Teradata Database

SQL Fundamentals

Release 14.0
B035-1141-111A
January 2012

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 Releases and Operating Systems

This book supports Teradata® Database 14.0.

Teradata Database 14.0 is supported on:

- SUSE Linux Enterprise Server (SLES)10
- SLES 11

Note that SLES 11 will be supported after the initial release of Teradata Database 14.0.

Teradata Database client applications support other operating systems.

Prerequisites

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

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

Changes to This Book

Release	Description
Teradata Database 14.0 January 2012	Updated system limits. Described Teradata SQL design philosophy.
Teradata Database 14.0 November 2011	Added new reserved, nonreserved, and TPT keywords. Users should not change cost profile settings in a production environment without direction from the Teradata Support Center. Clarified the distinction between SQL statements and SQL requests. Deleted former chapter 1 (“Objects”) from this book.

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.

URL	Description
developer.teradata.com/	Teradata Developer Exchange provides articles on using Teradata products, technical discussion forums, and code downloads.

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

Teradata Database Optional Features

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

- Teradata Row Level Security
- Temporal Table Support
- Teradata Columnar
- Teradata Virtual Storage (VS)

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

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

Table of Contents

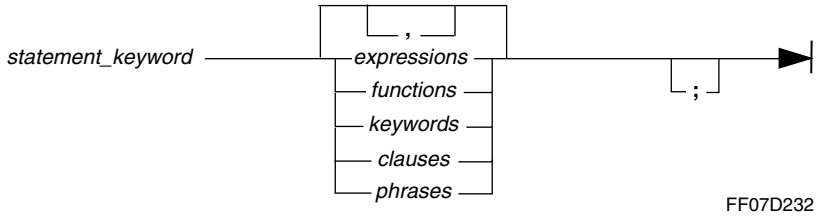
CHAPTER 1 Basic SQL Syntax and Lexicon

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. The request terminator is required for a request defined in the body of a macro.

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 13
keywords	
expressions	"Expressions" on page 14
functions	"Functions" on page 37
separators	"Separators" on page 39
terminators	"Terminators" on page 42

SQL Lexicon Characters

Client Character Data

The characters that make up the SQL lexicon can be represented on the client system in ASCII, EBCDIC, UTF-8, UTF-16, 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 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

Keywords are words that have special meanings in SQL statements. Keywords can be 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.

Other 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 identically:

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

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

Reserved Words and Object Names

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:

- Put the newly nonvalid name in double quotation marks.
- Rename the object.

In either case you must change your applications.

Expressions

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

Scalar Expressions

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

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

Query Expressions

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

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

Related Topics

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

Names on Systems with Standard Language Support

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

Rules

Following are the rules for naming Teradata Database database objects on systems enabled for standard language support:

- 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 quotation marks, 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 MARK Characters and Names" on page 16 and "Internal Hexadecimal Representation of a Name" on page 17.
- Names that are not enclosed within double quotation marks 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 with Japanese Language Support" on page 19).
- 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 18).

QUOTATION MARK Characters and Names

Enclosing names in QUOTATION MARK 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 MARK characters that delineate a name are not part of the name, but they are required if the name is not otherwise valid.

For example, these two names are identical, even though one is enclosed within QUOTATION MARK 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)

To include a QUOTATION MARK in an object name, use two consecutive QUOTATION MARK characters (""). Only one QUOTATION MARK character is counted in the length of the name and is stored in Dictionary tables that track name usage.

Using any of the following example names for an object is valid.

- Employee
- job_title
- CURRENT_SALARY
- DeptNo
- Population_of_Los_Angeles
- Totaldollars
- "Table A"
- "Today's Date"
- “Monthly””A””Sales”

Note: If you use names that are enclosed in double quotation marks, the QUOTATION MARK 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 display such names, they are displayed without the double quotation marks, and if the resulting names are used without adding double quotation marks, the likely outcome is an error report.

For example, if a statement used "D'Augusta" as the name of a table column, any HELP statements that return column names return the name as D'Augusta (without being enclosed in QUOTATION MARK 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 names enclosed within double quotation marks 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 full width Latin characters from the KANJIEBCDIC5035_0I character set on a system enabled for Japanese language support:

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

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

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

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

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:

```
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, the column names are returned exactly as defined they were defined, for example, *last_name*, then *first_nam*e.

```
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 with Japanese Language Support

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

Names that are not enclosed in double quotation marks can use the following characters when Japanese language support is enabled:

- Any character valid in a name that is not enclosed in double quotation marks under standard language support:
  - Uppercase or lowercase letters (A to Z and a to z)
  - Digits (0 through 9)
  - The special characters DOLLAR SIGN ($), NUMBER SIGN (#), and LOW LINE (_)
- The full-width (zenkaku) versions of the characters valid for names under standard language support:
  - Full-width uppercase or lowercase letters (A to Z and a to z)
  - Full-width digits (0 through 9)
  - The special characters full width DOLLAR SIGN ($), full width NUMBER SIGN (#), and full width LOW LINE (_)
- Full-width (zenkaku) and half-width (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 21.

Names cannot begin with a digit or a full-width 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 Names Enclosed in Double Quotation Marks and Internal Hexadecimal Representation of Names

As described in “QUOTATION MARK Characters and Names” on page 16 and “Internal Hexadecimal Representation of a Name” on page 17, 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 UTF-8, UTF-16, 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
```

Name Validation

Name validation occurs when the object is created or renamed:

- 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:

```
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:

```
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 21.

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**E**F**G**HI**J**K**LMNOPQR**	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).
Multibyte	Multibyte characters in object names are handled according to the character set in effect for the current session: • KanjiEBCDIC: 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. • KanjiEUC: Under code set 1, each multibyte character is translated from KanjiEUC to KanjiShift-JIS. Under code set 2, byte ss2 (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. • KanjiShift-JIS: Each multibyte character is processed without translation. It remains in the client encoding. For information on non-Japanese multibyte character sets, see *International Character Set Support*.

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 full-width Latin lowercase letter is equivalent to its corresponding full-width Latin uppercase letter. For example, the full-width letter 'a' is equivalent to the full-width 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

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 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
--------------------------------------------------------------- --------------
44626173652020202020202020202020202020202020202020202020202020 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 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:

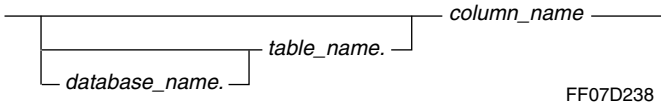
```
U&"x0044x0062x0061x0073x0065" UESCAPE 'x'
```

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

Standard Form for Data in Teradata Database

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



FF07D238

where:

This 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 27.
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 27.
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 26.

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 Systems with Standard Language Support" on page 15.

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

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 29)

- 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

- 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 29.
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

The default database is a Teradata extension to SQL that defines a database that Teradata Database uses to look for unqualified names, such as table, view, trigger, or macro names, in SQL statements.

The default database is not the only database that Teradata Database uses to find an unqualified name in an SQL statement. Teradata Database also looks for the name in:

- Other databases, if any, referenced by the SQL statement
- The login user database for a volatile table, if the unqualified object name is a table name
- The SYSLIB database, if the unqualified object name is a C or C++ UDF that is not in the default database

If the unqualified object name exists in more than one of the databases in which Teradata Database looks, the SQL statement produces an ambiguous name error.

Establishing a Permanent Default Database

You can establish a permanent default database that is invoked each time you log on.

TO ...	USE one of the following SQL Data Definition statements...
define a permanent default database	• CREATE USER, with a DEFAULT DATABASE clause. • 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:

```
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 25.
	“Unqualified Object Names” on page 27.
using profiles to define a default database	

Literals

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

Numeric Literals

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

- digits 0 through 9
- plus sign
- minus sign
- decimal point

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

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

DateTime Literals

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

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

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

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

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

Interval Literals

Interval literals provide a means for declaring spans of time.

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

```
INTERVAL '200' HOUR
```

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

Category	Type	Description
Year-Month	• YEAR • 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 apostrophes. A zero-length character literal is represented by two consecutive apostrophes ('').
Hexadecimal character literal	A hexadecimal character literal specifies a string of 0 to 62000 hexadecimal digits enclosed with apostrophes followed by the characters XCF for a CHARACTER data type or XCV for a VARCHAR data type.
Unicode character string literal	Unicode character string literals consist of 0 to 31000 Unicode characters and are useful for inserting character strings containing characters that cannot generally be entered directly from a keyboard.

Graphic Literals

A graphic literal specifies multibyte characters within the graphic repertoire.

Period Literals

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

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

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

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

A null represents either:

- An empty column
- An unknown value
- An unknowable value

Nulls are neither values nor do they signify values; they represent the absence of value. A null is a place holder indicating that no value is present.

NULL Keyword

The keyword NULL represents null, and is sometimes available as a special construct similar to, but not identical with, a literal.

ANSI Compliance

NULL is ANSI 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 78.

Operators

SQL operators express logical and arithmetic operations. Operators of the same precedence are evaluated from left to right. See "SQL Operations and Precedence" on page 36 for more detailed information.

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

The following definitions apply to SQL operators.

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

SQL Operations and Precedence

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

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

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 39.
- 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

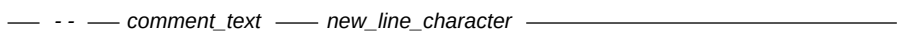
You can embed comments within an SQL request anywhere a pad character can occur.

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:

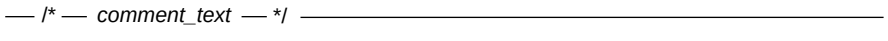
```
SELECT EmpNo, Name FROM Payroll_Test
ORDER BY Name -- Simple comment at the end of a line
;

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

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

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

A null statement has no content except for optional pad characters or SQL comments.

Example 1

The semicolon in the following request is a null statement.

```
/* This example shows a comment followed by
   a semicolon used as a null statement */
;  UPDATE Pay_Test SET ...
```

Example 2

The first SEMICOLON in the following request is a null statement. The second SEMICOLON is taken as statement separator:

```
/*    This example shows a semicolon used as a null
   statement and as a statement separator */
;  UPDATE Payroll_Test SET Name = 'Wedgewood A'
   WHERE Name = 'Wedgewood A'
;   SELECT ...
-- This example shows the use of an ANSI component
-- used as a null statement and statement separator ;
```

Example 3

A SEMICOLON that precedes the first (or only) statement of a request is taken as a null statement.

```
;DROP TABLE temp_payroll;
```

CHAPTER 2 SQL Data Definition, Control, and Manipulation

This chapter describes the functional families of the SQL language.

SQL Functional Families and Binding Styles

The SQL language can be characterized in several different ways. This chapter is organized around functional groupings of the components of the language with minor emphasis on binding styles.

Functional Family

SQL provides facilities for defining database objects, for defining user access to those objects, and for manipulating the data stored within them.

The following list describes the principal functional families of the SQL language.

- SQL Data Definition Language (DDL)
- SQL Data Control Language (DCL)
- SQL Data Manipulation Language (DML)
- Query and Workload Analysis Statements
- Help and Database Object Definition Tools

Some classifications of SQL group the data control language statements with the data definition language statements.

Binding Style

The ANSI SQL standards do not define the term binding style. The expression refers to a possible method by which an SQL statement can be invoked.

Teradata Database supports the following SQL binding styles:

- Direct, or interactive
- Embedded SQL
- Stored procedure
- SQL Call Level Interface (as ODBC)
- JDBC

The direct binding style is usually not qualified in this book set because it is the default style. Embedded SQL and stored procedure binding styles are always clearly specified, either explicitly or by context.

Related Topics

You can find more information on binding styles in the SQL book set and in other books.

For more information on ...	See ...
embedded SQL	• *Teradata Preprocessor2 for Embedded SQL Programmer Guide* • *SQL Stored Procedures and Embedded SQL*
stored procedures	*SQL Stored Procedures and Embedded SQL*
ODBC	*ODBC Driver for Teradata User Guide*
JDBC	*Teradata JDBC Driver User Guide*

Data Definition Language

The SQL Data Definition Language (DDL) is a subset of the SQL language and consists of all SQL statements that support the definition of database objects.

Purpose

Data definition language statements perform the following functions:

- Create, drop, rename, alter, modify, and replace database objects
- Comment on database objects
- Collect statistics on a column set or index
- Establish a default database
- Set a different collation sequence, account priority, DateForm, time zone, and database for the session
- Set roles
- Set the query band for a session or transaction
- Begin and end logging
- Enable and disable online archiving for all tables in a database or a specific set of tables

Rules on Entering DDL Statements

A DDL statement can be entered as:

- A single statement request.
- The solitary statement, or the last statement, in an explicit transaction (in Teradata mode, one or more requests enclosed by user-supplied BEGIN TRANSACTION and END

TRANSACTION statement, or in ANSI mode, one or more requests ending with the COMMIT keyword).

- The solitary statement in a macro.

DDL statements cannot be entered as part of a multistatement request.

Successful execution of a DDL statement automatically creates and updates entries in the Data Dictionary.

SQL DDL Statements

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

Altering Table Structure and Definition

You may need to change the structure or definition of an existing table or temporary table. In many cases, you can use ALTER TABLE and RENAME to make the changes. Some changes, however, may require you to use CREATE TABLE to recreate the table.

You cannot use ALTER TABLE on an error logging table.

Making Changes to a Table

Use the RENAME TABLE statement to change the name of a table, temporary table, queue table, or error logging table.

Use the ALTER TABLE statement to perform any of the following functions:

- Add or drop columns on an existing table or temporary table
- Add column default control, FORMAT, and TITLE attributes on an existing table or temporary table
- Add or remove journaling options on an existing table or temporary table
- Add or remove the FALLBACK option on an existing table or temporary table
- Change the DATABLOCKSIZE or percent FREESPACE on an existing table or temporary table
- Add or drop column and table level constraints on an existing table or temporary table
- Change the LOG and ON COMMIT options for a global temporary table
- Modify referential constraints
- Change the properties of the primary index for a table (some cases require an empty table)
- Change the partitioning properties of the primary index for a table, including modifications to the partitioning expression defined for use by a partitioned primary index (some cases require an empty table)
- Regenerate table headers and optionally validate and correct the partitioning of PPI table rows

- Define, modify, or delete the COMPRESS attribute for an existing column
- Change column attributes (that do not affect stored data) on an existing table or temporary table

Restrictions apply to many of the preceding modifications. For a complete list of rules and restrictions on using ALTER TABLE to change the structure or definition of an existing table, see *SQL Data Definition Language*.

To perform any of the following functions, use CREATE TABLE to recreate the table:

- Redefine the primary index or its partitioning for a non-empty table when not allowed for ALTER TABLE
- Change a data type attribute that affects existing data
- Add a column that would exceed the maximum lifetime column count

Interactively, the SHOW TABLE statement can call up the current table definition, which can then be modified and resubmitted to create a new table.

If the stored data is not affected by incompatible data type changes, an INSERT... SELECT statement can be used to transfer data from the existing table to the new table.

Dropping and Renaming Objects

Dropping Objects

To drop an object, use the appropriate DDL statement.

Object	Use
Constraint	DROP CONSTRAINT
Error logging table	DROP ERROR TABLE
	DROP TABLE
Hash index	DROP HASH INDEX
Join index	DROP JOIN INDEX
Macro	DROP MACRO
Profile	DROP PROFILE
Role	DROP ROLE
Secondary index	DROP INDEX
Stored procedure	DROP PROCEDURE
Table	DROP TABLE
Global temporary table or volatile table	
Primary index	

Object	Use
Trigger	DROP TRIGGER
User-defined function	DROP FUNCTION
User-defined method	ALTER TYPE
User-defined type	DROP TYPE
View	DROP VIEW

Renaming Objects

Teradata SQL provides RENAME statements that you can use to rename some objects. To rename objects that do not have associated RENAME statements, you must first drop them and then recreate them with a new name, or, in the case of primary indexes, use ALTER TABLE.

Object	Use
Hash index	DROP HASH INDEX and then CREATE HASH INDEX
Join index	DROP JOIN INDEX and then CREATE JOIN INDEX
Macro	RENAME MACRO
Primary index	ALTER TABLE
Profile	DROP PROFILE and then CREATE PROFILE
Role	DROP ROLE and then CREATE ROLE
Secondary index	DROP INDEX and then CREATE INDEX
Stored procedure	RENAME PROCEDURE
Table	RENAME TABLE
Global temporary table or volatile table	
Queue table	
Error logging table	
Trigger	RENAME TRIGGER
User-Defined Function	RENAME FUNCTION
User-Defined Method	ALTER TYPE and then CREATE METHOD
User-Defined Type	DROP TYPE and then CREATE TYPE
View	RENAME VIEW

Related Topics

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

Data Control Language

The SQL Data Control Language (DCL) is a subset of the SQL language and consists of all SQL statements that support the definition of security authorization for accessing database objects.

Purpose

Data control statements perform the following functions:

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

Rules on Entering DCL Statements

A data control statement can be entered as:

- A single statement request
- The solitary statement, or as the last statement, in an “explicit transaction” (one or more requests enclosed by user-supplied BEGIN TRANSACTION and END TRANSACTION statement in Teradata mode, or in ANSI mode, one or more requests ending with the COMMIT keyword).
- The solitary statement in a macro

A data control statement cannot be entered as part of a multistatement request.

Successful execution of a data control statement automatically creates and updates entries in the Data Dictionary.

Teradata SQL DCL Statements

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

Data Manipulation Language

The SQL Data Manipulation Language (DML) is a subset of the SQL language and consists of all SQL statements that support the manipulation or processing of database objects.

Selecting Columns

The SELECT statement returns information from the tables in a relational database. SELECT specifies the table columns from which to obtain the data, the corresponding database (if not defined by default), and the table (or tables) to be accessed within that database.

For example, to request the data from the name, salary, and jobtitle columns of the Employee table, type:

```
SELECT name, salary, jobtitle FROM employee ;
```

The response might be something like the following results table.

Name	Salary	JobTitle
Newman P	28600.00	Test Tech
Chin M	38000.00	Controller
Aquilar J	45000.00	Manager
Russell S	65000.00	President
Clements D	38000.00	Salesperson

Note: The left-to-right order of the columns in a result table is determined by the order in which the column names are entered in the SELECT statement. Columns in a relational table are not ordered logically.

As long as a statement is otherwise constructed properly, the spacing between statement elements is not important as long as at least one pad character separates each element that is not otherwise separated from the next.

For example, the SELECT statement in the above example could just as well be formulated like this:

```
SELECT   name,   salary,jobtitle
FROM employee;
```

Notice that there are multiple pad characters between most of the elements and that a comma only (with no pad characters) separates column name salary from column name jobtitle.

To select all the data in the employee table, you could enter the following SELECT statement:

```
SELECT * FROM employee ;
```

The asterisk specifies that the data in all columns (except system-derived columns) of the table is to be returned.

Selecting Rows

The SELECT statement retrieves stored data from a table. All rows, specified rows, or specific columns of all or specified rows can be retrieved. 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

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

Example: Three Level Subqueries

The following subqueries, nested to three levels, answer the question "Who manages the manager of Marston?"

```
SELECT Name
FROM Employee
WHERE EmpNo IN
 (SELECT MgrNo
  FROM Department
  WHERE DeptNo IN
   (SELECT DeptNo
    FROM Employee
    WHERE Name = 'Marston A') ) ;
```

The subqueries that pose the questions leading to the final answer are inverted:

- The third subquery asks the Employee table for the number of Marston's department.
- The second subquery asks the Department table for the employee number (MgrNo) of the manager associated with this department number.
- The first subquery asks the Employee table for the name of the employee associated with this employee number (MgrNo).

The result table looks like the following:

```
Name
--------
Watson L
```

This result can be obtained using only two levels of subquery, as the following example shows.

```
SELECT Name
FROM Employee
WHERE EmpNo IN
 (SELECT MgrNo
  FROM Department, Employee
  WHERE Employee.Name = 'Marston A'
  AND Department.DeptNo = Employee.DeptNo) ;
```

In this example, the second subquery defines a join of Employee and Department tables.

This result could also be obtained using a one-level query that uses correlation names, as the following example shows.

```
SELECT M.Name
FROM Employee M, Department D, Employee E
WHERE M.EmpNo = D.MgrNo AND
      E.Name = 'Marston A' AND
      D.DeptNo = E.DeptNo;
```

In some cases, as in the preceding example, the choice is a style preference. In other cases, correct execution of the query may require a subquery.

For More Information

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

Recursive Queries

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

Recursion is typically characterized by three steps:

1 Initialization

2 Recursion, or repeated iteration of the logic through the hierarchy

3 Termination

Similarly, a recursive query has three execution phases:

1 Create an initial result set.

2 Recursion based on the existing result set.

3 Final query to return the final result set.

Specifying a Recursive Query

You can specify a recursive query by:

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

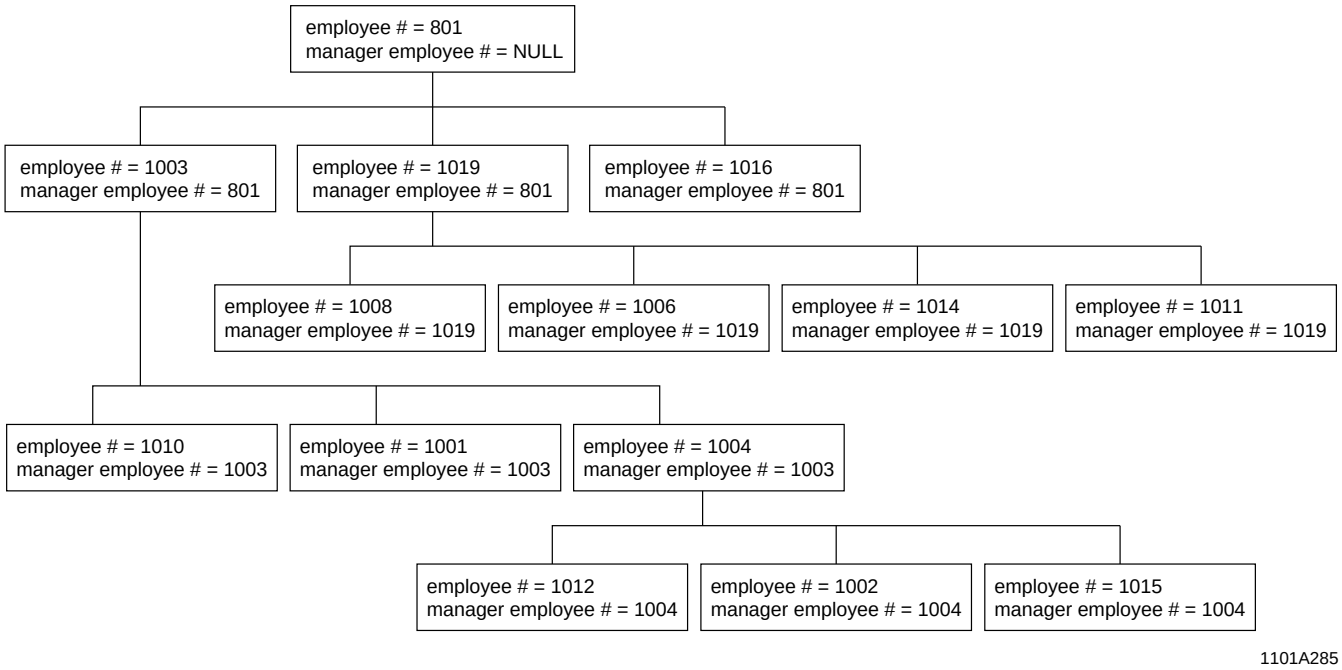
Using the WITH RECURSIVE Clause

Consider the following employee table:

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

The table represents an organizational structure containing a hierarchy of employee-manager data.

The following figure depicts what the employee table looks like hierarchically.



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 56. The following statement creates a view named hierarchy_801 using a recursive query that retrieves the employee numbers of all employees who directly or indirectly report to the manager with employee_number 801:

```
CREATE RECURSIVE VIEW hierarchy_801 (employee_number) AS
( SELECT root.employee_number
  FROM employee root
  WHERE root.manager_employee_number = 801
UNION ALL
  SELECT indirect.employee_number
  FROM hierarchy_801 direct, employee indirect
  WHERE direct.employee_number = indirect.manager_employee_number
);
```

The seed statement and recursive statement in the view definition are the same as the seed statement and recursive statement in the previous recursive query that uses the WITH RECURSIVE clause, except that the hierarchy_801 view name is different from the temp_table temporary result name.

To extract the employee information, use the following SELECT statement on the hierarchy_801 view:

```
SELECT * FROM hierarchy_801 ORDER BY employee_number;
```

Here are the results:

```
employee_number
---------------
           1001
           1002
           1003
           1004
           1006
           1008
           1010
           1011
           1012
           1014
           1015
           1016
           1019
```

Depth Control to Avoid Infinite Recursion

If the hierarchy is cyclic, or if the recursive statement specifies a bad join condition, a recursive query can produce a runaway query that never completes with a finite result. The best practice is to control the depth of the recursion:

- Specify a depth control column in the column list of the WITH RECURSIVE clause or recursive view
- Initialize the column value to 0 in the seed statements
- Increment the column value by 1 in the recursive statements
- Specify a limit for the value of the depth control column in the join condition of the recursive statements

Here is an example that modifies the previous recursive query that uses the WITH RECURSIVE clause of the employee table to limit the depth of the recursion to five cycles:

```
WITH RECURSIVE temp_table (employee_number, depth) AS
( SELECT root.employee_number, 0 AS depth
  FROM employee root
  WHERE root.manager_employee_number = 801
UNION ALL
  SELECT indirect.employee_number, direct.depth+1 AS newdepth
  FROM temp_table direct, employee indirect
  WHERE direct.employee_number = indirect.manager_employee_number
  AND newdepth <= 5
)
SELECT * FROM temp_table ORDER BY employee_number;
```

Related Topics

For details on ...	See ...
recursive queries	"WITH RECURSIVE" in *SQL Data Manipulation Language*.
recursive views	"CREATE VIEW" in *SQL Data Definition Language*.

Query and Workload Analysis Statements

Data Collection and Analysis

Teradata provides the following SQL statements for collecting and analyzing query and data demographics and statistics:

- BEGIN QUERY LOGGING
- COLLECT DEMOGRAPHICS
- COLLECT STATISTICS
- DROP STATISTICS
- DUMP EXPLAIN
- END QUERY LOGGING
- INITIATE INDEX ANALYSIS
- INITIATE PARTITION ANALYSIS
- INSERT EXPLAIN
- RESTART INDEX ANALYSIS
- SHOW QUERY LOGGING

Collected data can be used in several ways, for example:

- By the Optimizer, to produce the best query plans possible.
- To populate user-defined Query Capture Database (QCD) tables with data used by various utilities to analyze query workloads as part of the ongoing process to re-engineer the database design process.

  For example, the Teradata Index Wizard determines optimal secondary indexes, single-table join indexes, and PPIs to support the query workloads you ask it to analyze.

Index Analysis and Target Level Emulation

Teradata also provides diagnostic statements that support the Teradata Index Wizard and the cost-based and sample-based components of the target level emulation facility used to emulate a production environment on a test system:

- 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.

Note: Settings should be changed in the production environment only under the direction of Teradata Support Center personnel.

Related Topics

For details on BEGIN QUERY LOGGING, END QUERY LOGGING, and SHOW QUERY LOGGING, see *SQL Data Definition Language*. For more information on other query and workload analysis statements, see *SQL Data Manipulation Language*.

For more information on the Teradata Index Wizard, see *Teradata Index Wizard User Guide*.

Help and Database Object Definition Tools

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

HELP Statements

Various HELP statements return reports about the current column definitions for named database objects. The reports these statements return can be useful to database designers who need to fine tune index definitions, column definitions (for example, changing data typing to eliminate the necessity of ad hoc conversions), and so on.

For SQL HELP statements, see *SQL Data Definition Language Syntax and Examples*.

SHOW Statements

Most SHOW statements return a CREATE statement indicating the last data definition statement performed against the named database object. Some SHOW statements, such as SHOW QUERY LOGGING, return other information. These statements are particularly useful for application developers who need to develop exact replicas of existing objects for purposes of testing new software.

For SQL SHOW statements, see *SQL Data Definition Language Syntax and Examples*.

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 );
```

CHAPTER 3 SQL Data Handling

This chapter describes the fundamentals of Teradata Database data handling.

SQL Statement and SQL Requests

In Teradata SQL, a *statement* has three components.

- A statement keyword, such as SELECT, CREATE TABLE, or ALTER PROCEDURE.
- Zero or more clauses, such as WHERE, PRIMARY KEY, or UNIQUE PRIMARY INDEX.
- A semicolon.

In Teradata SQL, a *request* has several components, including the following.

- Request-level CLIv2 options parcel.
- One or more SQL statements.
- Metadata about the request data.

Request is a semantic concept. It defines a unit of work that is transmitted from a client application to Teradata Database in a single message. Each request either completes successfully (commits) or rolls back if it is not successful.

Note: All individual SQL statements are also individual SQL requests, but not all SQL requests are also SQL statements because one request can contain an unlimited number of SQL statements. This special case is called a multistatement request, and it is the primary superficial characteristic that distinguishes a statement from a request in the Teradata world.

Invoking SQL Statements

SQL provides 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 performs an action. The action can be on data or on a transaction or some other entity at a higher level than raw data.

Some examples of executable SQL statements include:

- SELECT
- CREATE TABLE
- COMMIT
- CONNECT
- PREPARE

Most executable SQL statements can be performed interactively from a terminal using an SQL query manager like BTEQ or Teradata SQL Assistant.

The following types of executable SQL commands cannot be performed interactively:

- Cursor control and declaration statements
- Dynamic SQL control statements
- Stored procedure control statements and condition handlers
- Connection control statements
- Special forms of SQL statements such as SELECT INTO

These statements can only be used within an embedded SQL or stored procedure application.

Nonexecutable SQL Statements

A nonexecutable SQL statement is one that declares an SQL statement, object, or host or local variable to the preprocessor or stored procedure compiler. Nonexecutable SQL statements are not processed during program execution.

Examples of nonexecutable SQL statements for embedded SQL applications include:

- DECLARE CURSOR
- BEGIN DECLARE SECTION
- END DECLARE SECTION
- EXEC SQL

Examples of nonexecutable SQL statements for stored procedures include:

- DECLARE CURSOR
- DECLARE

SQL Requests

A request to Teradata Database 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:

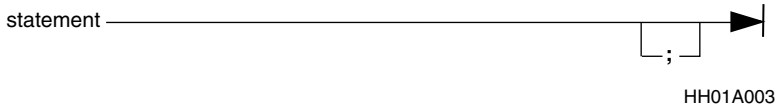
- Embedded in a client application program that is written in a procedural language.
- Embedded in a stored procedure.
- Entered interactively through BTEQ or Teradata SQL Assistant interfaces.
- Submitted in a BTEQ script as a batch job.
- Submitted through other supported methods (such as CLIv2, ODBC, 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.

For more information, see “Multistatement Requests” on page 69.

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 efficient 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 71.

ANSI Session Mode

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

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.

Following are the steps for abort processing:

1. Back out any changes made to the database as a result of any preceding statements in the transaction.
2. Delete any associated spooled output.
3. Release any associated locks.
4. Bypass any remaining statements in the transaction.

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

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*.

Teradata Extensions to ANSI SQL

In general, the Teradata Database SQL conforms closely to the American National Standards Institute/International Organization for Standardization (ANSI/ISO) SQL standard, but also supports extensions that enable users to take full advantage of the efficiency benefits of

parallelism. Teradata Database thus supports its own dialect of ANSI/ISO SQL guided by the following principles:

- When Teradata adds a new feature or enhancement to its SQL, the feature conforms to the ANSI/ISO SQL standard.
- When the difference between the Teradata SQL dialect and the ANSI/ISO SQL standard for a language feature is slight, Teradata defines the database feature so that ANSI/ISO SQL syntax is available as an option.
- When the difference between the Teradata SQL dialect and the ANSI/ISO SQL standard for a language feature is significant, the user is offered both Teradata and ANSI/ISO syntax and has the choice of operating either in Teradata or ANSI mode persistently or for a session, or of turning off the SQL Flagger.
- When Teradata adds a new feature or feature enhancement to Teradata SQL and that feature is not defined by the ANSI/ISO SQL standard, the feature is designed using the following criteria:
  - If other vendors offer a similar feature or feature extension, Teradata designs the new feature to broadly comply with other solutions but creates, where necessary, its own solution.
  - If other vendors do not offer a similar feature, Teradata designs the new feature as cleanly and generically as possible, creating syntax that does not violate any of the basic tenets of the ANSI/ISO SQL standard and that is not subject to major revisions so that it can accommodate future updates to the ANSI/ISO SQL standard.

Transaction Processing in ANSI Session Mode

In ANSI mode, transactions are always implicitly started and explicitly closed.

A transaction initiates when one of the following happens:

- The first SQL statement in a session executes.
- The first statement following the close of a transaction executes.

The COMMIT or ROLLBACK/ABORT statements close a transaction.

If a transaction includes a DDL statement, it must be the last statement in the transaction. DATABASE and SET SESSION are DDL statements. See “Rollback Processing” in *SQL Request and Transaction Processing*.

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

Two-Phase Commit

Sessions in ANSI session mode do not support Two-Phase Commit (2PC). If an attempt is made to use the 2PC protocol in ANSI session mode, the Logon process aborts and an error returns to the requestor.

Transaction Processing in Teradata Session Mode

A Teradata SQL transaction can be a single Teradata SQL statement, or a sequence of Teradata SQL statements, treated as a single unit of work.

Each request is processed as one of the following transaction types:

- Implicit
- Explicit
- Two-phase commit (2PC)

Implicit Transactions

An implicit transaction is a request that does not include the BEGIN TRANSACTION and END TRANSACTION statements. The implicit transaction starts and completes all within the SQL request: it is self-contained.

An implicit transaction can be a:

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

DDL statements are not valid in a multistatement request and are therefore not valid in an implicit multistatement transaction.

Explicit Transactions

In Teradata session mode, an explicit transaction contains one or more statements enclosed by BEGIN TRANSACTION and END TRANSACTION statements. The first BEGIN TRANSACTION initiates a transaction and the last END TRANSACTION terminates the transaction.

When multiple statements are included in an explicit transaction, you can only specify a DDL statement if it is the last statement in the series.

2PC Rules

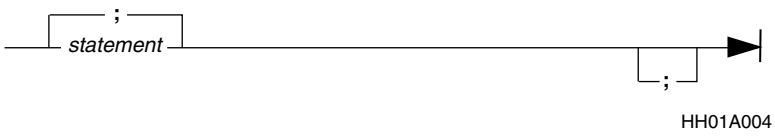
2PCprotocol is supported in Teradata session mode:

- A 2PC transaction contains one or more DML statements that affect multiple databases and are coordinated externally using the 2PC protocol.
- A DDL statement is not valid in a two-phase commit transaction.

Multistatement Requests

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

Syntax



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, a multistatement request is one form of implicit transaction. As such, the outcome of an implicit multistatement transaction is typically all-or-nothing. If one statement in the request fails, the entire implicit transaction fails and the system rolls it back.

Statement Independence for Simple INSERTs

When a multistatement request includes only simple INSERT statements, a failure of one or more INSERTs does not cause the entire request to be rolled back. In these cases, errors are reported for the INSERT statements that failed, so those statements can be resubmitted. INSERT statements that completed successfully are not rolled back.

This behavior is limited to requests submitted directly to the database using an SQL INSERT multistatement request or submitted using a JDBC application request.

Parallel Step Processing

Teradata Database can perform some requests in parallel (see "Parallel Steps" on page 71). 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.

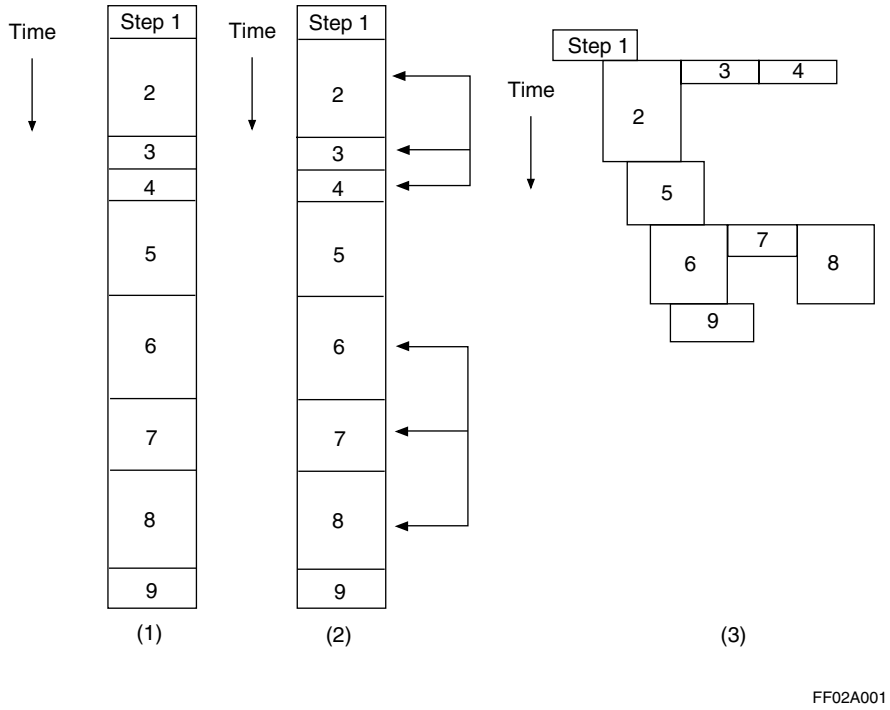
For example, if an INSERT step and a DELETE step are allowed to run in parallel, the AMP informs the PE that the DELETE step has progressed to the point where the INSERT step will not impact it adversely. This handshaking protocol also reduces the chance of a deadlock.

"Parallel Steps" on page 71 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



Related Topics

For more information on ...	See ...
multistatement requests, multistatment request processing	• *SQL Data Manipulation Language* • *SQL Request and Transaction Processing*

Iterated Requests

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 72.

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 iterated request must consist of a single DML statement from the following list:
  - ABORT
  - DELETE (excluding the positioned form of DELETE)
  - EXECUTE *macro_name*

    The fully-expanded macro must be equivalent to a single DML statement that is qualified to be in an iterated request.
  - INSERT
  - MERGE
  - ROLLBACK
  - SELECT
  - UPDATE (including atomic UPSERT, but excluding the positioned form of UPDATE)
- The DML statement must reference user-supplied input data, either as named fields in a USING modifier or as '?' parameter markers in a parameterized request.

Note: Iterated requests do not support the USING modifier with the TOP *n* operator.

- All the data records in a given request must use the same record layout. This restriction applies by necessity to requests where the record layout is given by a single USING modifier in the request text itself; but the restriction also applies to parameterized requests, where the request text has no USING modifier and does not fully specify the input record.
- The server processes the iterated request as if it were a single multistatement request, with each iteration and its response associated with a corresponding statement number.

Statement Independence for Simple INSERTs

When an iterated request includes only simple INSERT statements, a failure of one or more INSERTs does not cause the entire request to be rolled back. In these cases, errors are reported for the INSERT statements that failed, so those statements can be resubmitted. INSERT statements that completed successfully are not rolled back.

This behavior is limited to requests submitted directly to the database using an SQL INSERT multistatement request or submitted using a JDBC application request.

Related Topics

For more information on ...	See ...
iterated request processing	*SQL Request and Transaction Processing*
which DML statements can be specified in an iterated request	*SQL Data Manipulation Language*
CLIv2	• *Teradata Call-Level Interface Version 2 Reference for Network-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 User Guide*
BTEQ PACK command	*Basic Teradata Query Reference*

Dynamic and Static SQL

Dynamic SQL is a method of invoking an SQL statement by compiling and performing it at runtime from within an embedded SQL application program or a stored procedure. The specification of data to be manipulated by the statement is also determined at runtime. Static SQL is, by default, any method of invoking an SQL statement that is not dynamic.

ANSI Compliance

Dynamic SQL is ANSI 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.

For details, see *SQL Stored Procedures and Embedded SQL*.

Related Topics

For more information on ...	See ...
examples of dynamic SQL code in C, COBOL, and PL/I	Teradata Preprocessor2 for Embedded SQL Programmer Guide.
embedded SQL statements	*SQL Stored Procedures and Embedded SQL*.

Dynamic SQL in Stored Procedures

Stored procedures support of dynamic SQL statements is different from embedded SQL support.

Use the following statement to set up and invoke dynamic SQL in a stored procedure:

```
CALL DBC.SysExecSQL(string_expression)
```

where *string_expression* is any valid string expression that builds an SQL statement.

The string expression consists of string literals, status variables, local variables, input (IN and INOUT) parameters, and for-loop aliases. Dynamic SQL statements are not validated at compile time.

The resulting SQL statement cannot have status variables, local variables, parameters, for-loop aliases, or a USING or EXPLAIN modifier.

Example

The following example uses dynamic SQL within stored procedure source text:

```
CREATE PROCEDURE new_sales_table( my_table VARCHAR(30),
                                  my_database VARCHAR(30))
BEGIN
  DECLARE sales_columns VARCHAR(128)
    DEFAULT '(item INTEGER, price DECIMAL(8,2), sold INTEGER)';
  CALL DBC.SysExecSQL('CREATE TABLE ' || my_database ||
                '.' || my_table || sales_columns);
END;
```

A stored procedure can make any number of calls to SysExecSQL. The request text in the string expression can specify a multistatement request, but the call to SysExecSQL must be delimited by BEGIN REQUEST and END REQUEST keywords.

Because the request text of dynamic SQL statements can vary from execution to execution, dynamic SQL provides more usability and conciseness to the stored procedure definition.

Restrictions

Whether the creator, owner, or invoker of the stored procedure must have appropriate privileges on the objects that the stored procedure accesses depends on whether the CREATE

PROCEDURE statement includes the SQL SECURITY clause and which option the SQL SECURITY clause specifies.

The following SQL statements cannot be specified as dynamic SQL in stored procedures:

- ALTER PROCEDURE
- CALL
- CREATE PROCEDURE
- DATABASE
- EXPLAIN modifier
- HELP
- OPEN
- PREPARE
- REPLACE PROCEDURE
- SELECT
- SET ROLE
- SET SESSION ACCOUNT
- SET SESSION COLLATION
- SET SESSION DATEFORM
- SET TIME ZONE
- SHOW
- Cursor statements, including:
  - CLOSE
  - FETCH
  - OPEN

Related Topics

For rules and usage examples of dynamic SQL statements in stored procedures, see *SQL Stored Procedures and Embedded SQL*.

Using SELECT With Dynamic SQL

Unlike other executable SQL statements, SELECT returns information beyond statement responses and return codes to the requester.

DESCRIBE Statement

Because the requesting application needs to know how much (if any) data will be returned by a dynamically prepared SELECT, you must use an additional SQL statement, DESCRIBE, to make the application aware of the demographics of the data to be returned by the SELECT statement (see "DESCRIBE" in *SQL Stored Procedures and Embedded SQL*).

DESCRIBE writes this information to the SQLDA declared for the SELECT statement as follows.

THIS information ...	IS written to this field of SQLDA ...
number of values to be returned	SQLN
column name or label of n^{th} value	SQLVAR (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*

In this step, results tables are examined one row at a time using the selection cursor. This is because client programming languages do not support data in terms of sets, but only as individual records.

Event Processing Using Queue Tables

Teradata Database provides queue tables that you can use for event processing. Queue tables are base tables with first-in-first-out (FIFO) queue properties.

When you create a queue table, you define a timestamp column. You can query the queue table to retrieve data from the row with the oldest timestamp.

Usage

An application can perform peek, FIFO push, and FIFO pop operations on queue tables.

TO perform a ...	USE the ...
FIFO push	INSERT statement
FIFO pop	SELECT AND CONSUME statement
peek	SELECT statement

Here is an example of how an application can process events using queue tables:

- Define a trigger on a base table to insert a row into the queue table when the trigger fires.
- From the application, submit a SELECT AND CONSUME statement that waits for data in the queue table.
- When data arrives in the queue table, the waiting SELECT AND CONSUME statement returns a result to the application, which processes the event. Additionally, the row is deleted from the queue table.

Related Topics

For more information on ...	See ...
creating queue tables	the CREATE/REPLACE TABLE statement in *SQL Data Definition Language*
SELECT AND CONSUME	*SQL Data Manipulation Language*

Manipulating Nulls

Nulls are neither values nor do they signify values. They represent the absence of value. A null is a place holder indicating that no value is present.

You cannot solve for the value of a null because, by definition, it has no value. For example, the expression `NULL = NULL` has no meaning and therefore can never be true. A query that specifies the predicate `WHERE NULL = NULL` is not valid because it can never be true. The meaning of the comparison it specifies is not only unknown, but unknowable.

These properties make the use and interpretation of nulls in SQL problematic. The following sections outline the behavior of nulls for various SQL operations to help you to understand how to use them in data manipulation statements and to interpret the results those statements affect.

NULL Literals

See "NULL Keyword as a Literal" on page 34 for information on how to use the NULL keyword as a literal.

Nulls and DateTime and Interval Data

A DateTime or Interval value is either atomically null or it is not null. For example, you cannot have an interval of YEAR TO MONTH in which YEAR is null and MONTH is not.

Rules for the Result of Expressions That Contain Nulls

- When any component of a value expression is null, then the result is null.
- The result of a conditional expression that has a null component is unknown.
- If an operand of any arithmetic operator (such as + or -) or function (such as ABS or SQRT) is null, then the result of the operation or function is null with the exception of ZEROIFNULL. If the argument to ZEROIFNULL is NULL, then the result is 0.
- COALESCE, a special shorthand variant of the CASE expression, returns NULL if all its arguments evaluate to null. Otherwise, COALESCE returns the value of the first 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
```

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 80 and "Excluding Nulls" on page 80.

Using IS NULL is different from using the comparison operator =. When you use an operator like =, you specify a comparison between values or value expressions, whereas when you use the IS NULL operator, you specify an existence condition.

Rules for Nulls and CASE Expressions

- CASE and its related expressions COALESCE and NULLIF can return a null.
- NULL and null expressions are valid as the CASE test expression in a valued CASE expression.
- When testing for NULL, it is best to use a searched CASE expression using the IS NULL or IS NOT NULL operators in the WHEN clause.
- NULL and null expressions are valid as THEN clause conditions.

For details on the rules for nulls in CASE, NULLIF, and COALESCE expressions, see *SQL Functions, Operators, Expressions, and Predicates*.

Excluding Nulls

To exclude nulls from the results of a query, use the operator IS NOT NULL.

For example, to search for the names of all employees with a value other than null in the jobtitle column, enter the statement.

```
SELECT name
FROM employee
WHERE jobtitle IS NOT NULL ;
```

Searching for Nulls

To search for columns that contain nulls, use the operator IS NULL.

The IS NULL operator tests row data for the presence of nulls.

For example, to search for the names of all employees who have a null in the deptno column, you could enter the statement:

```
SELECT name
FROM employee
WHERE deptno IS NULL ;
```

This query produces the names of all employees with a null in the deptno field.

Searching for Nulls and 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
null	null

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

Replacing Nulls With Values on Return to Client in Record Mode

When Teradata Database returns information to a client system in record mode, nulls must be replaced with some value for the underlying column because client system languages do not recognize nulls.

The following table shows the values returned for various column data types.

Data Type	Substitute Value Returned for Null
CHARACTER(*n*) 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

Data Type	Substitute Value Returned for Null
BIGINT	0
INTEGER	
SMALLINT	
BYTEINT	
FLOAT	
DECIMAL	
REAL	
DOUBLE PRECISION	
NUMERIC	

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. BTEQ displays nulls as ?, which are not by themselves distinguishable from a CHAR or VARCHAR value of '?'.

Nulls and Aggregate Functions

With the important exception of COUNT(*), aggregate functions ignore nulls in their arguments. This treatment of nulls is very different from the way arithmetic operators and functions treat them.

This behavior can result in apparent nontransitive anomalies. For example, if there are nulls in either column A or column B (or both), then the following expression is virtually always true.

```
SUM(A) + (SUM B) <> SUM (A+B)
```

In other words, for the case of SUM, the result is never a simple iterated addition if there are nulls in the data being summed.

The only exception to this is the case in which the values for columns A and B are both null in the same rows, because in those cases the entire row is disregarded in the aggregation. This is a trivial case that does not violate the general rule.

The same is true, the necessary changes being made, for all the aggregate functions except COUNT(*).

If this property of nulls presents a problem, you can always do either of the following workarounds, each of which produces the desired result of the aggregate computation `SUM(A) + SUM(B) = SUM(A+B)`.

- Always define NUMERIC columns as `NOT NULL DEFAULT 0`.
- Use the ZEROIFNULL function within the aggregate function to convert any nulls to zeros for the computation, for example

  ```
  SUM(ZEROIFNULL(x) + ZEROIFNULL(y))
  ```

which produces the same result as this:

```
SUM(ZEROIFNULL(x) + ZEROIFNULL(y)).
```

COUNT(*) includes nulls in its result. For details, see *SQL Functions, Operators, Expressions, and Predicates*.

RANGE_N and CASE_N Functions

Nulls have special considerations in the RANGE_N and CASE_N functions. For details, see *SQL Functions, Operators, Expressions, and Predicates*.

Session Parameters

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

Parameter	Valid Keywords or System Variables
SQL Flagger	ON
	OFF
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

Parameter	Valid Keywords or System Variables
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 • UTF-8 • UTF-16 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 163.

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

To run transactions in this mode ...	Set the variable to this value ...
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*.

Collation Name	Description
HOST	The default. HOST collation defaults are: • 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.
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: Changing the performance group for your account changes the account name for accounting purposes because a performance group is part of an account name.

Date Form

You can change the format in which DATE data is imported or exported in your current session.

DATE data can be set to be treated either using the ANSI date format (DATEFORM=ANSIDATE) or using the Teradata date format (DATEFORM=INTEGERDATE).

For details, see “SET SESSION DATEFORM” in *SQL Data Definition Language*.

Setting the Client Character Set

To set the client character set, use one of the following:

- From BTEQ, use the BTEQ [.] SET SESSION CHARSET ‘*name*’ command.
- In a CLIv2 application, call CHARSET *name*.
- In the URL for selecting a Teradata JDBC driver connection to a Teradata Database, use the CHARSET=*name* database connection parameter.

where the '*name*' or *name* value is ASCII, EBCDIC, UTF-8, UTF-16, or a name assigned to the translation codes that define an available character set.

If not explicitly requested, the session default is the character set associated with the logon client. This is either the standard client default, or the character set assigned to the client by the database administrator.

HELP SESSION

The HELP SESSION statement identifies attributes in effect for the current session, including:

- Transaction mode
- Character set
- Collation sequence
- Date form
- Queryband

For details, see "HELP SESSION" in *SQL Data Definition Language*.

Session Management

Each session is logged on and off via calls to CLIv2 routines or through ODBC or JDBC, which offer a one-step logon-connect function.

Sessions are internally managed by dividing the session control functions into a series of single small steps that are executed in sequence to implement multithreaded tasking. This provides concurrent processing of multiple logon and logoff events, which can be any combination of individual users, and one or more concurrent sessions established by one or more users and applications.

Once connected and active, a session can be viewed as a work stream consisting of a series of requests between the client and server.

Session Pools

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 more information, see *Teradata Director Program Reference*.

Session Control

The major functions of session control are session logon and logoff.

Upon receiving a session request, the logon function verifies authorization and returns a yes or no response to the client.

The logoff function terminates any ongoing activity and deletes the session context.

Trusted Sessions

Applications that use connection pooling, 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 privileges 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

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 94 and "Error Response (ANSI Session Mode Only)" on page 93.

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

A statement that raises a completion condition can take further action such as querying the database and returning results to the requesting application, updating the database, initiating an SQL transaction, and so on.

Completion Condition	SQLSTATE Return Code	SQLCODE Return Code
Success	'00000'	0
Warning	'01901'	901
	'01800' to '01841'	901
	'01004'	902
No data found	'02000'	100

Related Topics

For more information, see:

- “Statement Responses” on page 91
- “Success Response” on page 92
- “Warning Response” on page 93

Return Codes for Stored Procedures

The return code values are different in the case of SQL control statements in stored procedures.

The return codes for stored procedures appear in the following table.

Completion Condition	SQLSTATE Return Code	SQLCODE Return Code
Successful completion	'00000'	0
Warning	SQLSTATE value corresponding to the warning code.	Teradata Database warning code.
No data found or any other Exception	SQLSTATE value corresponding to the error code.	Teradata Database error code.

How an Application Uses SQL Return Codes

An application program or stored procedure tests the status of a completed executable SQL statement to determine its status.

Condition	Action
Successful completion	None.

Condition	Action
Warning	The statement execution continues. If a warning condition handler is defined in the application, the handler executes.
No data found or any other exception	Whatever appropriate action is required by the exception. If an EXIT handler is defined for the exception, the statement execution terminates. If a CONTINUE handler is defined, execution continues after the remedial action.

Statement Responses

Response Types

Teradata Database responds to an SQL request with one of the following condition responses:

- Success response, with optional warning
- Failure response
- Error response (ANSI session mode only)

Depending on the type of statement, Teradata Database also responds with one or more rows of data.

Multistatement Responses

A response to a request that contains more than one statement, such as a macro, is not returned to the client until all statements in the request are successfully executed.

Returning a Response

The manner in which the response is returned depends on the interface that is being used.

For example, if an application is using a language preprocessor, then the activity count, warning code, error code, and fields from a selected row are returned directly to the program through its appropriately declared variables. If the application is a stored procedure, then the activity count is returned directly in the ACTIVITY_COUNT status variable.

If you are using BTEQ, then a success, error, or failure response is displayed automatically.

Response Condition Codes

SQL statements also return condition codes that are useful for handling errors and warnings in embedded SQL and stored procedure applications.

For information about SQL response condition codes, see the following in *SQL Stored Procedures and Embedded SQL*:

- SQLSTATE
- SQLCODE
- ACTIVITY_COUNT

Success Response

A success response contains an activity count that indicates the total number of rows involved in the result.

For example, the activity count for a SELECT statement is the total number of rows selected for the response. For a SELECT, CALL (when the stored procedure or external stored procedure creates result sets), COMMENT, or ECHO statement, the activity count is followed by the data that completes the response.

An activity count is meaningful for statements that return a result set, for example:

- SELECT
- INSERT
- UPDATE
- DELETE
- HELP
- SHOW
- EXPLAIN
- CREATE PROCEDURE
- REPLACE PROCEDURE
- CALL (when the stored procedure or external stored procedure creates result sets)

For other SQL statements, activity count is meaningless.

Example

The following interactive SELECT statement returns the successful response message.

```
SELECT AVG(f1)
FROM Inventory;

*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.
Average(f1)
-----------
         14
```

Warning Response

A success or OK response with a warning indicates either that an anomaly has occurred or informs the user about the anomaly and indicates how it can be important to the interpretation of the results returned.

Example

Assume the current session is running in ANSI session mode.

If nulls are included in the data for column f1, then the following interactive query returns the successful response message with a warning about the nulls.

```
SELECT AVG(f1) FROM Inventory;

 *** Query completed. One row found. One column returned.
 *** Warning: 2892 Null value eliminated in set function.
 *** Total elapsed time was 1 second.

Average(f1)
-----------
         14
```

This warning response is not generated if the session is running in Teradata session mode.

Error Response (ANSI Session Mode Only)

An error response occurs when a query anomaly is severe enough to prevent the correct processing of the request.

In ANSI session mode, an error for a request causes the request to rollback, and not the entire transaction.

Example 1

The following command returns the error message immediately following.

```
.SET SESSION TRANS ANSI;

 *** Error: You must not be logged on .logoff to change the SQLFLAG
or TRANSACTION settings.
```

Example 2

Assume that the session is running in ANSI session mode, and the following table is defined:

```
CREATE MULTISET TABLE inv, FALLBACK,
   NO BEFORE JOURNAL,
   NO AFTER JOURNAL
   (
   item INTEGER CHECK ((item >=10) AND (item <= 20) ))
PRIMARY INDEX (item);
```

You insert a value of 12 into the item column of the inv table.

This is valid because the defined integer check specifies that any integer between 10 and 20 (inclusive) is valid.

```
INSERT INTO inv (12);
```

The following results message returns.

```
 *** Insert completed. One row added....
```

You insert a value of 9 into the item column of the inv table.

This is not valid because the defined integer check specifies that any integer with a value less than 10 is not valid.

```
INSERT INTO inv (9);
```

The following error response returns:

```
 *** Error 5317 Check constraint violation: Check error in field
inv.item.
```

You commit the current transaction:

```
COMMIT;
```

The following results message returns:

```
 *** COMMIT done. ...
```

You select all rows from the inv table:

```
SELECT * FROM inv;
```

The following results message returns:

```
 *** Query completed. One row found. One column returned.
   item
-------
     12
```

Failure Response

A failure response is a severe error. The response includes a statement number, an error code, and an associated text string describing the cause of the failure.

Teradata Session Mode

In Teradata session mode, a failure causes the system to roll back the entire transaction.

If one statement in a macro fails, a single failure response is returned to the client, and the results of any previous statements in the transaction are backed out.

ANSI Session Mode

In ANSI session mode, a failure causes the system to roll back the entire transaction, for example, when the current request:

- Results in a deadlock
- Performs a DDL statement that aborts
- Executes an explicit ROLLBACK or ABORT statement

Example 1

The following SELECT statement

```
SELECT * FROM Inventory:;
```

in BTEQ, returns the failure response message:

```
*** Failure 3706 Syntax error: expected something between the word
'Inventory' and ':'.
                Statement# 1, Info =20
*** Total elapsed time was 1 second.
```

Example 2

Assume that the session is running in ANSI session mode, and the following table is defined:

```
CREATE MULTISET TABLE inv, FALLBACK,
   NO BEFORE JOURNAL,
   NO AFTER JOURNAL
   (
   item INTEGER CHECK ((item >=10) AND (item <= 20) ))
PRIMARY INDEX (item);
```

You insert a value of 12 into the item column of the inv table.

This is valid because the defined integer check specifies that any integer between 10 and 20 (inclusive) is valid.

```
INSERT INTO inv (12);
```

The following results message returns.

```
*** Insert completed. One row added....
```

You commit the current transaction:

```
COMMIT;
```

The following results message returns:

```
*** COMMIT done. ...
```

You insert a valid value of 15 into the item column of the inv table:

```
INSERT INTO inv (15);
```

The following results message returns.

```
*** Insert completed. One row added....
```

You can use the ABORT statement to cause the system to roll back the transaction:

```
ABORT;
```

The following failure message returns:

```
*** Failure 3514 User-generated transaction ABORT.
                Statement# 1, Info =0
```

You select all rows from the inv table:

```
SELECT * FROM inv;
```

The following results message returns:

```
 *** Query completed. One row found. One column returned.

   item
-------
     12
```

CHAPTER 4 Query Processing

This chapter discusses query processing, including single AMP requests and all AMP requests, and table access methods available to the Optimizer.

Queries and AMPs

An SQL query, which includes DELETE, INSERT, MERGE, and UPDATE as well as SELECT, can affect one AMP, several AMPs, or all AMPs in the configuration.

IF a query ...	THEN ...
(involving a single table) uses a unique primary index (UPI)	the row hash can be used to identify a single AMP. 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:

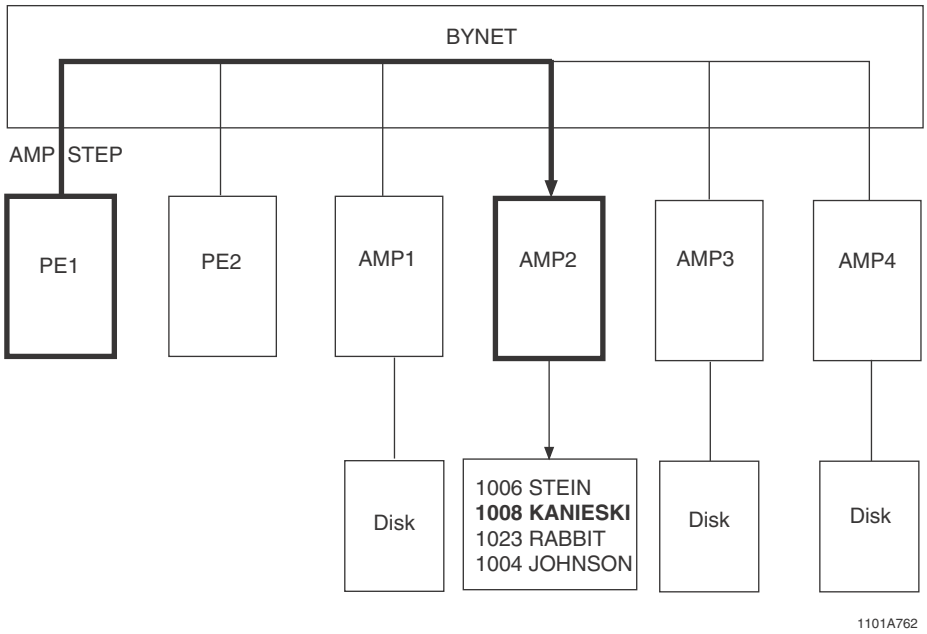
```
SELECT last_name
FROM Employee
WHERE employee_number = 1008;
```

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).

See “Flow Diagram of a Single AMP Request” on page 99 for a flow diagram that illustrates this process.

Flow Diagram of a Single AMP Request

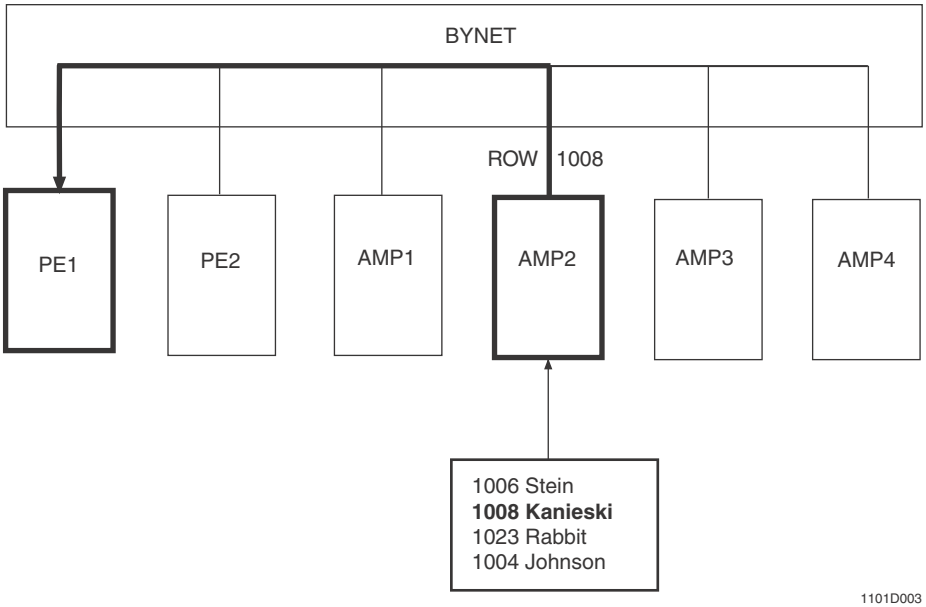
Only one BYNET is shown to simplify the illustration.



Assuming that AMP2 has the row, it accepts the message. As shown in the next diagram, AMP2 retrieves the row from disk, includes the row and the PE identification in a return message, and sends the message back to PE1 via the BYNET. PE1 accepts the message and returns the response row to the requesting application.

Flow Diagram of a Single AMP Response to Requesting PE

The following diagram illustrates a single AMP request with partition elimination.



All AMP Request

Assume PE1 receives a SELECT statement that specifies a range of primary index values as a search condition as shown in the following example:

```
SELECT last_name, employee_number
FROM employee
WHERE employee_number BETWEEEN 1001 AND 1010
ORDER BY last_name;
```

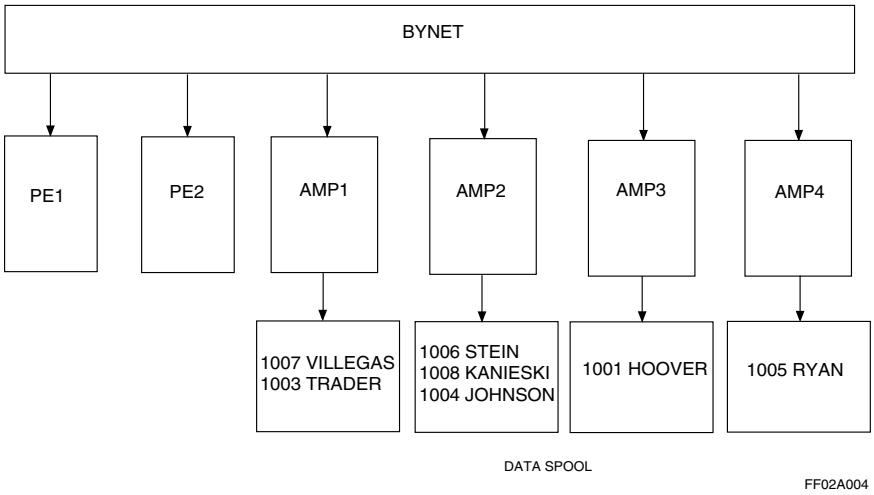
In this case, each value hashes differently, and all AMPs must search for the qualifying rows.

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

- Retrieve rows between 1001 and 1010
- Sort ascending on last_name
- Merge the sorted rows to form the answer set

PE1 then builds a message for each AMP step and puts that message onto the BYNET. Typically, each AMP step is completed before the next one begins; note, however, that some queries can generate parallel steps.

When PE1 puts the message for the first AMP step on the BYNET, that message is broadcast to all processors as illustrated in the following diagram.



The process is:

1 All AMPs accept the message, but the PEs do not.

2 Each AMP checks for qualifying rows on its disk storage units.

3 If any qualifying rows are found, the data in the requested columns is converted to the client format and copied to a spool file.

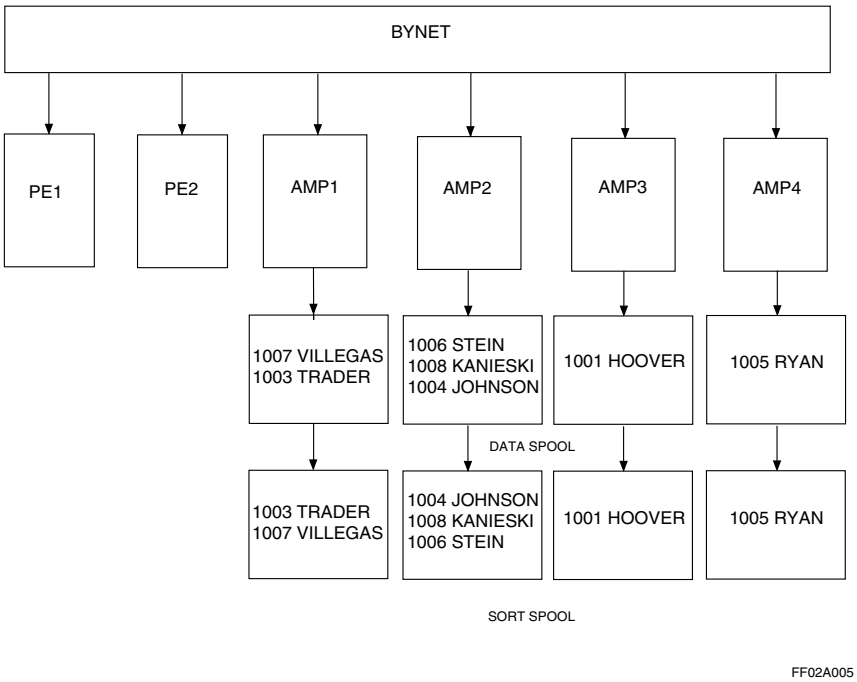
4 Each AMP completes the step, whether rows were found or not, and puts a completion message on the BYNET.

The completion messages flow across the BYNET to PE1.

5 When all AMPs have returned a completion message, PE1 transmits a message containing AMP Step 2 to the BYNET.

Upon receipt of Step 2, the AMPs sort their individual answer sets into ascending sequence by *last_name* , as illustrated in the following diagram.

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

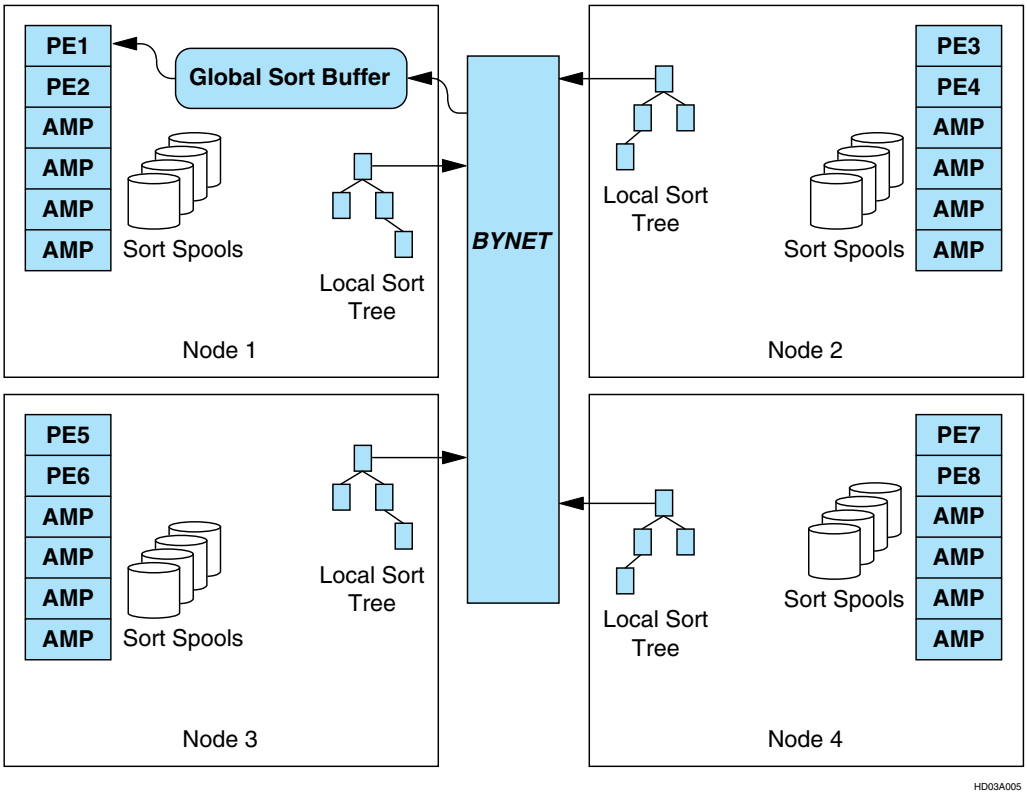


6 Each AMP sorts its answer set, then puts a completion message on the BYNET.

7 When PE1 has received all completion messages for Step 2, it sends a message containing AMP Step 3.

8 Upon receipt of Step 3, each AMP copies the first block from its sorted spool to the BYNET.

Because there can be multiple AMPs on a single node, each node might be required to handle sort spools from multiple AMPs (see the diagram at the top of the next page).



9 Nodes that contain multiple AMPs must first perform an intermediate sort of the spools generated by each of the local AMPs.

When the local sort is complete on each node, the lowest sorting row from each node is sent over the BYNET to PE1. From this point on, PE1 acts as the Merge coordinator among all the participating nodes.

10 The Merge continues with PE1 building a globally sorted buffer.

When this buffer fills, PE1 forwards it to the application and begins building subsequent buffers.

11 When a participant node has exhausted its sort spool, it sends a Done message to PE1.

This causes PE1 to prune this node from the set of Merge participants.

When there are no remaining Merge participants, PE1 sends the final buffer to the application along with an End Of File message.

Partition Elimination

A PPI can increase query efficiency through partition elimination, where partitions can automatically be skipped because they cannot contain qualifying rows.

Teradata Database supports several types of partition elimination.

Type	Description
Static	Based on constant conditions such as equality or inequality on the partitioning columns.
Dynamic	The partitions to eliminate cannot be determined until the query is executed and the data is scanned.
Delayed	Occurs with conditions comparing a partitioning column to a USING variable or built-in function such as CURRENT_DATE, where the Optimizer builds a somewhat generalized plan for the query but delays partition elimination until specific values of USING variables and built-in functions are known.

The degree of partition elimination depends on the:

- Partitioning expressions for the primary index of the table
- Conditions in the query
- Ability of the Optimizer to detect partition elimination

It is not always required that all values of the partitioning columns be specified in a query to have partition elimination occur.

IF a query ...	THEN ...
specifies values for all the primary index columns	the AMP where the rows reside can be determined and only a single AMP is accessed. • If conditions are not specified on the partitioning columns, each partition can be probed to find the rows based on the hash value. • If conditions are also specified on the partitioning columns, partition elimination may reduce the number of partitions to be probed on that AMP. For an illustration, see "Single AMP Request With Partition Elimination" on page 103.
does not specify the values for all the primary index columns	an all-AMP full file scan is required for a table with an NPPI. However, with a PPI, if conditions are specified on the partitioning columns, partition elimination may reduce an all-AMP full file scan to an all-AMP scan of only the non-eliminated partitions.

Single AMP Request With Partition Elimination

If a SELECT specifies values for all the primary index columns, the AMP where the rows reside can be determined and only a single AMP is accessed.

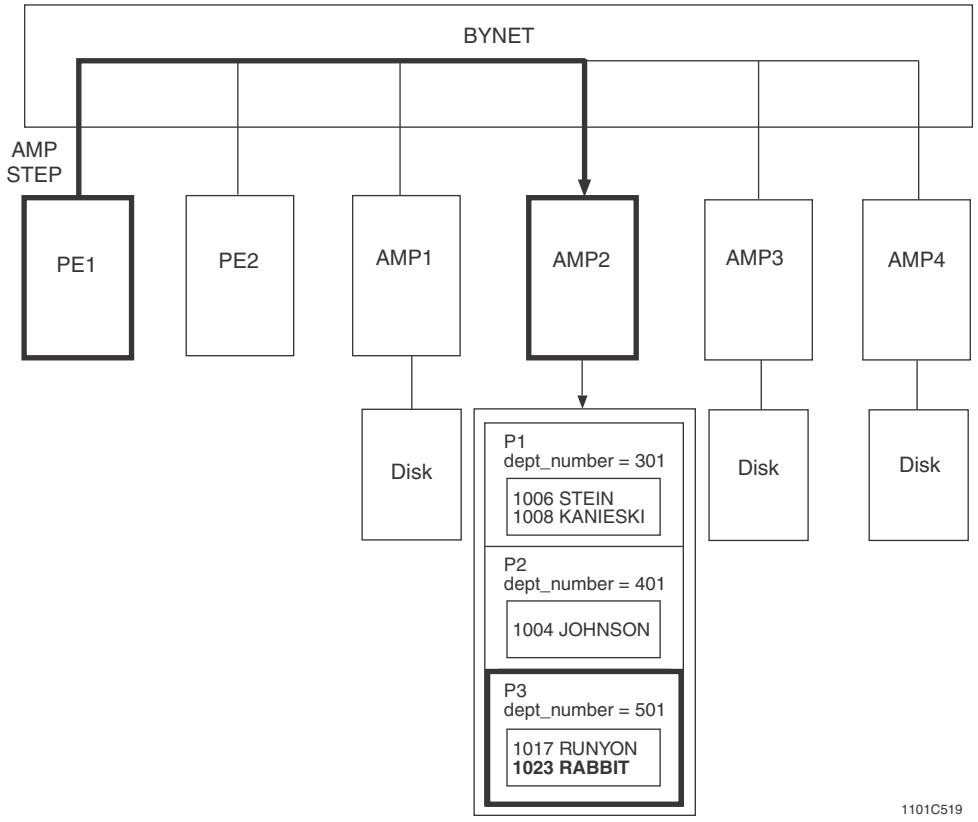
If conditions are also specified on the partitioning columns, partition elimination may reduce the number of partitions to be probed on that AMP.

Suppose the Employee table is defined with a single-level PPI where the partitioning column is dept_number.

Assume that a PE receives the following SELECT statement:

```
SELECT last_name
FROM Employee
WHERE employee_number = 1023
AND dept_number = 501;
```

The following flow diagram illustrates this process.



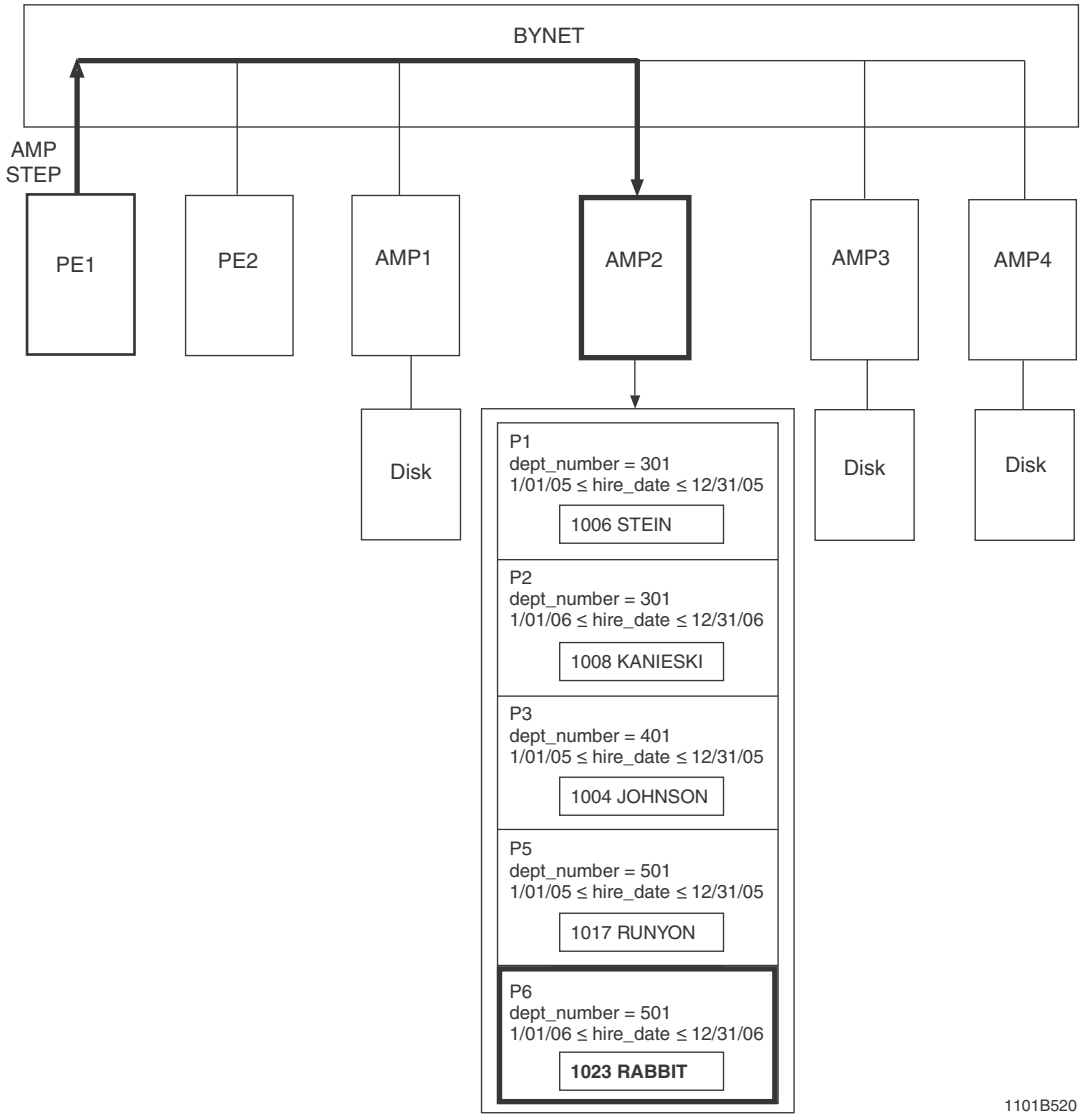
The AMP Step includes the list of partitions (in this case, P3) to access. Partition elimination (in this case, static partition elimination) reduces access to the partitions that satisfy the query requirements. In each partition in the list (in this case, only P3), look for rows with a given row hash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

Partition elimination is similar for the Employee table with a multilevel PPI where one partitioning expression uses the dept_number column and another partitioning expression uses the hire_date column.

Assume that a PE receives the following SELECT statement:

```
SELECT last_name
FROM Employee
WHERE employee_number = 1023
AND dept_number = 501
AND hire_date BETWEEN DATE '2006-01-01' AND DATE '2006-12-31';
```

The following flow diagram illustrates this process.



No one was hired in department number 401 in 2006, so partition P4 is empty.

For more information on partition elimination, see *Database Design*.

Table Access

Teradata Database uses indexes and partitions to access the rows of a table. If indexed or partitioned access is not suitable for a query, or if a query accesses a NoPI table that does not have an index defined on it, the result is a full-table scan.

Access Methods

The following table access methods are available to the Optimizer:

- Unique Primary Index
- Unique Partitioned Primary Index
- Nonunique Primary Index
- Nonunique Partitioned Primary Index
- Unique Secondary Index
- Nonunique Secondary Index
- Join Index
- Hash Index
- Full-Table Scan
- Partition Scan

Effects of Conditions in WHERE Clause

For a query on a table that has an index defined on it, the predicates or conditions that appear in the WHERE clause of the query determine whether the system can use row hashing, or do a table scan with partition elimination, or whether it must do a full-table scan.

The following functions are applied to rows identified by the WHERE clause, and have no effect on the selection of rows from the base table:

- GROUP BY
- HAVING
- INTERSECT
- MINUS/EXCEPT
- ORDER BY
- QUALIFY
- SAMPLE
- UNION
- WITH ... BY
- WITH

Statements that specify any of the following WHERE clause conditions result in full-table scans (FTS). If the table has a PPI, partition elimination might reduce the FTS access to only the affected partitions.

- nonequality comparisons
- *column_name* IS NOT NULL
- *column_name* NOT IN (explicit list of values)
- *column_name* NOT IN (subquery)
- *column_name* BETWEEN ... AND ...
- condition_1 OR condition_2
- NOT condition_1
- *column_name* LIKE
- column_1 || column_2 = value
- table1.column_x = table1.column_y
- table1.column_x [computation] = value
- table1.column_x [computation] - table1.column_y
- INDEX (*column_name*)
- SUBSTR (*column_name*)
- SUM
- MIN
- MAX
- AVG
- DISTINCT
- COUNT
- ANY
- ALL
- missing WHERE clause

The type of table access that the system uses when statements specify any of the following WHERE clause conditions depends on whether the column or columns are indexed, the type of index, and its selectivity:

- *column_name* = value or constant expression
- *column_name* IS NULL
- *column_name* IN (explicit list of values)
- *column_name* IN (subquery)
- condition_1 AND condition_2
- different data types
- table1.column_x = table2.column_x

In summary, a query influences processing choices:

- A full-table scan (possibly with partition elimination if the table has a PPI) is required if the query includes an implicit range of values, such as in the following WHERE examples.

  When a small BETWEEN range is specified, the Optimizer can use row hashing rather than a full-table scan.

  ```
  ... WHERE column_name [BETWEEN <, >, <>, <=, >=]
  ... WHERE column_name [NOT] IN (SELECT...)
  ... WHERE column_name NOT IN (val1, val2 [,val3])
  ```
- Row hashing can be used if the query includes an explicit value, as shown in the following WHERE examples:

  ```
  ... WHERE column_name = val
  ... WHERE column_name IN (val1, val2, [,val3])
  ```

Related Topics

For more information on ...	See ...
the efficiency, number of AMPs used, and the number of rows accessed by all table access methods	*Database Design*
strengths and weaknesses of table access methods	*Introduction to Teradata*
full-table scans	"Full-Table Scans" on page 107

Full-Table Scans

A full-table scan is a retrieval mechanism that touches all rows in a table.

Teradata Database always uses a full-table scan to access the data of a table if a query:

- Accesses a NoPI table that does not have an index defined on it
- Does not specify a WHERE clause

Even when results are qualified using a WHERE clause, indexed or partitioned access may not be suitable for a query, and a full-table scan may result.

A full-table scan is always an all-AMP operation, and should be avoided when possible. Full-table scans may generate spool files that can have as many rows as the base table.

Full-table scans are not something to fear, however. The architecture that Teradata Database uses makes a full-table scan an efficient procedure, and optimization is scalable based on the number of AMPs defined for the system. The sorts of unplanned, ad hoc queries that characterize the data warehouse process, and that often are not supported by indexes, perform very effectively for Teradata Database using full-table scans.

Accessing Rows in a Full-Table Scan

Because full-table scans necessarily touch every row on every AMP, they do not use the following mechanisms for locating rows:

- Hashing algorithm and hash map
- Primary indexes
- Secondary indexes or their subtables
- Partitioning

Instead, a full-table scan uses the Master Index and Cylinder Index file system tables to locate each data block. Each row within a data block is located by a forward scan.

Because rows from different tables are never mixed within the same data block and because rows never span blocks, an AMP can scan up to 128K bytes of the table on each block read, making a full-table scan a very efficient operation. Data block read-ahead and cylinder reads can also increase efficiency.

Related Topics

For more information on ...	See ...
full-table scans	*Database Design*
cylinder reads	*Database Administration*
enabling data block read-ahead operations	DBS Control Utility in *Utilities*

Collecting Statistics

The COLLECT STATISTICS (Optimizer form) statement collects demographic data for one or more columns of a base table, hash index, or join index, computes a statistical profile of the collected data, and stores the synopsis in the data dictionary.

The Optimizer uses the synopsis data when it generates its table access and join plans.

Usage

You should collect statistics on newly created, empty data tables. An empty collection defines the columns, indexes, and synoptic data structure for loaded collections. You can easily collect statistics again after the table is populated for prototyping, and again when it is in production.

You can collect statistics on a:

- Unique index, which can be:
  - Primary or secondary
  - Single or multiple column
  - Partitioned or nonpartitioned
- Nonunique index, which can be:
  - Primary or secondary
  - Single or multiple column
  - Partitioned or nonpartitioned
  - With or without COMPRESS fields
- Non-indexed column or set of columns, which can be:
  - Partitioned or nonpartitioned
  - With or without COMPRESS fields
- Join index
- Hash index
- NoPI table
- Temporary table
  - If you specify the TEMPORARY keyword but a materialized table does not exist, the system first materializes an instance based on the column names and indexes you specify. This means that after a true instance is created, you can update (re-collect) statistics on the columns by entering COLLECT STATISTICS and the TEMPORARY keyword without having to specify the desired columns and index.
  - If you omit the TEMPORARY keyword but the table is a temporary table, statistics are collected for an empty base table rather than the materialized instance.
- Sample (system-selected percentage) of the rows of a data table or index, to detect data skew and dynamically increase the sample size when found.
  - The system does not store both sampled and defined statistics for the same index or column set. Once sampled statistics have been collected, implicit re-collection hits the same columns and indexes, and operates in the same mode. To change this, specify any keywords or options and name the columns and/or indexes.

Related Topics

For more information on ...	See ...
using the COLLECT STATISTICS statement	*SQL Data Definition Language*
collecting statistics on a join index	*Database Design*
collecting statistics on a hash index	
when to collect statistics on base table columns instead of hash index columns	
database administration and collecting statistics	*Database Administration*

APPENDIX A Notation Conventions

This appendix describes the notation conventions used in this book.

Throughout this book, three conventions describe the SQL syntax and code:

- Syntax diagrams, used to describe SQL syntax form, including options. See "Syntax Diagram Conventions" on page 111.
- Square braces in the text, used to represent options. The indicated parentheses are required when you specify options.

  For example:
  - DECIMAL [(n[,m])] means the decimal data type can be defined optionally:
    - without specifying the precision value *n* or scale value *m* specifying precision (n) only
    - specifying both values (n and m)
    - you cannot specify scale without first defining precision.
  - CHARACTER [(n)] means that use of (n) is optional.

    The values for *n* and *m* are integers in all cases
  - Japanese character code shorthand notation, used to represent unprintable Japanese characters. See "Character Shorthand Notation In This Book" on page 116.

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 Linux command. • Mixed Case letters represent exceptions to uppercase and lowercase rules. The exceptions are noted in the syntax explanation. • *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.

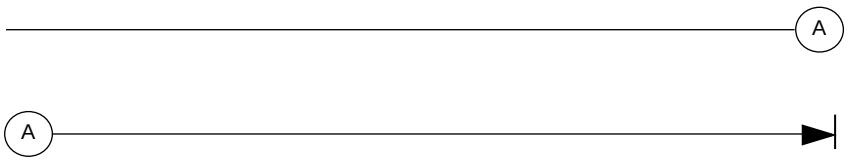
Paths

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.

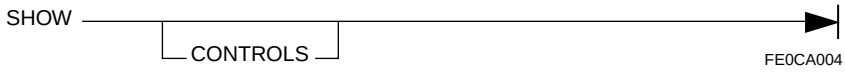
Required Entries

Required entries appear on the main path:

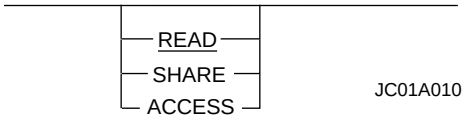
If you can choose from more than one entry, the choices appear vertically, in a stack. The first entry appears on the main path:

Optional Entries

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



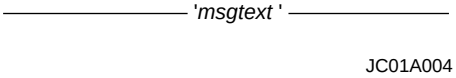
If you can optionally choose from more than one entry, all the choices appear below the main path:



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 apostrophes:



Abbreviations

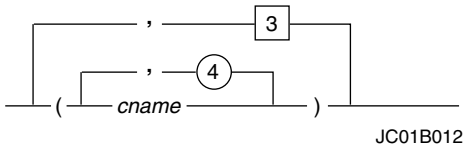
If a keyword or a reserved word has a valid abbreviation, the unabbreviated form always appears on the main path. The shortest valid abbreviation appears beneath.

In the above syntax, the following formats are valid:

- SHOW CONTROLS
- SHOW CONTROL

Loops

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



Read loops from right to left.

The following conventions apply to loops:

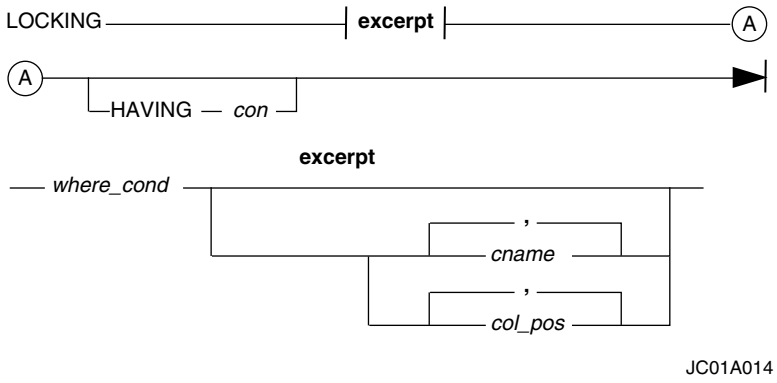
IF...	THEN...
there is a maximum number of entries allowed	the number appears in a circle on the return path. In the example, you may type *cname* a maximum of 4 times.
there is a minimum number of entries required	the number appears in a square on the return path. In the example, you must type at least 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

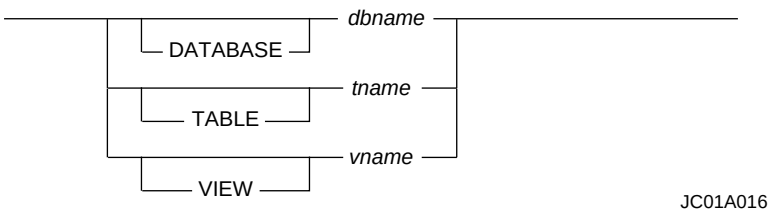
Sometimes a piece of a syntax phrase is too large to fit into the diagram. Such a phrase is indicated by a break in the path, marked by (|) terminators on each side of the break. The name for the excerpted piece appears between the terminators in boldface type.

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

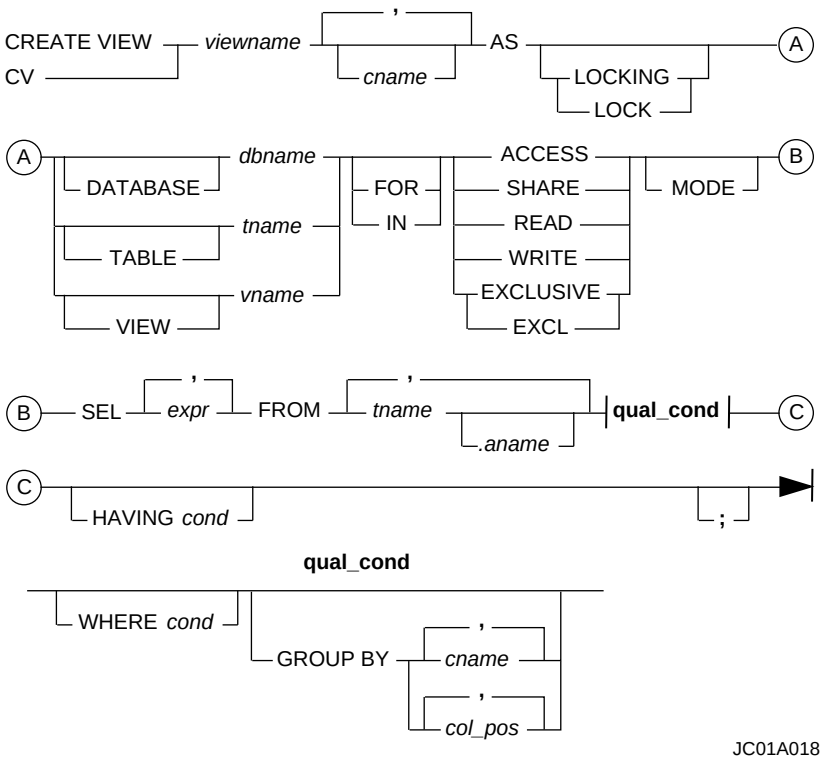


Diagram Identifier

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 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.

You can use the SQLRestrictedWords view or the SQLRestrictedWords_TBF function to retrieve a list of all Teradata SQL restricted words, including the Teradata reserved words. For more information, see "SQLRestrictedWords Function" on page 128.

The following notation appears in the list of reserved words.

Notation	Explanation
AR superscript	This word is also an ANSI SQL:2008 reserved word.
AN superscript	This word is also an ANSI SQL:2008 nonreserved word.
Asterisk (*)	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 COALESCE[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]
NONTEMPORAL NORMALIZE[AR] NOT[AR] NOWAIT NULL[AR] NULLIF[AR]
NULLIFZERO NUMBER[*] 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 TD_ANYTYPE[*] 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] TRANSACTIONTIME 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 UNTIL_CLOSED UPD UPDATE[AR] UPPER[AR] UPPERCASE USER[AR] USING[AR]

VALIDTIME 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 they may in the future be used as reserved keywords.

You can use the SQLRestrictedWords view or the SQLRestrictedWords_TBF function to retrieve a list of all Teradata SQL restricted words, including the Teradata nonreserved words. For more information, see "SQLRestrictedWords Function" on page 128.

The following notation appears in the list of nonreserved words.

Notation	Explanation
AR superscript	This word is also an ANSI SQL:2008 reserved word.
AN superscript	This word is also an ANSI SQL:2008 nonreserved word.
Plus sign (+)	This word must always follow the AS keyword if it appears as an alias name in the select list of a SELECT statement.
Asterisk (*)	This word is new for this release.

ACCESS AG ALLOCATE[AR] ALLOCATION ALLPARAMS ALWAYS[AN]
ANALYSIS ANCHOR APPLNAME ARCHIVE ARRAY[*] ASCII
ASSIGNMENT[AN] ATTR ATTRIBUTE[AN] ATTRIBUTES[AN] ATTRS
ATTR AUTO[*] AUTOTEMP[*]

BLOCKCOMPRESSION[*]

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

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

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

FINAL[AN] FOLLOWING[AN] FRIDAY

G[AN] GLOBAL[AR] GLOP

HIGH HOST

IFP INCREMENT[AN] INDEXESPERTABLE INDEXMAINTMODE INIT
INLINE INSTANTIABLE[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] MANUAL[*] MATCHED[AN] MAXCHAR MAXINTERVALS[*]
MAXLOGONATTEMPTS MAXVALUE[AN] MAXVALUELENGTH[*] MEDIUM
MESSAGE_TEXT[AN] MINCHAR MINVALUE[AN] MODIFIED MONDAY
MONTH_BEGIN MONTH_END MORE[AN] MULTINATIONAL

NAME[AN] NEVER[*] NODDLTEXT NONOPTCOST NONOPTINIT NONSEQUENCED

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

P_INTERSECT[+] P_NORMALIZE[+] PARAMID PARTITION[AR]
PARTITIONED PARTITION#Ln (where n=1-62)[*]
PERIOD PRECEDES[+] PRECEDING[AN] PRINT PRIOR[AN] PROTECTED

QUARTER_BEGIN[*] QUARTER_END[*] QUERY QUERY_BAND

RANDOMIZED RANGE[AR] RANGE#Ln (where n=1-15) RDIFF[+]
READ[AN] RECALC REPLACEMENT RESET RESTRICTWORDS RETAIN
RETURNED_SQLSTATE[AN] REUSE ROW_COUNT[AN] RU RULES RULESET

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

TARGET TD_GENERAL TD_INTERNAL TEMPORAL_DATE

TEMPORAL_TIMESTAMP TEXT THROUGH THURSDAY TIES[AN]
TPA TRANSACTION_ACTIVE[AN] TRUST_ONLY TUESDAY

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

VARRAY[*]

WEEK_BEGIN[*] WEEK_END[*] WARNING WEDNESDAY WORKLOAD WRITE[AN]

XML

YEAR_BEGIN[*] YEAR_END[*]

Teradata Future Reserved Words

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

You can use the SQLRestrictedWords view or the SQLRestrictedWords_TBF function to retrieve a list of all Teradata SQL restricted words, including the Teradata future reserved words. For more information, see "SQLRestrictedWords Function" on page 128.

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

Notation	Explanation
AR superscript	This word is also an ANSI SQL:2008 reserved word.
AN superscript	This word is also an ANSI SQL:2008 nonreserved word.
Asterisk (*)	New word for this release.

ALIAS

DESCRIPTOR[AN]

GO[AN] GOTO[AN]

INDICATOR[AR]

PRIVATE

WAIT

Teradata Parallel Transporter Reserved Keywords

Teradata Parallel Transporter (PT) reserved keywords cannot be used in Teradata PT job scripts as identifiers. Identifiers are non-quoted strings used for column names, script object names, or attributes.

The following notation appears in the list of Teradata PT reserved keywords.

Notation	Explanation
TR superscript	This word is also a Teradata reserved word.
Asterisk (*)	New word for this release.

ACCOUNT[TR] ALL[TR] ALLOW AND[TR] ANSIDATE[TR] APPLY ARRAY
ATTRIBUTES AS[TR] ATTR

BIGINT[TR] BLOB[TR] BY[TR] BYTE[TR] BYTEINT[TR]

CASE[TR] CAST[TR] CATEGORY CHAR[TR] CHARACTER[TR] CHARACTERS[TR]
CHARS[TR] CHECK[TR] CHECKPOINT[TR] CLOB[TR] COMMAND COMPONENT
CONSUMER

DATABASE[TR] DATACONNECTOR DATAGENERATOR DATE[TR] DATETIME
DAY[TR] DBMS DDL DEBUG DEC[TR] DECIMAL[TR] DEFAULT[TR]
DEFERRED[TR] DEFINE DELETE[TR] DELIM DELIMITER DELTATIME

DESCRIPTION DIRECTORY DISABLE DISALLOW DISCARD DUPLICATE ELSE[TR] ENABLE END[TR] EXECUTE[TR] EXPORT EXTERNAL[TR] EXTRA

FALSE FASTEXPORT[TR] FASTLOAD FILE FILTER FLOAT[TR] FOR[TR] FROM[TR]

GENERIC GRAPHIC[TR]

HOUR[TR]

ID IF[TR] IGNORE INCLUDE INFO INMOD INPUT[TR] INSERT INSERTER INSTANCE INT[TR] INTDATE INTEGER[TR] INTERFACE INTERVAL[TR] INTO[TR] IS[TR]

JOB

KEEP

LANGUAGE[TR] LATENCY LENGTH LIBRARY LOAD LOCAL[TR] LONG[TR] LOG[TR] LOGGER LOGON[TR] LOGREP LANGORT

MACROCHARSET[*] MARK MAX[TR] MAXIMUM[TR] MESSAGE METADATA MIN[TR] MINUTE[TR] MINUTES MISSING MODE[TR] MONTH[TR] MSGCATALOG MSGNUMBER MSGTEXT MULTILOAD MULTIPHASE MSGTEXT MULTILOAD MULTIPHASE

NAME NODE NODEBUG NODENAME NOINFO NOT[TR] NOTRACE NULL[TR] NULLIF[TR] NUMERIC[TR]

ODBC OFF[TR] OFFSET ON[TR] OPERATOR OR[TR] OS OTHER OUTMOD OUTPUT

PARALLEL PASSWORD[TR] PATH PERIOD PROCESSID PRODUCER PXOPER

REMOTE RESTARTABLE ROWS[TR]

SCHEMA SEC SECOND[TR] SECONDS SELECT[TR] SELECTOR SEQUENTIAL SERIALIZE SET[TR] SMALLINT[TR] SOURCE STANDALONE STEP STREAM SUPPORT SYSTEM

TABLE[TR] TASKID TASKNAME TERADATA THEN[TR] TIME[TR]
TIMESTAMP[TR] TO[TR] TRACE[TR] TRUE TYPE[TR]

UNION[TR] UNKNOWN UPDATE[TR] USE USER[TR] USING[TR]

VARBYTE[TR] VARCHAR[TR] VARGRAPHIC[TR] VARYING[TR] VERSION VIA
VIEW[TR]

WHEN[TR] WHERE[TR] WITH[TR] WORKING

XSMOD

YEAR[TR]

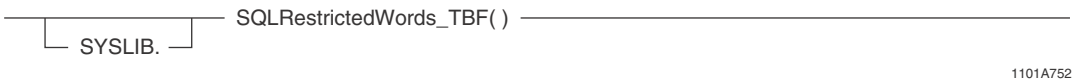
ZONE[TR]

SQLRestrictedWords Function

Purpose

Returns a table of Teradata SQL restricted words including Teradata reserved words, Teradata nonreserved keywords and Teradata future reserved words.

Syntax



1101A752

ANSI Compliance

SQLRestrictedWords_TBF is a Teradata extension to the ANSI SQL:2008 standard.

Invocation

Before you can use this function, you must run the Database Initialization Program (DIP) utility and execute the DIPUDT script. The DIPALL or DIPUDT script will create the SQLRestrictedWords_TBF function and the associated SQLRestrictedWords view in the SYSLIB database. For more information about the DIP utility, see *Utilities*.

Teradata Database searches for the SQLRestrictedWords_TBF function and the associated view in the current database. If the function or view is not found, Teradata Database searches in the SYSLIB database. Alternatively, you may invoke the function and view by using the fully qualified syntax:

- SYSLIB.SQLRestrictedWords_TBF()
- SYSLIB.SQLRestrictedWords

Description

SQLRestrictedWords_TBF is a table function that returns a table of Teradata SQL restricted words. You can use this function to retrieve the restricted words for the current or a specified Teradata Database release. This is useful for migration or upgrade planning, or in cases when you want to back down or revert to a prior Teradata Database release.

SQLRestrictedWords_TBF returns only Teradata reserved words, Teradata nonreserved keywords, and Teradata future reserved words. It does not return ANSI reserved words or ANSI nonreserved words. However, if a Teradata restricted word is also an ANSI reserved word or ANSI nonreserved word, the function indicates this in the ANSI_restricted column of the result table.

You can also query for Teradata SQL restricted words using the SQLRestrictedWords view, which selects data from the result table of the function. This view only returns restricted words for the current Teradata Database release.

Note: Queries using the SQLRestrictedWords_TBF function and the SQLRestrictedWords view are case specific.

Result

The output table contains the following columns:

Column Name	Data Type and Attributes	Column Value
restricted_word	VARCHAR(30) CHARACTER SET LATIN	The Teradata SQL restricted word.
release_introduced	CHAR(5) CHARACTER SET LATIN	The release when the restricted word was introduced. The release version numbers are in the following format: *MM.mm* where *MM* are 2 digits representing a major release number and *mm* are 2 digits representing a minor release number. For example, '06.02', '12.00', or '13.10'. For all restricted words that were introduced in Teradata Database 6.2 or earlier, the value of the release_introduced field will be '06.02'.
release_dropped	CHAR(5) CHARACTER SET LATIN	The release when the restricted word was dropped. The release version numbers are in the same format as for the release_introduced column. A NULL value indicates that the restricted word is still active.

Column Name	Data Type and Attributes	Column Value
category	CHAR(1) CHARACTER SET LATIN	The valid values are 'R', 'F', or 'N', indicating that the restricted word is: • a Teradata reserved word (R) • a Teradata future reserved word (F) • a Teradata nonreserved keyword (N)
ANSI_restricted	CHAR(1) CHARACTER SET LATIN	The valid values are 'R', 'N', or 'T', indicating that the restricted word is: • an ANSI reserved word (R) • an ANSI nonreserved word (N) • a Teradata restricted word only (T) Note: This column always reflects the ANSI standard that the latest Teradata Database release is compliant with.

SQLRestrictedWords View

You can use the SQLRestrictedWords view to retrieve the Teradata SQL restricted words that are applicable to the current Teradata Database release. The view selects the following columns from the result table of the SQLRestrictedWords_TBF function:

- restricted_word
- category
- ANSI_restricted

To retrieve Teradata SQL restricted words for Teradata Database releases other than the current release, use the SQLRestrictedWords_TBF function.

Downloading the SQLRestrictedWords Package

The DIPUDT script creates the SQLRestrictedWords_TBF function and the SQLRestrictedWords view in the SYSLIB database for the current Teradata Database release.

If you are running a previous Teradata Database release and do not have this function, you can download it at Teradata Developer Exchange (developer.teradata.com/).

The SQLRestrictedWords package includes the following:

- The SQLRestrictedWords_TBF function.
- View definitions for all Teradata Database releases that are currently supported.
- Documentation for installing the function and creating a view to retrieve Teradata restricted words for your Teradata Database release.

For future Teradata Database releases, the package will be updated and available for download in advance of the release so that you can see the new Teradata restricted words of the upcoming release and plan accordingly.

Example 1: Getting the Restricted Words for the Current Release

The following query uses the SQLRestrictedWords view to retrieve the Teradata SQL restricted words that are applicable to the current Teradata Database release.

```
SELECT * FROM SYSLIB.SQLRestrictedWords;
```

Example 2: Getting the Restricted Words for a Specific Release

The following queries use the SQLRestrictedWords_TBF function to retrieve the Teradata SQL restricted words for a specified release.

Retrieve all restricted words for Teradata Database 12.0:

```
SELECT *
FROM TABLE (SYSLIB.SQLRestrictedWords_TBF()) AS t1
WHERE release_introduced <= '12.00' AND
      (release_dropped > '12.00' OR release_dropped IS NULL);
```

Retrieve the restricted words that were introduced in Teradata Database 13.0:

```
SELECT *
FROM TABLE (SYSLIB.SQLRestrictedWords_TBF()) AS t1
WHERE release_introduced = '13.00';
```

Example 3: Dropped and Reintroduced Restricted Words

The restricted word 'INTERFACE' was introduced in Teradata Database 12.0 and dropped in Teradata Database 13.10. Therefore, the output table of the SQLRestrictedWords_TBF function will include a single row for 'INTERFACE':

restricted_word	release_introduced	release_dropped	category	ANSI_restricted
'INTERFACE'	'12.00'	'13.10'	'N'	'N'

If in a later release, for example Teradata Database 16.0, 'INTERFACE' is reintroduced as a Teradata reserved word, a new row will be added for 'INTERFACE' as follows:

restricted_word	release_introduced	release_dropped	category	ANSI_restricted
'INTERFACE'	'12.00'	'13.10'	'N'	'N'
'INTERFACE'	'16.00'	NULL	'R'	'N'

Example 4: Changes Between Reserved and Nonreserved Words

If a Teradata reserved word called 'RESTRICTX' was introduced in Teradata Database 13.10, but the word is changed in Teradata Database 14.0 to be a Teradata nonreserved word, the output table of the SQLRestrictedWords_TBF function will include two rows for 'RESTRICTX':

restricted_word	release_introduced	release_dropped	category	ANSI_restricted
'RESTRICTX'	'13.10'	'14.00'	'R'	'T'
'RESTRICTX'	'14.00'	NULL	'N'	'T'

Example 5: Tracking Future Reserved Words

'WAIT' is a Teradata future reserved word introduced in Teradata Database 06.02, so as of Teradata Database 13.10, the row for 'WAIT' in the output table of the SQLRestrictedWords_TBF function is as follows:

restricted_word	release_introduced	release_dropped	category	ANSI_restricted
'WAIT'	'06.02'	NULL	'F'	'T'

If 'WAIT' becomes a Teradata reserved word in Teradata Database 14.0, the row will be updated as follows when Teradata Database 14.0 is available:

restricted_word	release_introduced	release_dropped	category	ANSI_restricted
'WAIT'	'14.00'	NULL	'R'	'T'

If 'WAIT' is removed as a Teradata future reserved word in Teradata Database 14.0, the row for 'WAIT' will be removed from the table.

ANSI SQL:2008 Reserved Words

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

The following notation appears in the list of reserved words.

Notation	Explanation
TR superscript	This word is also a Teradata reserved word and cannot be used as an identifier.
TN superscript	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.
TF superscript	This word is also a Teradata future reserved word and cannot be used as an identifier.

Note: Words in the 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[TN] 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[TR] 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 notation appears in the list of nonreserved words.

Notation	Explanation
TR superscript	This word is also a Teradata reserved word and cannot be used as an identifier.
TN superscript	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.
TF superscript	This word is also a Teradata future reserved word and cannot be used as an identifier.

Note: Words in the 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 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
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 STACKED STATE
STATEMENT[TR] STATSUSAGE 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 limits 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 *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.	1,073,741,824
Maximum size for a table header.	1 MB
Maximum size of table header cache.	8×10^6 bytes
Maximum size of a response spool file row.	1 MB
Maximum number of change logs that can be specified for a system at any point in time.	1,000,000
Maximum number of locks that can be placed on aggregate online archive logging tables, databases, or both per request. The maximum number of locks that can be placed per LOGGING ONLINE ARCHIVE ON request is fixed at 25,000 and cannot be altered.	25,000
Maximum number of 64KB parse tree segments allocated for parsing requests.	12,000
Maximum number of nodes per system configuration. Limits on vprocs of each type restrict systems with a large number of nodes to fewer vprocs per node. Systems with the maximum vprocs per node cannot approach the maximum number of nodes.	1,024

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.	~ 12 TB/AMP
Maximum number of sectors per data block.	255
Minimum data block size with large cylinders enabled. CREATE TABLE and ALTER TABLE requests allow you to specify a minimum data block size of 21,248 bytes (41.5 512-byte sectors), but Teradata Database rounds that value up internally to 21,504 bytes (42 512-byte sectors), and that is the actual minimum data block size the system uses.	21,504 bytes 42 512-byte sectors
Maximum data block size.	11,894,784 bytes 23,232 512-byte sectors
Minimum data block size with large cylinders not enabled. CREATE TABLE and ALTER TABLE requests allow you to specify a minimum data block size of 8,960 bytes (17.5 512-byte sectors), but Teradata Database rounds that value up internally to 9,216 bytes (18 512-byte sectors), and that is the actual minimum data block size the system uses.	9,216 bytes 18 512-byte sectors
Maximum number of data blocks that can be merged per data block merge.	8
Maximum merge block ratio	100% of the maximum multirow block size for a table.
Data block header size	Depends on several factors. • For a data block that is new or has been updated, the data block header size is 72 bytes. • For a data block that has not been updated, the data block header size is 40 bytes,

Gateway and Vproc Limits

Parameter	Value
Maximum number of sessions per PE.	120
Maximum number of gateways per node.	Multiple. This is true because the gateway runs in its own vproc on each node. See *Utilities* for details. 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: 1 - 2,147,483,647. 1,200 maximum certified. The default is 600. See *Utilities* for details.
Maximum number of vprocs per system.	30,720 This includes the sum of all of the following types of vproc for a configuration. • AMP Access Module Processor vprocs • GTW Gateway Control vprocs • PE Parsing Engine vprocs • RSG Relay Services Gateway vprocs • TVS Teradata Virtual Storage allocator vprocs
Maximum number of AMP vprocs per system.	16,200

Parameter	Value
Maximum number of GTW vprocs per system. This is a soft limit that Teradata technical support personnel can reconfigure for you. Each GTW vproc on a node must be in a different host group. If two GTW vprocs are in the same host group, they must be on different nodes.	The default is one GTW vproc per node with all of them assigned to the same host group. The maximum depends on two factors. • Number of IP addresses assigned to each node. • Number of host groups configured in the database. Each gateway on a node must be assigned to a different host group from any other gateway *on the same node* and each gateway needs to be assigned a disjoint set of IP addresses to service. This does *not* mean that all gateways *in a system* must be assigned to a different host group. It means that each gateway *on the same node* must be assigned to a different host group. Gateways on *different* nodes can be assigned to the same host group.
Maximum number of PE vprocs per system. This is a soft limit that Teradata technical support personnel can reconfigure for you.	2,048
Maximum number of RSG vprocs per system. This is a soft limit that Teradata technical support personnel can reconfigure for you.	1,024 See *Teradata Replication Services Using Oracle GoldenGate*.
Maximum number of TVS allocator vprocs per system. This is a soft limit that Teradata technical support personnel can reconfigure for you.	2,048
Maximum number of vprocs, in any combination, per node.	127
Maximum number of AMP vprocs per cluster.	8
Maximum number of external routine protected mode platform tasks per PE or AMP. 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. The valid range is 0 to 20, inclusive. The limit is 20 platform tasks for *each* platform type, not 20 combined for both. See *Utilities* for details.	20
Maximum number of external routine secure mode platform tasks per PE or AMP. 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.	20

Parameter	Value
Size of a request control block	~ 40 bytes
Default number of lock segments per AMP vproc. This is controlled by the NumLokSegs parameter in DBS Control.	2
Maximum number of lock segments per AMP vproc. This is controlled by the NumLokSegs parameter in DBS Control.	8
Default size of a lock segment. This is controlled by the LockLogSegmentSize parameter in DBS Control.	64 KB
Maximum size of a lock segment. This is controlled by the LockLogSegmentSize parameter in DBS Control.	1 MB
Default number of locks per AMP	3,200
Maximum number of locks per AMP.	209.000
Maximum size of the lock table per AMP. The AMP lock table size is fixed at 2 MB and cannot be altered.	2 MB
Maximum size of the queue table FIFO runtime cache per PE.	• 100 queue table entries • 1 MB
Maximum number of SELECT AND CONSUME requests that can be 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
Amount of private disk swap space required per protected or secure mode server for Java external routines per node.	30 MB

Hash Bucket Limits

Parameter	Value
Number of hash buckets per system. This value is user-selectable. See the topic “Hash Maps” in Chapter 8 of *Database Design* for details.	The number is user-selectable per system. The choices are the following. • 65,536 • 1,048,576 Bucket numbers range from 0 to the system maximum.

Parameter	Value
Size of a hash bucket	The size depends on the number of hash buckets on the system. • If the system has 65,536 hash buckets, the size of a hash bucket is 16 bits. • If the system has 1,048,576 hash buckets, the size of a hash bucket is 20 bits. You set the default hash bucket size for your system using the DBS Control utility (see *Utilities: Volume 1 (A-K)* for details).

Interval Histogram Limits

Parameter	Value
Number of hash values.	4.2×10^9
Maximum number of intervals per index or column set histogram. The system-wide maximum number of interval histograms is set using the MaxStatsInterval parameter of the DBS Control record or your cost profile. For descriptions of the other parameters listed, see *SQL Request and Transaction Processing*.	500
Default number of intervals per index or column set histogram.	250
Maximum number of equal-height intervals per interval histogram.	500

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. This limit includes 2,048 data table columns plus 13 error table columns.	2,061
Maximum number of UDT columns per base data table. The same limit is true for both distinct and structured UDTs. 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.	~1,600

Parameter	Value
Maximum number of LOB columns per base data table. This limit includes 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.	32
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 table header that is large enough to require more than ~64,000 bytes uses multiple 64KB rows per AMP up to 1 MB. A table header that requires 64,000 or fewer bytes uses only a single row and does not use the additional rows that are required to contain a larger table header. The *maximum* size for a table header is 1 MB.	1 MB
Maximum number of characters per SQL index constraint.	16,000
Maximum row size.	64,256 bytes
Maximum size of the queue table FIFO runtime cache per table.	2,211 row entries
Maximum logical row size. 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. 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.	67,106,816,000 bytes
Maximum non-LOB column size for an NPPI table. This limit is based on subtracting the minimum row overhead value for an NPPI table row (12 bytes) from the system-defined maximum row length (64,256 bytes).	64,244 bytes
Maximum non-LOB column size for a PPI table. This limit is based on subtracting the minimum row overhead value for a PPI table row (16 bytes) from the system-defined maximum row length (64,256 bytes).	64,240 bytes
Maximum database, user, base table, view, macro, index, trigger, stored procedure, UDF, UDM, UDT, replication group name, constraint, or column name size. Other rules apply for Japanese character sets, which might restrict names to fewer than 30 bytes. See Chapter 2 in *SQL Fundamentals* for the applicable rules.	30 bytes in LATIN or KANJI1 internal representation
Maximum number of values that can be multi-value compressed per base table column.	255
Maximum number of bytes per column that can be multi-value compressed for byte, Kanji1, and KanjiSJIS data	~7,800 bytes
Maximum number of characters per column that can be multi-value compressed for GRAPHIC, LATIN, and UNICODE server character set data	~7,800 characters ~15,600 bytes

Parameter	Value
Maximum number of columns that can be compressed per primary-indexed table using multi-value compression or algorithmic compression. This assumes that the object is not a non-partitioned NoPI table, a column-partitioned table or join index, or a global temporary trace table. All other tables, hash indexes, and join indexes must have a primary index, and primary indexes cannot be either multi-value compressed or algorithmically compressed. Because of this, the limit is the maximum number of columns that can be defined for a table, which is 2,048, minus 1. The limit for multi-value compression is far more likely to be reached because of table header overflow, but the amount of table header space that is available for multi-value compressed values is limited by a number of different factors. See *Database Design* for details. Join index columns inherit their compression characteristics from their parent tables.	2,047
Maximum number of columns that can be compressed per non-partitioned NoPI table or column-partitioned table using multi-value compression or algorithmic compression Column-partitioned join index columns inherit their compression characteristics from their parent tables.	2,048
Maximum number of algorithmically compressed values per base table column.	Unlimited
Maximum width of data that can be multi-value compressed for BYTE, BYTEINT, CHARACTER, GRAPHIC, VARCHAR, and VARGRAPHIC data types	510 bytes
Maximum width of data that can be multi-value compressed for data types other than BYTE, BYTEINT, CHARACTER, DATE, GRAPHIC, VARCHAR, and VARGRAPHIC	Unlimited
Maximum width of data that can be algorithmically compressed. The maximum data width is unlimited if you specify only algorithmic compression for a column. If you specify a mix of multi-value and algorithmic compression for a column, then the limits for multi-value compression also apply for algorithmic compression.	Unlimited
Maximum number of table-level CHECK constraints that can be defined per table.	100
Maximum number of primary indexes per table, hash index, or join index that is not a NoPI or column-partitioned database object.	1
Minimum number of primary indexes per primary-indexed table, hash index, or join index.	1
Maximum number of primary indexes per non-partitioned NoPI table, column-partitioned table or join index, or global temporary trace table. This maximum applies *only* to non-partitioned NoPI tables, column-partitioned tables and join indexes, and global temporary trace tables. All other tables, hash indexes, and join indexes *must* have a primary index.	0
Maximum number of columns per primary index.	64
Maximum number of column partitions per table, including two columns partitions reserved for internal use).	2,050

Parameter	Value
Minimum number of column partition numbers that must be available for use by an ALTER TABLE request to alter a column partition.	1
Maximum partition number for a column partitioning level.	Maximum number of partitions for that level + 1
Maximum combined partition number for a single-level column-partitioned table or column-partitioned join index.	The same as the maximum partition number for the single partitioning level.
Maximum number of rows per hash bucket for a 44-bit uniqueness value.	17,592,186,044,415
Maximum combined partition number for a multilevel partitioned primary index, multilevel column-partitioned table, multilevel partitioned join index, or multilevel column-partitioned join index for 2-byte partitioning.	65,535
Maximum combined partition number for a multilevel partitioned primary index, multilevel column-partitioned table, multilevel partitioned join index, or multilevel column-partitioned join index for 8-byte partitioning.	9,223,372,036,854,775,807
Maximum number of ranges, including the NO RANGE, UNKNOWN, and NO RANGE OR UNKNOWN and UNKNOWN partitions, for a RANGE_N partitioning expression for 2-byte partitioning. This value is limited by the largest possible INTEGER value.	65,533
Maximum number of ranges, including the NO RANGE, UNKNOWN, and NO RANGE OR UNKNOWN and UNKNOWN partitions, for a RANGE_N partitioning expression for 8-byte partitioning. This value is limited by the largest possible BIGINT value.	9,223,372,036,854,775,805
Minimum value for *n* in a RANGE#L*n* expression.	1
Maximum value for *n* in a RANGE#L*n* expression for 2-byte partitioning	15
Maximum value for *n* in a RANGE#L*n* expression for 8-byte partitioning.	62
Maximum number of partitions, including the NO RANGE, UNKNOWN, and NO RANGE OR UNKNOWN partitions, for a single-level partitioning expression composed of a single RANGE_N function with INTEGER data type	2.147.483.647
Maximum number of ranges for a single-level partitioning expression composed of a single RANGE_N function with INTEGER data type that is used as a partitioning expression if the NO RANGE and UNKNOWN partitions are not specified.	65,533
Maximum number of ranges for a single-level partitioning expression composed of a single RANGE_N function with BIGINT data type that is used as a partitioning expression if the NO RANGE and UNKNOWN partitions are not specified.	9,223,372,036,854,775,805
Maximum number of ranges for a single-level partitioning expression composed of a single RANGE_N function with BIGINT data type that is used as a partitioning expression if both the NO RANGE and UNKNOWN partitions are specified.	9,223,372,036,854,775,807
Maximum value for a partitioning expression that is not based on a RANGE_N or CASE_N function. This is allowed only for single-level partitioning.	65,535

Parameter	Value
Maximum number of defined partitions for a column partitioning level	The number of column partitions specified + 2 The 2 additional partitions are reserved for internal use.
Maximum number of defined partitions for a row partitioning level if the row partitions specify the RANGE_N or CASE_N function.	The number of row partitions specified.
Maximum number of defined partitions for a row partitioning level if the row partitions do not specify the RANGE_N or CASE_N function.	65,535
Maximum number of partitions for a partitioning level when you specify an ADD clause. This value is computed by adding the number of defined partitions for the level plus the value of the integer constant specified in the ADD clause.	9,223,372,036,854,775,807
Maximum number of partitions for a column partitioning level when you do not specify an ADD clause and at least one row partitioning level does not specify an ADD clause	The number of column partitions defined + 10.
Maximum number of column partitions for a column partitioning level when you do not specify an ADD clause, you also specify row partitioning, and each of the row partitions specifies an ADD clause.	The largest number for the column partitioning level that does not cause the partitioning to be 8-byte partitioning.
Maximum number of partitions for each row partitioning level without an ADD clause in level order, if using the number of row partitions defined as the maximum for this and any lower row partitioning level without an ADD clause.	The largest number for the column partitioning level that does not cause the partitioning to be 8-byte partitioning.
Maximum partition number for a row partitioning level.	The same as the maximum number of partitions for the level.
Minimum number of partitions for a row partitioning level.	2
Maximum number of partitions, including the 2 partitions reserved for internal use, for a column partitioning level when you do not specify an ADD clause and there is only one partitioning level.	65,534
Maximum number of partitions for a CASE_N partitioning expression. This value is limited by the largest possible INTEGER value.	2,147,483,647
Maximum value for a RANGE_N function with an INTEGER data type.	2,147,483,647
Maximum value for a RANGE_N function with a BIGINT data type that is part of a partitioning expression.	9,223,372,036,854,775,805
Maximum value for a CASE_N function for both 2-byte and 8-byte partitioning.	2,147,483,647
Maximum number of partitioning levels or partitioning expressions for a multilevel partitioned primary index or column-partitioned table or join index for 2-byte partitioning. Other limits can further restrict the number of levels for a specific partitioning.	15

Parameter	Value
Maximum number of partitioning levels or partitioning expressions for a multilevel partitioned primary index or column-partitioned table or join index for 8-byte partitioning. Other limits can further restrict the number of levels for a specific partitioning.	62
Maximum value for n for the system-derived column PARTITION#L*n* for 2-byte partitioning.	15
Maximum value for *n* for the system-derived column PARTITION#L*n* for 8-byte partitioning.	62
Minimum number of partitions per partitioning level for a multilevel partitioned primary index.	2
Maximum number of partition number ranges from each level that are not eliminated for static partition elimination for an 8-byte PPI table or join index.	8,000
Maximum number of table-level constraints per base data table.	100
Maximum size of the SQL text for a table-level index CHECK constraint definition.	16,000 characters
Maximum number of referential integrity constraints per base data table.	64
Maximum number of columns per foreign and parent keys in a referential integrity relationship.	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 in the configuration)
Maximum number of row-level security constraints per table, user, or profile.	5
Maximum number of row-level security statement-action UDFs that can be defined per table.	4
Maximum number of non-set row-level security constraint encodings that can be defined per constraint. The valid range is from 1 to 10,000. 0 is not a valid non-set constraint encoding.	10,000
Maximum number of set row-level security constraint encodings that can be defined per constraint.	256

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

Parameter	Value
Maximum number of LOB rows per rowkey per AMP for NoPI LOB tables	~ 256M The exact number is 268,435,455 LOB rows per rowkey per AMP.
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, ARRAY Data Type, and VARRAY Data Type Limits

Parameter	Value
Maximum structured UDT size. This value is 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 64,256 bytes.	• 64,242 bytes (NoPI or column-partitioned table) • 64,241 bytes (NPPI table) • 64,239 bytes (PPI table)
Maximum number of UDT columns per base data table. 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. The figure of 1,600 UDT columns assumes a FAT table header. This limit is true whether the UDT is a distinct or a structured type.	~1,600
Maximum database, user, base table, view, macro, index, trigger, stored procedure, UDF, UDM, UDT, replication group name, constraint, or column name size. Other rules apply for Japanese character sets, which might restrict names to fewer than 30 bytes. See *SQL Fundamentals* for the applicable rules.	30 bytes in Latin or Kanji1 internal representation
Maximum number of attributes that can be specified for a structured UDT per CREATE TYPE or ALTER TYPE request. The maximum is platform-dependent, not absolute.	300 - 512
Maximum number of attributes that can be defined for a structured UDT. While you can specify no more than 300 to 512 attributes for a structured UDT per CREATE TYPE or ALTER TYPE request, you can submit any number of ALTER TYPE requests with the ADD ATTRIBUTE option specified as necessary to add additional attributes to the type up to the upper limit of approximately 4,000.	~4,000
Maximum number of levels of nesting of attributes that can be specified for a structured UDT.	512

Parameter	Value
Maximum number of methods associated with a UDT. 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 "ALTER TYPE" in SQL Data Definition Language Detailed Topics for details.	~500
Maximum number of input parameters with a UDT data type of VARIANT_TYPE that can be declared for a UDF definition.	8
Minimum number of dimensions that can be specified for a multidimensional ARRAY or VARRAY data type.	2
Maximum number of dimensions that can be specified for a multidimensional ARRAY or VARRAY data type.	5

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 MB
Maximum number of open cursors per stored procedure.	15
Maximum number of result sets a stored procedure can return	15
Maximum number of columns returned by a dynamic result table function. The valid range is from 1 to 2,048. There is no default.	2,048
Maximum number of dynamic SQL requests per stored procedure	15
Maximum length of a dynamic SQL request in a stored procedure This includes its SQL text, the USING data (if any), and the CLIv2 parcel overhead.	64,256 bytes
Maximum number of parameters specified in a UDF defined without dynamic UDT parameters.	128
Maximum number of parameters that can be defined for a constructor method for all types except ARRAY/VARRAY	128
Maximum number of parameters that can be defined for a constructor method of an ARRAY/VARRAY type	*n* where *n* is the number of elements defined for the type
Maximum number of combined return values and local variables that can be declared in a single UDF.	Unlimited
Maximum number of combined external routine return values and local variables that can be instantiated at the same time per session.	1,000

Parameter	Value
Maximum combined size of the parameters defined for a UDF.	64KB
Maximum number of parameters specified in a UDF defined with dynamic UDT parameters. The valid range is from 0 to 15. The default is 0.	1,144
Maximum number of parameters specified in a method.	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 size of an ARRAY or VARRAY UDT. This limit does not include the number of bytes used by the row header and the primary index of a table.	64 KB
Maximum length of external name string for an external routine. 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.	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 KB
Maximum number of Statement Areas per SQL stored procedure diagnostics area. See *SQL Stored Procedures and Embedded SQL* and *SQL External Routine Programming* for details.	1
Maximum number of Condition Areas per SQL stored procedure diagnostics area. See *SQL Stored Procedures and Embedded SQL* and *SQL External Routine Programming* for details.	16

Query and Workload Analysis Limits

Parameter	Value
Maximum size of the Index Wizard workload cache. The default is 48 Mbytes and the minimum is 32 MB.	187 MB
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

Parameter	Value
Maximum number of pseudoindexes on which multicolumn statistics can be collected and maintained at one time. 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. 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 or join index if single-column PARTITION statistics are *not* collected on the table or index.	32
Maximum number of sets of multicolumn statistics that can be collected on a table or join index if single-column PARTITION statistics *are* collected on the table or index.	31
Maximum size of SQL query text overflow stored in QCD table *QryRelX* that can be read by the Teradata Index Wizard.	1 MB

Secondary, Hash, and Join Index Limits

Parameter	Value
Minimum number of secondary, hash, and join indexes, in any combination, per base data table. The only index required for any Teradata Database table or index is a primary index. A primary index is *not* required or allowed for the following. • Global temporary trace tables • Non-partitioned NoPI tables • Column-partitioned tables and join indexes • Secondary indexes	0
Maximum number of secondary, hash, and join indexes, in any combination, per base data table. Each composite NUSI defined with an ORDER BY clause counts as 2 consecutive indexes in this calculation. The number of system-defined secondary and single-table join indexes contributed by PRIMARY KEY and UNIQUE constraints counts against the combined limit of 32 secondary, hash, and join indexes per base data table.	32
Maximum number of columns referenced per secondary index.	64
Maximum number of columns referenced per single table in a hash or join index.	64
Maximum number of rows per secondary, hash, or join index.	Limited only by disk capacity.

Parameter	Value
Maximum number of columns referenced in the fixed part of a compressed join index. Teradata Database implements 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 the compression of column values, the term refers one of the following. • Multi-value compression, in which Teradata Database stores the compressed values for a column one time only in the table header, not in the row itself. • Algorithmic compression, in which you specify scalar UDFs to compress and decompress specified byte, character, or graphic data values. The compression and decompression algorithms used by algorithmic compression are determined by you and follow the rules that are defined for a given pair of compression and decompression algorithms.	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

Reference Index Limits

Parameter	Value
Maximum number of reference indexes per base data table. There is a maximum of 128 Reference Indexes in a table header, 64 from a parent table to child tables and 64 from child tables to a parent table.	64

SQL Request and Response Limits

Parameter	Value
Maximum SQL text size per request. This includes SQL request text, USING data, and parcel overhead.	1 MB
Maximum number of entries in an IN list. 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.	Unlimited
Maximum SQL text title size.	60 characters
Maximum SQL activity count size. This value is derived from 2^{32} - 1.	4,294,967,295 rows
Maximum number of tables and single-table views that can be joined per query block. This limit is controlled by the MaxJoinTables DBS Control parameter and the Cost Profile flags.	128

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. This limit is controlled by the MaxJoinTables DBS Control parameter and the Cost Profile flags.	128
Maximum number of fields in a USING row descriptor.	2,543
Maximum number of open cursors per embedded SQL program	16
Maximum SQL text response size.	1 MB
Maximum number of columns per DML request ORDER BY clause.	64
Maximum number of columns per DML request GROUP BY clause.	64
Maximum number of ORed conditions or IN list values per request	1,048,576

Row-Level Security Constraint Limits

Parameter	Value
Maximum number of row-level security constraints per table.	5
Maximum number of hierarchical row-level security constraints per user or profile.	6
Maximum number of values per hierarchical row-level security constraint.	10,000
Maximum number of non-hierarchical row-level security constraints per user or profile.	2
Maximum number of values per non-hierarchical row-level security constraint.	256

Session Limits

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

Parameter	Value
Maximum number of sessions per PE.	120
Maximum number of sessions per gateway vproc. See *Utilities* for details.	Tunable: 1 - 2,147,483,647. 1,200 maximum certified. The default is 600.
Maximum number of active request result spool files per session.	16

Parameter	Value
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).	20
Maximum number of channels required for various parallel step operations. The value per request is determined as follows:	
• Primary index request with an equality constraint.	0 channels
• Requests that do not involve row distribution.	2 channels
• Requests that involves redistribution of rows to other AMPs, such as a join or an INSERT … SELECT operation.	4 channels
Maximum number of materialized global temporary tables that can be materialized simultaneously per session.	2,000
Maximum number of volatile tables that can be instantiated simultaneously per session.	1,000
Maximum number of SQL stored procedure Diagnostic Areas that can be active per session.	1

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

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.

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

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.

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 commonly used among 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.

WHEN ...	THEN ...
the difference between the Teradata SQL dialect and the ANSI SQL standard for a language feature is significant	both syntaxes are offered and the user has the choice of operating in either Teradata or ANSI mode or of turning off SQL Flagger. The mode can be defined 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.

a new feature or feature enhancement is added to Teradata SQL and that feature is not defined by the ANSI SQL standard	that feature is designed using the following criteria:
	• If other vendors offer a similar feature or feature extension, Teradata designs the new feature to broadly comply with other solutions, but consolidates the best ideas from all and, where necessary, creates its own, cleaner solution.
	• If other vendors do not offer a similar feature or feature extension, Teradata designs the new feature:
	  • 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.

Third Party Books on the ANSI SQL Standard

There are few third party books on the ANSI SQL standard, and none about the SQL:2003 or SQL:2008 standards. The following books on earlier ANSI SQL standards were all either written or co-written by Jim Melton, 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.

  Provides coverage of 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 covers 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)
Module	Not defined

ANSI	Teradata
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

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 Mainframe-Attached Systems* and *Teradata Call-Level Interface Version 2 Reference for Network-Attached Systems* for details on setting the conformance field.	

APPENDIX E

Performance Considerations

This Appendix provides suggestions for improving database query performance.

Using the 2 PC Protocol

Two-Phase Commit (2PC) is an IMS and CICS protocol for committing update transactions processed by multiple systems that do not share the same locking and recovery mechanism.

For detailed information on 2PC, see *Teradata Director Program Reference*.

Performance Impact

Consider the following disadvantages of using the 2PC protocol:

- Performance may decrease because, at the point of synchronization, up to two additional messages are exchanged between the coordinator and participant, in addition to the normal messages that update the database

  If your original Teradata Database SQL request took longer to complete than your other requests, the performance impact due to the 2PC overhead will be less noticeable.
- • If Teradata Database restarts, and a session using the 2PC protocol ends up in an IN-DOUBT state, Teradata Database holds data locks indefinitely until you resolve the IN-DOUBT session. During this time, other work could be blocked if it accesses the same data for which Teradata Database holds those locks.

  To resolve this situation, perform the following steps:
  - Use the COMMIT/ROLLBACK command to resolve manually the IN-DOUBT sessions.
  - Use the RELEASE LOCKS command.
  - Use the RESTART command to restart your system.

2PC causes no system overhead when it is disabled.

Glossary

2PC Two-Phase Commit

AMP Access Module Processor vproc

ANSI American National Standards Institute

BLOB Binary Large Object

BTEQ Basic Teradata Query facility

BYNET Banyan Network - High speed interconnect

CJK Chinese, Japanese, and Korean

CLIv2 Call Level Interface Version 2

CLOB Character Large Object

cs0, cs1, cs2, cs3 Four code sets (codeset 0, 1, 2, and 3) used in EUC encoding.

distinct type A UDT that is based on a single predefined data type

E2I External-to-Internal

EUC Extended UNIX Code

external routine UDF, UDM, or external stored procedure that is written using C, C++, or Java

external stored procedure a stored procedure that is written using C, C++, or Java

FK Foreign Key

HI Hash Index

I2E Internal-to-External

JI Join Index

JIS Japanese Industrial Standards

LOB Large Object

LT/ST Large Table/Small Table (join)

NoPI tables Tables that are defined with no primary index (PI)

NPPI Nonpartitioned Primary Index

NUPI Nonunique Primary Index

NUSI Nonunique Secondary Index

OLAP Online Analytical Processing

OLTP Online Transaction Processing

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

R

S

T

U

V

Z