	A	X	X	X
GT	T	X	X	X
HASHAMP	T	X	X	X
HASHBAKAMP	T	X	X	X
HASHBUCKET	T	X	X	X
HASHROW	T	X	X	X
IN NOT IN	A	X	X	X
INDEX	T	X	X	X
INTERSECT	A, T	X	X	X
ALL	T	X	X	X
INTERVAL	T	X		
IS NULL IS NOT NULL	A	X	X	X
IS UNTIL_CHANGED IS NOT UNTIL_CHANGED	T	X		
KURTOSIS	A	X	X	X
LAST	T	X		
LDIFF	T	X		
LE	T	X	X	X
LIKE NOT LIKE	A	X	X	X
LN	T	X	X	X
LOG	T	X	X	X
LOWER	A	X	X	X
LT	T	X	X	X
MAVG	T	X	X	X

Function / Operator / Expression		Compliance	13.0	12.0	V2R6.2
MAX/ MAXIMUM		A T	X	X	X
	OVER	A	X	X	X
	PARTITION BY *value_expression*	A	X	X	X
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	X
	ROWS *window_frame_extent*	A	X	X	X
MCHARACTERS		T	X	X	X
MDIFF		T	X	X	X
MEETS		T	X		
MIN/ MINIMUM		A T	X	X	X
	OVER	A	X	X	X
	PARTITION BY *value_expression*	A	X	X	X
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	X
	ROWS *window_frame_extent*	A	X	X	X
MINUS		T	X	X	X
	ALL	T	X	X	X
MLINREG		T	X	X	X
MOD		T	X	X	X
MSUM		T	X	X	X
NE		T	X	X	X
NEW		A(P)	X	X	X
NEW VARIANT_TYPE		T	X		
NEXT		T	X		
NOT		A	X	X	X
NOT=		T	X	X	X
NULLIF		A	X	X	X
NULLIFZERO		T	X	X	X

Function / Operator / Expression	Compliance	13.0	12.0	V2R6.2
OCTET_LENGTH	A	X	X	X
OR	A	X	X	X
OVERLAPS	A	X	X	X
P_INTERSECT	T	X		
P_NORMALIZE	T	X		
PERCENT_RANK	A	X	X	X
OVER	A	X	X	X
PARTITION BY *value_expression*	A	X	X	X
ORDER BY *value_expression*/	A	X	X	X
ORDER BY *value_expression* RESET WHEN *condition*	T	X		
PERIOD	T	X		
POSITION	A	X	X	X
PRECEDES	T	X		
PRIOR	T	X		
PROFILE	T	X	X	X
QUANTILE	T	X	X	X
RADIANS	T	X	X	
RANDOM	T	X	X	X
RANGE_N	T	X	X	X
RANK	T	X	X	X
RANK	A	X	X	X
OVER	A	X	X	X
PARTITION BY *value_expression*	A	X	X	X
ORDER BY *value_expression*/	A	X	X	X
ORDER BY *value_expression* RESET WHEN *condition*	T	X		
RDIFF	T	X		

Function / Operator / Expression		Compliance	13.0	12.0	V2R6.2
REGR_AVGX		A	X	X	X
	OVER	A	X	X	
	PARTITION BY *value_expression*	A	X	X	
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	
	ROWS *window_frame_extent*	A	X	X	
REGR_AVGY		A	X	X	X
	OVER	A	X	X	
	PARTITION BY *value_expression*	A	X	X	
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	
	ROWS *window_frame_extent*	A	X	X	
REGR_COUNT		A	X	X	X
	OVER	A	X	X	
	PARTITION BY *value_expression*	A	X	X	
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	
	ROWS *window_frame_extent*	A	X	X	
REGR_INTERCEPT		A	X	X	X
	OVER	A	X	X	
	PARTITION BY *value_expression*	A	X	X	
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	
	ROWS *window_frame_extent*	A	X	X	

Function / Operator / Expression	Compliance	13.0	12.0	V2R6.2
REGR_R2	A	X	X	X
OVER	A	X	X	
PARTITION BY *value_expression*	A	X	X	
ORDER BY *value_expression*/	A	X	X	
ORDER BY *value_expression* RESET WHEN *condition*	T	X		
ROWS *window_frame_extent*	A	X	X	
REGR_SLOPE	A	X	X	X
OVER	A	X	X	
PARTITION BY *value_expression*	A	X	X	
ORDER BY *value_expression*/	A	X	X	
ORDER BY *value_expression* RESET WHEN *condition*	T	X		
ROWS *window_frame_extent*	A	X	X	
REGR_SXX	A	X	X	X
OVER	A	X	X	
PARTITION BY *value_expression*	A	X	X	
ORDER BY *value_expression*/	A	X	X	
ORDER BY *value_expression* RESET WHEN *condition*	T	X		
ROWS *window_frame_extent*	A	X	X	
REGR_SXY	A	X	X	X
OVER	A	X	X	
PARTITION BY *value_expression*	A	X	X	
ORDER BY *value_expression*/	A	X	X	
ORDER BY *value_expression* RESET WHEN *condition*	T	X		
ROWS *window_frame_extent*	A	X	X	

Function / Operator / Expression		Compliance	13.0	12.0	V2R6.2
REGR_SYY		A	X	X	X
	OVER	A	X	X	
	PARTITION BY *value_expression*	A	X	X	
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	
	ROWS *window_frame_extent*	A	X	X	
ROLE		T	X	X	X
ROW_NUMBER		A	X	X	X
	OVER	A	X	X	X
	PARTITION BY *value_expression*	A	X	X	X
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	X
SESSION		T	X	X	X
SIN		T	X	X	X
SINH		T	X	X	X
SKEW		A	X	X	X
SOME		A	X	X	X
SOUNDEX		T	X	X	X
SQRT		T	X	X	X
STDDEV_POP		A	X	X	X
	OVER	A	X	X	
	PARTITION BY *value_expression*	A	X	X	
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	
	ROWS *window_frame_extent*	A	X	X	

Function / Operator / Expression		Compliance	13.0	12.0	V2R6.2
STDDEV_SAMP		A	X	X	X
	OVER	A	X	X	
	PARTITION BY *value_expression*	A	X	X	
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	
	ROWS *window_frame_extent*	A	X	X	
STRING_CS		T	X	X	X
SUBSTR		T	X	X	X
SUBSTRING		A	X	X	X
SUCCEEDS		T	X		
SUM		A	X	X	X
	OVER	A	X	X	X
	PARTITION BY *value_expression*	A	X	X	X
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	X
	ROWS *window_frame_extent*	A	X	X	X
TAN		T	X	X	X
TANH		T	X	X	X
TIME		T	X	X	X
TITLE		T	X	X	X
TRANSLATE		A	X	X	X
TRANSLATE_CHK		T	X	X	X
TRIM		A(P)	X	X	X
TYPE		T	X	X	X
UNION		A, T	X	X	X
	ALL	T	X	X	X
UPPER		A	X	X	X
USER		A	X	X	X

Function / Operator / Expression		Compliance	13.0	12.0	V2R6.2
VAR_POP		A	X	X	X
	OVER	A	X	X	
	PARTITION BY *value_expression*	A	X	X	
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	
	ROWS *window_frame_extent*	A	X	X	
VAR_SAMP		A	X	X	X
	OVER	A	X	X	
	PARTITION BY *value_expression*	A	X	X	
	ORDER BY *value_expression*/ ORDER BY *value_expression* RESET WHEN *condition*	A T	X X	X	
	ROWS *window_frame_extent*	A	X	X	
VARGRAPHIC		T	X	X	X
WIDTH_BUCKET		A	X	X	X
ZEROIFNULL		T	X	X	X

Glossary

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 On-Line Analytical Processing

OLTP On-Line Transaction Processing

QCD Query Capture Database

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

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

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z