Teradata
a division of NCR

Teradata® RDBMS

SQL Reference - Volume 6
Data Manipulation Statements

V2R5.0

B035-1101-122A
May 2003

The product described in this book is a licensed product of NCR Corporation.

BYNET is an NCR trademark registered in the U.S. Patent and Trademark Office.
CICS, CICS/400, CICS/600, CICS/ESA, CICS/MVS, CICSPLEX, CICSVIEW, CICS/VSE, DB2, DFSMS/MVS, DFSMS/VM, IBM, NQS/MVS, OPERATING SYSTEM/2, OS/2, PS/2, MVS, QMS, RACF, SQL/400, VM/ESA, and VTAM are trademarks or registered trademarks of International Business Machines Corporation in the U. S. and other countries.
DEC, DECNET, MICROVAX, VAX and VMS are registered trademarks of Digital Equipment Corporation.
HEWLETT-PACKARD, HP, HP BRIO, HP BRIO PC, and HP-UX are registered trademarks of Hewlett-Packard Co.
KBMS is a trademark of Trinzic Corporation.
INTERTEST is a registered trademark of Computer Associates International, Inc.
MICROSOFT, MS-DOS, MSN, The Microsoft Network, MULTIPLAN, SQLWINDOWS, WIN32, WINDOWS, WINDOWS 2000, and WINDOWS NT are trademarks or registered trademarks of Microsoft Corporation.
SAS, SAS/C, SAS/CALC, SAS/CONNECT, and SAS/CPE are registered trademarks of SAS Institute Inc.
SOLARIS, SPARC, SUN and SUN OS are trademarks of Sun Microsystems, Inc.
TCP/IP protocol is a United States Department of Defense Standard ARPANET protocol.
TERADATA and DBC/1012 are registered trademarks of NCR International, Inc.
UNICODE is a trademark of Unicode, Inc.
UNIX is a registered trademark of The Open Group.
X and X/OPEN are registered trademarks of X/Open Company Limited.
YNET is a trademark of NCR Corporation.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS-IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. IN NO EVENT WILL NCR CORPORATION (NCR) BE LIABLE FOR ANY INDIRECT, DIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS OR LOST SAVINGS, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The information contained in this document may contain references or cross references to features, functions, products, or services that are not announced or available in your country. Such references do not imply that NCR intends to announce such features, functions, products, or services in your country. Please consult your local NCR representative for those features, functions, products, or services available in your country.

Information contained in this document may contain technical inaccuracies or typographical errors. Information may be changed or updated without notice. NCR may also make improvements or changes in the products or services described in this information at any time without notice.

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

or write:

Information Engineering
NCR Corporation
100 North Sepulveda Boulevard
El Segundo, CA 90245-4361
U.S.A.

Any comments or materials (collectively referred to as "Feedback") sent to NCR will be deemed non-confidential. NCR will have no obligation of any kind with respect to Feedback and will be free to use, reproduce, disclose, exhibit, display, transform, create derivative works of and distribute the Feedback and derivative works thereof without limitation on a royalty-free basis. Further, NCR will be free to use any ideas, concepts, know-how or techniques contained in such Feedback for any purpose whatsoever, including developing, manufacturing, or marketing products or services incorporating Feedback.

Copyright © 1996-2002, NCR Corporation
All Rights Reserved

Preface

Supported Software Release

This book supports Teradata RDBMS Release V2R5.0.

Changes to This Book

This book includes the following changes to support the current release:

Date	Description
May 2003	• Removed all references to performing DML on ROWIDs, noting that such operations are not valid.
December 2002	• Enhanced SAMPLE clause for stratified samples. • Added new MERGE statement. • Moved miscellaneous embedded SQL-only declarations into chapter 4. • Moved cursor control statements into chapter 7. • Moved dynamic SQL statements into chapter 4. • Moved old chapter 4 of *Teradata RDBMS SQL Reference, Volume 4* into chapter 5. • Moved result code variables from chapter 1 of *Teradata RDBMS SQL Reference, Volume 1* into chapter 8. • Moved host variables from chapter 2 of *Teradata RDBMS SQL Reference, Volume 1* into chapter 4. • Moved multisession asynchronous programming material from chapter 2 of *Teradata RDBMS SQL Reference, Volume 1* into chapter 6. • Moved stored procedures chapter from 4 to 9. • Updated stored procedures. • General reorganization and rewriting of material to enhance usability. • Performed DR fixes and enhancement requests.
June 2001	• Added new upsert form of UPDATE statement • Added new SQL control statements for stored procedures • Performed DR fixes and enhancement requests
September 2000	• Usability enhancements

Date	Description
June 2000	• Addition and extensive rewriting and reorganization of embedded SQL material formerly found in *Teradata Application Programming with C, COBOL, and PL/I*. • New SQL DML and stored procedure control language statements. • General reorganization and rewriting of material to enhance usability. • Performed DR fixes and enhancement requests.

About This Book

Purpose

Teradata RDBMS SQL Reference, Volume 6 describes how to use SQL to manipulate data.

This preface describes the organization of *Teradata RDBMS SQL Reference, Volume 6* and identifies information you should know before using it. This book should be used in conjunction with the other volumes of *Teradata RDBMS SQL Reference*.

Audience

Application programmers and end users are the principal audience for this manual. System administrators, database administrators, security administrators, NCR field engineers, and other technical personnel responsible for designing, maintaining, and using the Teradata RDBMS might also find this manual to be useful.

How This Book Is Organized

This book has four chapters and four appendixes.

- Chapter 1: "The SELECT Statement," describes the SELECT statement, its associated clauses (FROM, WHERE, GROUP BY, HAVING, QUALIFY, SAMPLE, ORDER BY, WITH), the SAMPLEID expression, and the query expression set operators UNION, INTERSECT, and EXCEPT.
- Chapter 2: "Join Expressions," describes how to use joins to enrich the power and subtlety of queries. Particular emphasis is given to the outer join, including case studies.
- Chapter 3: "SQL Data Manipulation Language Statement Syntax," describes the SQL DML statements except for SELECT, SQL preprocessor declarative statements, cursor control statements, and dynamic SQL statements.
- Chapter 4: "Embedded SQL," describes embedded SQL and the following embedded SQL-related topics: host and indicator variables, multistatement requests, embedded SQL declaratives and related statements, and dynamic SQL and its associated SQL statements.
- Chapter 5: "Client-Server Connectivity Statements," describes the client-server connectivity statements used by embedded SQL applications to establish and maintain connections between client applications and the Teradata server.
- Chapter 6: "Multisession Asynchronous Programming With Embedded SQL," describes multisession asynchronous programming with embedded SQL.

- Chapter 7: "Cursors and Cursor Control Statements," describes cursors and cursor control statements for embedded SQL and stored procedures.
- Chapter 8: "Result Code Variables," describes the embedded SQL and stored procedure results code variables SQLSTATE, SQLCODE, and ACTIVITY_COUNT.
- Chapter 9: "SQL Stored Procedures," describes the SQL control statements and how to use them to create stored procedures and provides guidelines on debugging stored procedure-based applications. The chapter also includes a sample stored procedure.
- Appendix A: "Notation Conventions," explains how to read syntax diagrams, character shorthand notation, and predicate calculus notation.
- Appendix B: "SQL Descriptor Area (SQLDA)," describes the SQLDA data structure used by embedded SQL applications.
- Appendix C: "SQL Communications Area (SQLCA)," describes the SQLCA data structure used by older embedded SQL applications.
- Appendix D: "SQLSTATE Mappings," describes the special SQLSTATE codes used by stored procedure language applications.

Prerequisites

If you are not familiar with the Teradata RDBMS, you should read *Introduction to the Teradata RDBMS* before reading this document.

Additional information about developing applications using embedded SQL is found in *Teradata Preprocessor2 Programmer Guide*.

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

List of Acronyms

The following acronyms, listed in alphabetical order, are applicable to this release:

AMP	Access Module Processor
ANSI	American National Standards Institute
BTEQ	Basic TEradata Query facility
BYNET	Banyan Network - High speed interconnect
CJK	Chinese, Japanese, and Korean
CLIv2	Call Level Interface Version 2
cs0, cs1, cs2, cs3	Four code sets (codeset 0, 1, 2, and 3) used in EUC encoding.
DB2	Database 2
E2I	External-to-Internal
EUC	Extended UNIX Code
FK	Foreign Key
HI	Hash Index
I2E	Internal-to-External
IBM	International Business Machines
JI	Join Index
JIS	Japanese Industrial Standards
LT/ST	Large Table/Small Table (join)
NUPI	Non-Unique Primary Index
NUSI	Non-Unique Secondary Index
OLTP	On-Line Transaction Processing
PDE	Parallel Database Extensions
PE	Parsing Engine
PK	Primary Key
RDBMS	Relational Database Management System
SQL	Structured Query Language
UCS-2	Universal Coded Character Set containing 2 bytes
UPI	Unique Primary Index

USI	Unique Secondary Index
vproc	Virtual Process

Technical Information on the Web

The NCR home page (http://www.ncr.com) provides links to numerous sources of information about Teradata. Among the links provided are sites that deal with the following subjects:

- Contacting technical support
- Enrolling in customer education courses
- Ordering and downloading product documentation
- Accessing case studies of customer experiences with Teradata
- Accessing third party industry analyses of Teradata data warehousing products
- Accessing white papers
- Viewing or subscribing to various online periodicals

Contents

Preface

About This Book iii
List of Acronyms v
Technical Information on the Web vii

Section 1:
General SQL Data Manipulation Language

Chapter 1:
The SELECT Statement
SELECT 1–2
Simple SELECT Statement 1–13
SELECT … INTO 1–15
DISTINCT and ALL Options 1–20
FROM Clause 1–22
Derived Tables 1–29
WHERE Clause 1–32
Subqueries in Search Conditions 1–37
Correlated Subqueries 1–41
Correlated Subqueries and the SELECT Statement 1–49
Correlated Subqueries and the UPDATE Statement 1–52
Correlated Subqueries and the DELETE Statement 1–54
Correlated Subqueries and the ABORT/ROLLBACK Statements 1–56
GROUP BY Clause 1–57
HAVING Clause 1–61
QUALIFY Clause 1–64
SAMPLE Clause 1–67
SAMPLEID Expression 1–75
ORDER BY Clause 1–77
WITH Clause 1–85

Chapter 2:
Join Expressions

Joins 2–2
Inner Joins 2–4
Ordinary Inner Join 2–5
Cross Join 2–8
Self-Join 2–10
Outer Joins 2–11
Definition of the Outer Join 2–12
Review of Relevant Relational Algebra for the Outer Join 2–18
Left Outer Join 2–20
Right Outer Join 2–22
Full Outer Join 2–24
Multitable Joins 2–26
Coding ON Clauses for Outer Joins 2–30
Coding ON Clauses With WHERE Clauses for Outer Joins 2–33
Rules And Recommendations for Coding ON and WHERE Clauses for Outer Joins 2–37
Outer Join Case Study 2–38
Case Study Examples 2–39
Heuristics for Determining a Reasonable Answer Set 2–41
First Attempt 2–43
Second Attempt 2–46
Third Attempt 2–48
Final Attempt: The Correct Answer 2–51
Guidelines for Getting the Desired Answer Using Outer Joins 2–53

Chapter 3:
SQL Data Manipulation Language Statement Syntax

ABORT 3–3
BEGIN TRANSACTION 3–7
CALL 3–13
CHECKPOINT 3–28
COMMENT (Placing Form) 3–31
COMMIT 3–33
DELETE 3–36
DELETE (Basic/Searched Form) 3–42
DELETE (Join Condition Form) 3–44
DELETE (Positioned Form) 3–46
ECHO 3–48

END TRANSACTION 3–50
EXECUTE (Macro Form) 3–52
EXPLAIN Modifier 3–55
EXPLAIN Modifier Terminology 3–59
EXPLAIN Modifier in Greater Depth 3–64
EXPLAIN: Examples of Complex Queries 3–65
EXPLAIN and Join Processing 3–71
EXPLAIN and Standard Indexed Access 3–77
EXPLAIN and Parallel Steps 3–80
EXPLAIN and Partitioned Primary Index Access 3–82
EXPLAIN and MERGE Conditional Steps 3–90
EXPLAIN and UPDATE (Upsert Form) Conditional Steps 3–97
GROUP BY Clause 3–104
HAVING Clause 3–105
INSERT 3–106
LOCKING Modifier 3–119
MERGE 3–132
ORDER BY Clause 3–139
QUALIFY Clause 3–140
ROLLBACK 3–141
SAMPLE Clause 3–145
SELECT 3–146
SELECT … INTO 3–147
UPDATE (Searched Form) 3–148
UPDATE (Positioned Form) 3–157
UPDATE (Upsert Form) 3–160
USING Row Descriptor 3–168
WHERE Clause 3–180
WITH Clause 3–181

Section 2:
Embedded SQL

Chapter 4:
Embedded SQL
Embedded SQL 4–2
Host Structures 4–5
Host Variables 4–7
Input Host Variables 4–12
Output Host Variables 4–15
SQL Character Strings as Host Variables 4–19

Indicator Variables 4–20
Multistatement Requests With Embedded SQL 4–24
Miscellaneous Embedded SQL-Only Statement Syntax 4–28
BEGIN DECLARE SECTION 4–29
COMMENT (Returning Form) 4–30
DATABASE 4–32
DECLARE STATEMENT 4–34
DECLARE TABLE 4–35
END DECLARE SECTION 4–37
END-EXEC Statement Terminator 4–38
EXEC 4–39
EXEC SQL Statement Prefix 4–40
INCLUDE 4–42
INCLUDE SQLCA 4–44
INCLUDE SQLDA 4–46
WHENEVER 4–48
Dynamic SQL 4–52
Dynamic SQL Statement Syntax 4–54
DESCRIBE 4–55
EXECUTE (Dynamic SQL Form) 4–59
EXECUTE IMMEDIATE 4–61
PREPARE 4–63

Chapter 5:
Client-Server Connectivity Statements

Connecting a Client Application to the Teradata Server 5–2
CONNECT 5–5
GET CRASH 5–8
LOGOFF 5–9
LOGON 5–11
SET BUFFERSIZE 5–13
SET CHARSET 5–15
SET CONNECTION 5–17
SET CRASH 5–19

Chapter 6:
Multisession Asynchronous Programming With Embedded SQL
Multisession Programming With Embedded SQL 6–2
Embedded SQL Statements Supporting Multisession Asynchronous Request Programming 6–5
ASYNC Statement Modifier 6–6
TEST 6–9
WAIT 6–13

Section 3:
Constructs Common to Embedded SQL and Stored Procedures

Chapter 7:
Cursors and Cursor Control Statements
Cursors 7–2
Positioned Cursors 7–9
CLOSE 7–15
DECLARE CURSOR 7–17
DECLARE CURSOR (Dynamic SQL Form) 7–19
DECLARE CURSOR (Macro Form) 7–21
DECLARE CURSOR (Request Form) 7–23
DECLARE CURSOR (Selection Form) 7–26
DECLARE CURSOR (Stored Procedures Form) 7–29
FETCH (Embedded SQL Form) 7–33
FETCH (Stored Procedures Form) 7–37
OPEN (Embedded SQL Form) 7–42
OPEN (Stored Procedures Form) 7–45
POSITION 7–47
REWIND 7–49

Chapter 8:
Result Code Variables
SQLSTATE 8–2
SQLCODE 8–6
ACTIVITY_COUNT 8–10

Section 4:
SQL Stored Procedures

Chapter 9:
SQL Stored Procedures
Stored Procedure Overview 9–2
DDL Statements for Stored Procedure Objects 9–4
Granting Privileges on Stored Procedures 9–5
Performing a Stored Procedure 9–6
Restrictions on Stored Procedures 9–7
Stored Procedure Lexicon 9–8
Supported DDL Statements in Stored Procedures 9–17
Unsupported DDL Statements in Stored Procedures 9–20
Supported DCL Statements in Stored Procedures 9–21
Supported DML Statements in Stored Procedures 9–22
SQL Operations on Stored Procedures 9–23
Control Statements in Stored Procedures 9–24
Completion Condition and Exception Condition Handlers 9–25
Cursor Declarations 9–26
Rules for Using SQL Statements in Stored Procedures 9–27
Using Dynamic SQL in Stored Procedures 9–31
Self-Referencing Stored Procedures 9–34
Archiving and Restoring Stored Procedures 9–38
Control Statements 9–39
BEGIN - END 9–40
CASE Statement 9–46
DECLARE 9–55
DECLARE HANDLER 9–58
Condition Handling: Overview 9–60
DECLARE HANDLER (CONTINUE Type) 9–80
DECLARE HANDLER (EXIT Type) 9–85
DECLARE HANDLER (SQLEXCEPTION Form) 9–92
DECLARE HANDLER (SQLWARNING Form) 9–97
DECLARE HANDLER (NOT FOUND Form) 9–101
DECLARE HANDLER (Control Statement Handling) 9–105
FOR 9–111
IF 9–118
ITERATE 9–125
LEAVE 9–129
LOOP 9–132
REPEAT 9–136

SET 9–139
WHILE 9–141
Stored Procedures and Tactical Queries 9–145
Debugging Stored Procedures 9–150
Sample Stored Procedure 9–153

Section 5:
Appendixes

Appendix A:
Notation Conventions
Syntax Diagram Conventions A–2
Character Shorthand Notation Used In This Book A–6
Predicate Calculus Notation Used in This Book A–8

Appendix B:
SQL Descriptor Area (SQLDA)
The SQL Descriptor Area B–2
SQLDA Structure B–4
SQLDA Data Type Codes B–7

Appendix C:
SQL Communications Area (SQLCA)
The SQL Communications Area C–2
Result Reporting C–4
SQLCA Fields C–7
Mapping SQLCODE Values to Teradata Error Message Numbers C–11
Mapping SQLCODE Values to SQLSTATE Values C–15
Mapping SQLCODE Values to SQLSTATE Values C–16

Appendix D:
SQLSTATE Mappings
SQLSTATE Codes D–2
Mapping Teradata Error Messages to SQLSTATE Values D–5

Index Index–1



Section 1:

General SQL Data Manipulation Language

Section Contents

Chapter 1:

The SELECT Statement

Queries against a relational database are made using the SQL SELECT statement.

This chapter describes the form, usage, and examples of the SELECT statement, along with the FROM clause, and the various SELECT statement clauses.

Topics include:

- SELECT statement
- Simple SELECT statement
- SELECT INTO statement
- DISTINCT and ALL options
- FROM clause
- Derived tables
- WHERE clause
- Subqueries in search conditions
- Correlated subqueries
- GROUP BY clause
- HAVING clause
- QUALIFY clause
- SAMPLE clause
- SAMPLEID expression
- ORDER BY clause
- WITH clause

SELECT

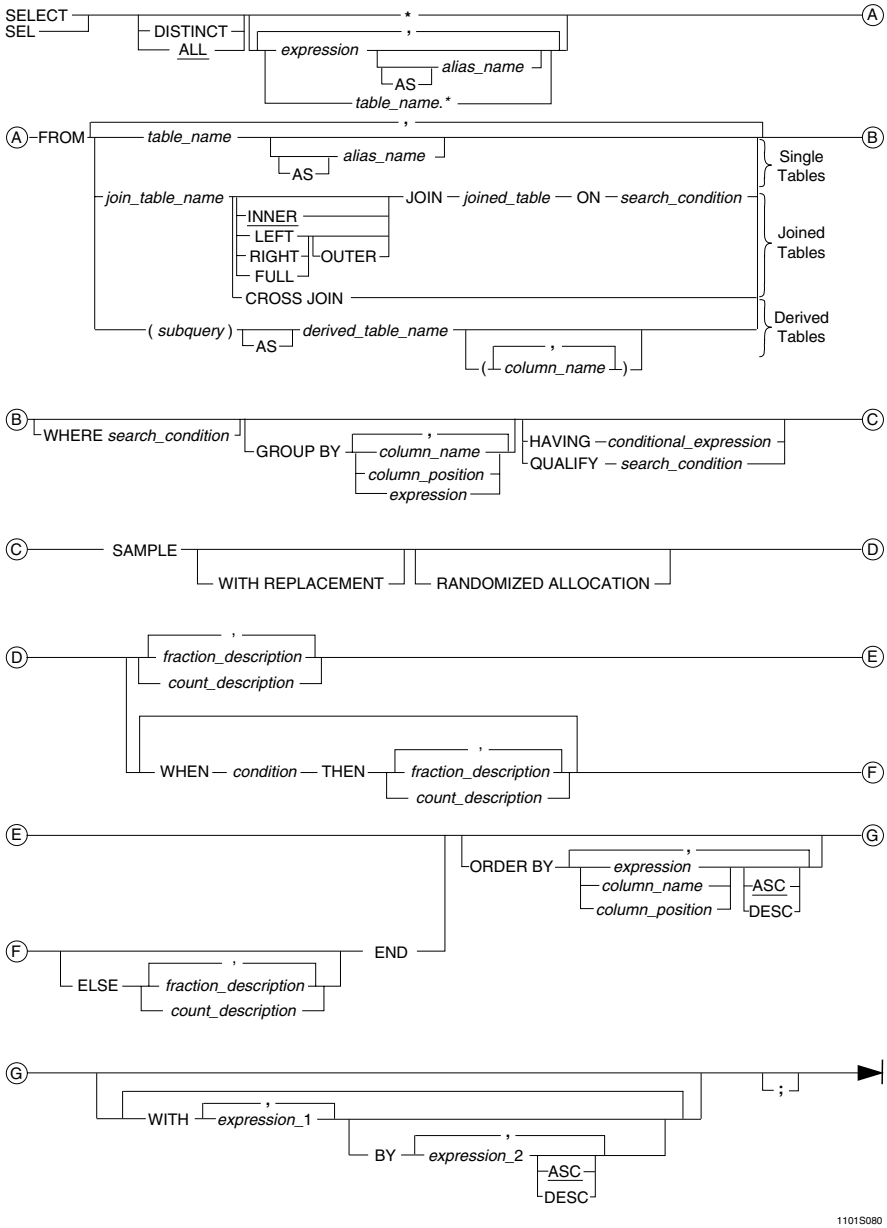
Purpose

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

Invocation

Executable.

Syntax



where:

Syntax Element ...	Specifies ...
DISTINCT	that only one row is to be returned from any set of duplicates that might result from a given expression list. Two rows are considered duplicates only if each value in one is equal to the corresponding value in the other.
ALL	that all rows, including duplicates, are to be returned in the results of the expression list. This is the default value.
*	that all columns of all tables referenced in the FROM clause be returned. When qualified by *table_name*, specifies that all columns of *table_name* only are to be returned. View columns are explicitly enumerated when views are defined; therefore, if a table is changed after a view is defined, those changes will not appear if the you perform a SELECT * query. Note that `SELECT * ...` is a column operation, projecting all the columns of a table[a], while `SELECT COUNT(*) ...` is a row operation, restricting and then counting all the rows of a table, reporting the cardinality of the table in question. The semantics of the asterisk are orthogonal in the two cases.
expression	any valid SQL expression.
AS	an optional introduction to *alias_name*.
alias_name	an alias for the table that is referenced by *table_name*. You must specify an *alias_name* for self-join operations. *alias_name* is also used to name expressions. The ANSI SQL standard refers to table aliases as correlation names. They are also referred to as range variables.
table_name	the name of a table, derived table, or view. *table_name*.* in the SELECT list can define the table from which rows are to be returned when two or more tables are referenced in the FROM clause.
FROM	an introduction to the names of one or more tables, views, or derived tables from which *expression* is to be derived. See "FROM Clause" on page 1-22 for more detail.

Syntax Element ...	Specifies ...
	Single Table
This option allows the FROM clause to specify single tables. A FROM clause can include a sequence of single table references, which creates an implicit join; to be distinguished from explicit joins in which the keyword JOIN is part of the syntax.	
table_name	the name of a single table, derived table, or view referred to in the FROM clause.
AS	an optional introduction to *alias_name*.
alias_name	an alias for the table referenced in the FROM clause. The ANSI SQL standard refers to table aliases as correlation names. They are also referred to as range variables.
	Joined Tables
Options for joined tables allow the FROM clause to specify multiple tables joined in explicit ways, as described below:	
join_table_name	the name of a joined table.
INNER	a join in which qualifying rows from one table are combined with qualifying rows from another table according to some join condition. Inner join is the default join type.
OUTER	a join in which qualifying rows from one table that do *not* have matches in the other table, are included in the join result. The rows from the first table are extended with nulls.
LEFT OUTER	the table that was listed first in the FROM clause. In a LEFT OUTER JOIN, the rows from the left table that are *not* returned in the result of the inner join of the two tables are returned in the outer join result and extended with nulls.
RIGHT OUTER	the table that was listed second in the FROM clause. In a RIGHT OUTER JOIN, the rows from the right table that are *not* returned in the result of the inner join of the two tables are returned in the outer join result and extended with nulls.
FULL OUTER	that rows be returned from both tables. In a FULL OUTER JOIN, rows from both tables that have *not* been returned in the result of the inner join will be returned in the outer join result and extended with nulls.
JOIN	an introduction to the name of the second table to participate in the join.
joined_table	the name of the joined table.

Syntax Element ...	Specifies ...
ON *search_condition*	one or more conditional expressions that must be satisfied by the result rows. An ON condition clause is required if the FROM clause specifies an outer join.
CROSS JOIN	an unconstrained, or Cartesian join; it returns all rows from all tables specified in the FROM clause.

Derived Tables

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

(*subquery*)	nested SELECT statements.
AS	an optional introductory clause to derived table name.
derived_table_name	the name of a derived table.
column_name	the column name. The *column_name* field is for the column name only; forms such as Databasename.Tablename.Columnname, or Tablename.Columnname should not be used.
WHERE	an introduction to the search condition in the SELECT statement.
search_condition	a conditional search expression that must be satisfied by the row or rows returned by the statement.
GROUP BY	an introduction to a reference to one or more expressions in the select expression list.
column_name	a group for which an aggregate or ordered analytical operation is specified in the expression list. *column_name* specifies the names of columns specified by the GROUP BY clause in the SELECT statement.
column_position	the numeric position of the columns specified by the GROUP BY clause.
expression	any valid SQL expression or expression list specified by the GROUP BY clause.
HAVING	an introduction to the conditional clause in the SELECT statement.
conditional_expression	one or more conditional expressions that must be satisfied by the result rows. Aggregate operators can be used with HAVING. HAVING condition selects rows from a single group defined in the select expression list that has only aggregate results, or it selects rows from the group or groups defined in a GROUP BY clause.

Syntax Element ...	Specifies ...
QUALIFY	an introduction to a conditional clause that, similar to HAVING, further filters rows from a WHERE clause. The major difference between QUALIFY and HAVING is that with QUALIFY the filtering is based on the result of performing various ordered analytical functions on the data.
search_condition	one or more conditional expressions that must be satisfied by the result rows. You can use ordered analytical functions with QUALIFY.
SAMPLE	an introduction to a clause that permits sampling of rows in the SELECT statement.
fraction_description	one (or many) floating point fractional constants in the closed interval (0,1). The sum of the *fraction_description* values specified cannot exceed 1.
count_description	a positive integer constant list of row counts.
ORDER BY	the order in which result rows are to be sorted.
expression	an expression in the SELECT expression list, either by name, or by means of a constant that specifies the numeric position of the expression in the expression list. If the sort field is a character string, the *expression* used in the ORDER BY phrase can include a type modifier to force the sort to be either CASESPECIFIC or NOT CASESPECIFIC. See "Defining Case Sensitivity for Table Columns" in Chapter 4: "Character Data Types" of *Teradata RDBMS SQL Reference, Volume 3*.
column_name	the names of columns used in the ORDER BY clause in the SELECT statement. These can be ascending or descending.
column_position	the numeric position of the columns specified by the ORDER BY clause. This can be ascending or descending.
ASC DESC	either an ascending or a descending sort order. ASC specifies ascending order; DESC specifies descending order. The default order is ASC. If a sort option is not given, result values are sorted in ascending order according to the client system's collating sequence. If ORDER BY is not specified, rows are returned unsorted.
WITH *expression_1*	an introduction to the condition to be fulfilled by the SELECT statement. Specifies a summary line (such as a total) for the values in a column of the select result. *expression_1* can contain one or more aggregate expressions that are applied to column values.

Syntax Element ...	Specifies ...
BY *expression_2*	one or more result columns (expressions) for which *expression_1* is provided. BY is valid only when used with WITH. *expression_2* can refer to an expression in the select expression list either by name or by means of a constant that specifies the numeric position of the expression in the expression list.
ASC DESC	either an ascending or a descending sort order. ASC specifies ascending order; DESC specifies descending order. The default order is ASC. If a sort option is not given, result values are sorted in ascending order according to the client system's collating sequence. If the BY clause is not specified, at that level, there is no ordering of the result.

a. The exception being PPI tables, where you must explicitly specify PARTITION to project the PARTITION column for the table.

ANSI Compliance

SELECT is ANSI SQL-99-compliant with extensions.

WITH, SAMPLE, and QUALIFY are extensions to the ANSI SQL-99 standard.

The syntax shown specifies the ANSI SQL-99 ordering of options FROM, WHERE, GROUP BY, HAVING, and ORDER BY. Teradata does not enforce this ordering, but you should observe it when you write applications to maintain ANSI compliance.

Authorization

To select data from a table, you must have SELECT privilege on that table.

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

Rules for Embedded SQL

The following rules apply to SELECT used with embedded SQL.

- SELECT must be performed from within a Selection DECLARE CURSOR construct.
- For updatable cursors opened for SELECT statements only, no ORDER BY, GROUP BY, HAVING or WITH... BY clause is allowed.
- The SELECT privilege is required on all tables specified in the FROM clause and in any subquery contained in the query specification, or on the database set containing the tables.

- The number of columns specified by the select list must match the number of host variable specifications.
- Values are assigned to the host variables specified in the INTO clause in the order in which the host variables were specified. A value is assigned to SQLCODE last.
- The main host variable and the corresponding column in the returned data must be of the same data type group, except that if the main host variable data type is approximate numeric, the temporary table column data type can be either approximate numeric or exact numeric.
- If the temporary table is empty, the value +100 is assigned to SQLCODE and no values are assigned to the host variables specified in the INTO clause.
- If exactly one row of data is returned, the values from the row are assigned to the corresponding host variables specified in the INTO clause.
- If more than one row of data is returned, the value -810 is assigned to SQLCODE, and no values are assigned to the host variables specified in the INTO clause.
- If an error occurs in assigning a value to a host variable, one of the values -303, -304, or -413 is assigned to SQLCODE, and no further assignment to host variables occurs.
- If a column value in the returned data is NULL and a corresponding indicator host variable is specified, the value -1 is assigned to the indicator host variable and no value is assigned to the main host variable. If no corresponding indicator host variable is specified, the value -305 is assigned to SQLCODE and no further assignment to host variables occurs.
- If a column value in the returned data is NOT NULL and a corresponding indicator host variable is specified, the indicator host variable is set as follows:
  - If the column and main host variable are of character data type and the column value is longer than the main host variable, the indicator host variable is set to the length of the column value.
  - In all other cases, the indicator variable is set to zero.
- If no other value is assigned to SQLCODE, the value zero is assigned to SQLCODE.
- Column values are set in the corresponding main host variables according to the rules for host variables.
- SELECT... INTO cannot be performed as a dynamic statement.

READ LOCK on Tables, Rows Referenced

A SELECT operation sets a Read lock on the tables or rows referenced in the SELECT statement. If the SELECT statement references a view, then a Read lock is placed on the underlying tables.

Only SELECT Uses Set Operators

The SELECT statement is the only SQL statement that can use the set operators UNION, INTERSECT, and MINUS. The set operators allow you to manipulate the answers to two or more queries by combining the results of each query into a single result set.

The set operators can also be used within the following operations:

- View definitions
- Derived tables
- Subqueries
- INSERT … SELECT statements

Joins

A SELECT operation can reference data in one or more (possibly mixed) sets of tables and views. In this case, the statement defines a join of the tables or views.

Success Response Indicates Returned Rows

When the result of a SELECT statement is returned, the activity count indicates the number of returned rows. If no rows are returned, the activity count is zero.

Uses of the SELECT Statement

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

- The set of result data that is returned
- The order in which result data is returned
- The format in which result data is returned

Be careful that SELECT statements that reference objects in multiple databases use fully qualified names. Name resolution problems may occur if databases referenced in the query contain tables or views with identical names and these objects are not fully qualified. Name resolution problems may even occur if the identically named objects are not referenced in the query.

SELECT and Subqueries

You can specify SELECT statements in an outer query, main query or subquery. The SELECT syntax requires that, if there are any subqueries in the request, tables referenced in the SELECT must be specified in a FROM clause.

If a SELECT is specified in a subquery, any table referenced in the subquery must be specified in the FROM clause of either the subquery or some outer query.

Examples

The following examples illustrate the use of SELECT.

Example 1: Miscellaneous SELECT Statements

The following SELECT statements are syntactically correct in Teradata. The first statement does not reference any tables, so it does not require a FROM clause. This syntax does not conform to the ANSI SQL-99 standard.

```
SELECT DATE, TIME;

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

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

Example 2: SELECT With Correlated Subquery

The following SELECT statement uses a correlated subquery to return the names of the employees who have the highest salary in each department.

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

Example 3: SELECT With Correlated Subquery

The following SELECT statement uses a correlated subquery to return the names of publishers with no books in the library.

```
SELECT PubName
FROM Publisher
WHERE 0 =
 (SELECT COUNT(*)
  FROM book
  WHERE book.PubNum=Publisher.pubnum);
```

See "Correlated Subqueries" on page 1-41 and the following related topics for detailed explanations of correlated subqueries and their use:

- "Correlated Subqueries and the SELECT Statement" on page 1-49
- "Correlated Subqueries and the UPDATE Statement" on page 1-52
- "Correlated Subqueries and the DELETE Statement" on page 1-54
- "Correlated Subqueries and the ABORT/ROLLBACK Statements" on page 1-56

Example 4: SELECT With PARTITION Column in Select-List

You can specify PARTITION in the select list of a query made against a PPI table. This example specifies an unqualified PARTITION column because its use in unambiguous:

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

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

```
SELECT Orders.*, Lineitem.*, Orders.PARTITION
FROM Orders, Lineitem
WHERE Orders.PARTITION = 3
AND Lineitem.PARTITION = 5
AND Orders.o_orderkey = Lineitem.1_orderkey;
```

Notice that you must specify PARTITION explicitly in the select list. If you specify * only, then PARTITION column information is not returned.

Further Information About the SELECT Statement

The sections that follow describe how to use the SELECT statement to request data from a Teradata database. Topics include the following:

- Simple SELECT statement
- DISTINCT option
- FROM clause
- WHERE clause, including subqueries
- SAMPLE clause
- GROUP BY clause
- HAVING clause
- QUALIFY clause
- ORDER BY clause
  - CASESPECIFIC option
  - International sort orders
- WITH clause
- Query expressions and set operators

Simple SELECT Statement

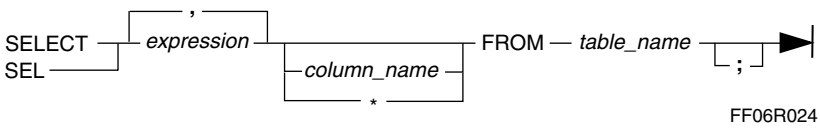
Purpose

Returns a result that is based on a list of column names or column expressions.

Invocation

Executable.

Syntax



Examples

The following four examples demonstrate the use of the simple SELECT statement.

Example 1

In the following example, Name is the only column name in the SELECT expression list.

The statement returns all of the employee names in the Employee table:

```
SELECT Name
FROM Employee;
```

Example 2

The following statement selects all of the data in the Employee table:

```
SELECT *
FROM Employee;
```

Example 3

The following statement selects the name, job title, and department number for each employee in the Employee table. In this example, Name, JobTitle, and DeptNo are the column names in the select expression list):

```
SELECT Name, JobTitle, DeptNo
FROM Employee;
```

Example 4

The following statement selects the name of, and calculates the monthly salary for, each employee:

```
SELECT Name, Salary/12 AS MSalary
FROM Employee;
```

The expression list of the preceding SELECT statement contains two expressions:

- Name
- Salary/12 AS MSalary

SELECT … INTO

Purpose

Selects at most one row from a table and assigns the values in that row to host variables in an embedded SQL application or to local variables or parameters in stored procedures.

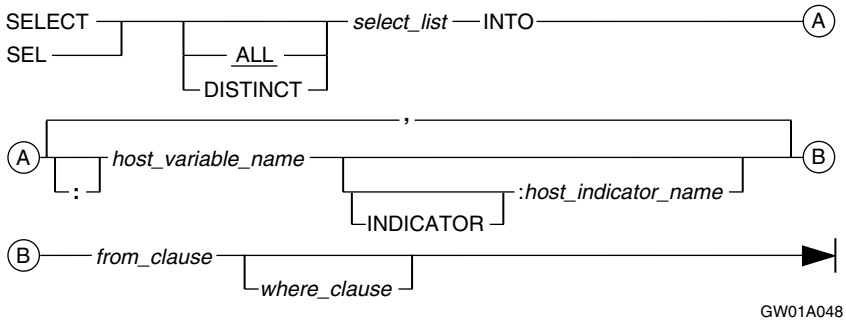
Invocation

Executable.

Embedded SQL and stored procedures only.

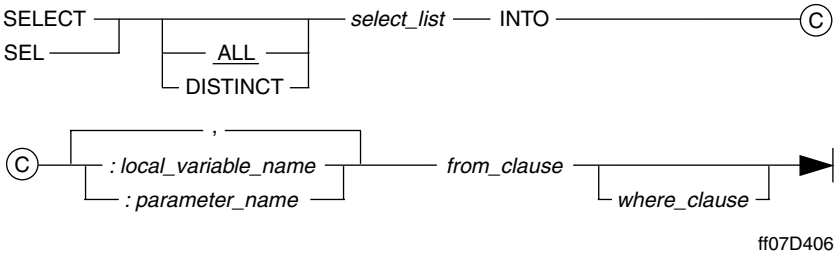
Syntax 1

This syntax is applicable to embedded SQL only.



Syntax 2

The following syntax is applicable to stored procedures only.



where:

Syntax element …	Specifies …
select_list	a complete SELECT statement. See "SELECT" on page 1-2.
host_variable_name	the name of the host variable into which the selected data is to be placed.

Syntax element ...	Specifies ...
host_indicator_name	the name of the host indicator variable.
from_clause	a clause listing the tables or views referenced by the SELECT. See "FROM Clause" on page 1-22.
where_clause	a clause narrowing a SELECT to those rows that satisfy a conditional expression that it specifies. See "WHERE Clause" on page 1-32.
local_variable_name	the name of the local variable declared in the stored procedure into which the SELECTed data is to be placed. You cannot use stored procedure status variables here.
parameter_name	the name of the stored procedure parameter into which the SELECTed data is to be placed. Only output parameters (INOUT and OUT type) can be specified.

ANSI Compliance

SELECT INTO is ANSI SQL-99-compliant.

Authorization

To select data from a table, you must have SELECT privilege on that table.

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

Colon Usage

In embedded SQL, blanks before and after a colon are optional; use of the colon before *host_variable_name* is optional; a colon must precede a *host_indicator_name*.

In stored procedures, the colon must precede a *local_variable_name* and *param_name;* blanks before and after a colon are optional.

Rules for Embedded SQL

The following rules apply to SELECT INTO use with embedded SQL:

- The SELECT privilege is required on all tables specified in the FROM clause and in any subquery contained in the query specification, or on the database(s) containing the tables.
- The number of columns specified by *select_list* must match the number of host variable specifications.
- Values are assigned to the host variables specified in the INTO clause in the order in which the host variables were specified. A value is assigned to SQLCODE last.
- The main host variable and the corresponding column in the returned data must be of the same data type group, except that if the main host variable data type is approximate numeric, the temporary table column data type can be either approximate numeric or exact numeric.
- If the temporary table contains zero rows (is empty), the value +100 is assigned to SQLCODE and no values are assigned to the host variables specified in the INTO clause.
- If exactly one row of data is returned, the values from the row are assigned to the corresponding host variables specified in the INTO clause.
- If more than one row of data is returned, the value -810 is assigned to SQLCODE, and no values are assigned to the host variables specified in the INTO clause.
- If an error occurs in assigning a value to a host variable, one of the values -303, -304, or -413 is assigned to SQLCODE, and no further assignment to host variables occurs.
- If a column value in the returned data is NULL and a corresponding indicator host variable is specified, the value -1 is assigned to the indicator host variable and no value is assigned to the main host variable. If no corresponding indicator host variable is specified, the value -305 is assigned to SQLCODE and no further assignment to host variables occurs.
- If a column value in the returned data is NOT NULL and a corresponding indicator host variable is specified, the indicator host variable is set as follows:
  - If the column and main host variable are of character data type and the column value is longer than the main host variable, the indicator host variable is set to the length of the column value.
  - In all other cases, the indicator variable is set to zero.
- If no other value is assigned to SQLCODE, the value zero is assigned to SQLCODE.
- Column values are set in the corresponding main host variables according to the rules for host variables.
- SELECT... INTO cannot be performed as a dynamic statement.

Rules for Stored Procedures

The following rules apply to the use of a SELECT INTO statement within stored procedures:

- The order of specifying the various clauses in SELECT INTO is significant in stored procedures. The following must be specified in the given order:
  - SELECT clause
  - INTO list
  - FROM clause

  If any other element intervenes between the INTO list and FROM, it will result in an error. You can specify all other clauses in the statement in any order.
- You have to specify the column list explicitly in the SELECT clause. The SELECT * syntax is not allowed in stored procedures.
- The SELECT privilege is required on all tables specified in the FROM clause and in any subquery contained in the query specification, or on the database(s) containing the tables. See "CALL" on page 3-13 for more details on required authorizations.
- UNION, INTERSECT and MINUS clauses are not supported in the SELECT INTO statement.
- The number of columns specified by the select list must match the number of local variable- and parameter- specifications.
- The local variable or parameter and the corresponding column in the returned data must be of compatible data type.

- The SELECT … INTO statement is normally expected to return at most one row. One of the following actions is taken after performing the statement.

IF SELECT ... INTO returns ...	The stored procedure status variables show these values ...	Which mean ...
more than one row	SQLCODE = 7627 SQLSTATE = '21000' ACTIVITY_COUNT = number of rows found.	an exception condition (a failure in Teradata mode and error in ANSI mode) A specific condition handler or a generic handler can be specified to handle this condition. The values of local variables and parameters do not change.
no rows, *without* an execution warning	SQLCODE = 7632 SQLSTATE = '02000' ACTIVITY_COUNT = 0	a completion condition other than successful completion. A specific condition handler can be specified to handle this completion condition. The values of local variables and parameters do not change.
no rows, *with* an execution warning	SQLCODE = the warning code. SQLSTATE = SQLSTATE value corresponding to the warning. ACTIVITY_COUNT = 0.	a completion condition other than successful completion. A specific condition handler can be specified to handle this completion condition. The values of local variables and parameters do not change.
exactly one row *without* an execution warning	SQLCODE = 0 SQLSTATE = '00000' ACTIVITY_COUNT = 1	the fetched values are assigned to the local variables and parameters. This is a successful completion. A specific handler cannot be specified to handle this.
exactly one row *with* an execution warning	*SQLCODE* = the warning code. SQLSTATE = SQLSTATE value corresponding to the warning. ACTIVITY_COUNT = 1	the fetched values are assigned to the local variables and parameters. This is a completion condition other than successful completion. A specific handler can be specified to handle this condition.

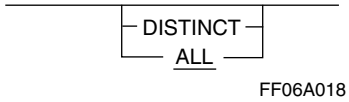
- If an error or failure occurs for the statement, normal exception condition handling takes place.

DISTINCT and ALL Options

Purpose

Specifies whether duplicate values are to be returned when an expression is processed.

Syntax



FF06A018

where:

Syntax Element ...	Specifies ...
DISTINCT	only one row is to be returned from any set of duplicates that might result from a given expression list. Two rows are considered duplicates only if each value in one is equal to the corresponding value in the other.
ALL	all rows, including duplicates, are to be returned in the results of the expression list. This is the default value. It is optional in the query (Set Operator) expressions.

ANSI Compliance

DISTINCT and ALL are ANSI SQL-99-compliant.

SELECT DISTINCT Cannot Contain a WITH Clause

When you use the DISTINCT option, the SELECT statement cannot contain a WITH clause.

Sorting Rows for Before Eliminating Duplicates

Before performing the sort operation required to eliminate duplicates, Teradata creates a sort key and appends it to the rows to be sorted. If the length of this temporary data structure exceeds the system limit of 64K bytes, the operation fails.

The BYNET only looks at the first 4 096 bytes of the sort key created to sort the specified fields, so if the field the sort key is based on is greater than 4 096 bytes, the key is truncated and the data might or might not come back in the desired order.

Examples

The following examples illustrate the use of the DISTINCT and ALL options.

Example 1

The following example returns the number of unique job titles.

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

Example 2: DISTINCT With More Than One Aggregate Operator

In a single query, only one expression can be used as the argument for an aggregate operator that uses the DISTINCT option. However, more than one aggregate operator can use DISTINCT with other occurrences of the expression in the same query.

```
SELECT SUM(DISTINCT x), AVG(DISTINCT x)
FROM …
```

DISTINCT aggregate operations can be done only at the first level of aggregation.

Example 3

The following statement can be used to list unique department numbers in the Employee table:

```
SELECT DISTINCT DeptNo
FROM Employee ;
```

The result returned is:

```
DeptNo
------
   100
   300
   500
   600
   700
```

Example 4: Using ALL

In contrast, the following statement returns a department number for every employee.

```
SELECT ALL Deptno
FROM Employee ;
```

ALL is the default option, except in query (Set Operator) expressions, where ALL is an option, but not the default.

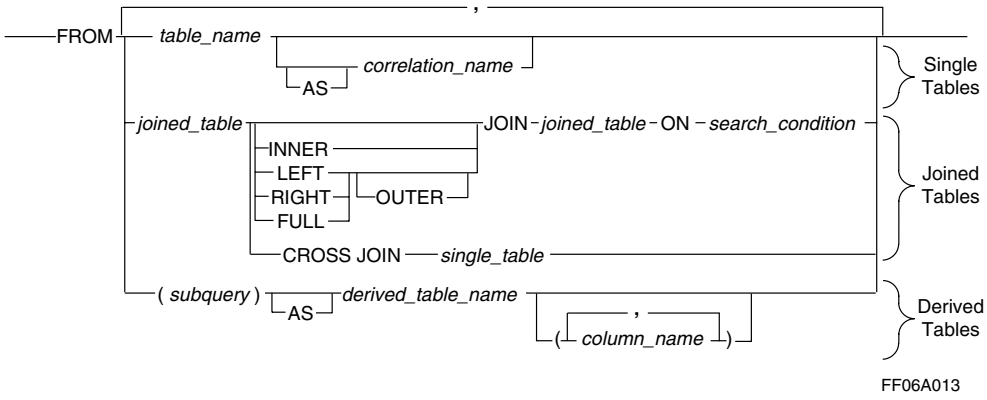
FROM Clause

Purpose

Defines either of the following things:

- the tables or views that are referenced by the SELECT operation
- a temporary alias name for a table or view for a self-join

Syntax



where:

Syntax Element ...	Specifies ...
Single Tables	
A FROM clause can include a sequence of single table references. This action creates an implicit join, which is to be distinguished from explicit joins where the keyword JOIN is part of the syntax.	
table_name	the name of a table, view or macro. If database is omitted, it is inferred from context.
AS *correlation_name*	an alias for the table that is referenced by *table_name*. *correlation_name* is required for a self-join. ANSI SQL refers to table aliases as correlation names. They are also referred to as range variables. The use of AS introducing *correlation_name* is optional.

Syntax Element ...	Specifies ...
Joined Tables	
Options for joined tables allows the FROM clause to specify multiple tables joined in explicit ways, as described below.	
joined_table	either a single table name with optional alias name, or a joined table, indicating nested joins.
INNER	a join in which qualifying rows from one table are combined with qualifying rows from another table according to some join condition. This is the default join type.
OUTER	a join in which qualifying rows from one table that do not have matches in the other table, are included in the join result. The rows from the first table are extended with nulls.
LEFT OUTER JOIN	a left outer join. LEFT indicates the table that was listed first in the FROM clause. In a LEFT OUTER JOIN, the rows from the left table that are not returned in the result of the inner join of the two tables, are returned in the outer join result, and extended with nulls.
RIGHT OUTER JOIN	a right outer join. RIGHT indicates the table that was listed second in the FROM clause. In a RIGHT OUTER JOIN, the rows from the right table that are not returned in the result of the inner join of the two tables, are returned in the outer join result, and extended with nulls.
FULL OUTER JOIN	a full outer join. FULL OUTER JOIN returns rows from both tables. In a FULL OUTER JOIN, rows from both tables that have not been returned in the result of the inner join, will be returned in the outer join result, and extended with nulls.
CROSS JOIN	a cross join. A CROSS JOIN is an unconstrained, or Cartesian join; it returns all rows from all tables specified in the FROM clause. Two joined tables can be CROSS joined.
ON *search_condition*	one or more conditional expressions that must be satisfied by the result rows. An ON condition clause is required if the FROM clause specifies an outer join.

Syntax Element ...	Specifies ...
Derived Tables	
The derived table allows the FROM clause to specify a spool file, comprised of selected data from underlying table(s). The derived table acts like a viewed table.	
subquery	the subquery that defines the derived table contents.
AS *derived_table_name*	an assigned name for the temporary derived table. AS in an introductory optional keyword for derived table.
column_name	a list of column names or expressions listed in the subquery. Allows referencing subquery columns by name.

ANSI Compliance

The FROM clause is ANSI SQL-99-compliant.

ANSI SQL also requires that all fields referenced in an outer query be in tables listed in the FROM clause.

When Fully Qualified Names Are Required

If the FROM clause refers to a table name or view name that is used in more than one database, you must use the fully qualified form of the name, for example:

```
databasename.tablename
```

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

FROM Clause Required for Subqueries

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

Self-Joins

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

For more information about self-joins, see "Self-Join" on page 2-10.

FROM Clause Using Both a Permanent Table and an Alias Name

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

For a successful self-join operation, column references must be fully qualified (that is, specified in the form tablename.columnname) and the qualifiers should involve both the permanent and the alias table names, as shown in the following example.

Once an alias is declared, any subsequent reference to the base table name causes a new instance of the table to be used; the result of the select can be a Cartesian self-join.

In the following example, table_1 is given the alias T; the subsequent use of table_1 causes a Cartesian product to be performed.

```
SELECT *
FROM table_1 T
WHERE table_1.column_1 = 2;
```

How Column References Affect Self-Joins

The effect of column references on self-join processing is as follows (these rules apply only to cases where a single aliasing FROM clause is supplied):

IF this column reference ...	IS ...	THEN a self-join is ...
all	unqualified (the preceding tablename or aliasname is omitted)	not performed.
	qualified (but the qualifiers reference only the alias name)	not performed.
some, but not all	qualified and the qualifiers reference only the alias name	not performed.
	qualified and the qualifiers reference only the permanent table name	moot because no assumptions can be made about the owner of the unqualified columns. This condition producers the following error message, where *n* is a number indicating the position of the statement in the request. *** Failure 3809 Column 'columnname' is ambiguous. Statement# n, Info = 0

Examples

The following examples illustrate the use of the FROM clause in queries.

Example 1: Simple FROM Clause

The following statement can be used to list employees who have more years of work experience than their department managers:

```
SELECT Workers.Name, Workers.YrsExp, Workers.DeptNo,
 Managers.Name, Managers.YrsExp
FROM Employee Workers, Employee Managers
WHERE Managers.DeptNo = Workers.DeptNo
AND UPPER (Managers.JobTitle) IN ('MANAGER' OR 'VICE PRES')
AND Workers.YrsExp > Managers.YrsExp ;
```

The FROM clause in the preceding statement allows the Employee table to be treated as though it were two identical tables: one named Workers and the other named Managers.

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

The request is processed by first selecting Managers rows that contain a value of either 'MANAGER' or 'VICE PRES' in the JobTitle field. These rows are then joined to the Workers rows using a merge join operation with the following join condition:

- A Workers row must contain a DeptNo value that matches the DeptNo value in a Managers row.
- The matching Workers row must also contain a YrsExp value that is greater than the YrsExp value in the Managers row.

The following result is returned:

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

Example 2: FROM Clause OUTER Join

The following example illustrates an outer join. In the example, the Skills table lists various skills and the associated skill number, and the Employee table lists employee numbers and skills.

Skills

SkillNo	SkillName
1	baker
2	doctor
3	farmer
4	lawyer
5	mason
6	tailor

Employee

EmpNo	SkillNo
6123	1
6234	1
6392	3
7281	5
7362	4
6169	1

You could use the following query to determine which skill areas do not have assigned employees:

```
SELECT Skills.SkillName, Employee.EmpNo
FROM Skills
LEFT OUTER JOIN Employee ON Skills.SkillNo = Employee.SkillNo;
```

The following result is returned. Notice that nulls are displayed as a ? character.

SkillName	EmpNo
baker	6123
baker	6234
baker	6169
doctor	?
farmer	6392
lawyer	7362
mason	7281
tailor	?

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

```
...
FROM Employee, Skills
...
```

Derived Tables

Introduction

With respect to a query, a table can be a base table, a derived table, or a view.

A derived table is obtained from one or more other tables as the result of a subquery.

ANSI Compliance

Derived tables are ANSI SQL-99-compliant.

Uses for Derived Tables

Derived tables can do all of the following things:

- Enhance performance

  How derived tables are implemented determines how or if performance is enhanced or not. For example, one way of optimizing a query is to use derived tables to control how data from different tables is accessed in joins.

  The use of a derived table in a SELECT forces the subquery to create a spool file, which then becomes the derived table. Derived tables can then be treated in the same way as base tables.
- Avoid CREATE and DROP TABLE statements for storing retrieved information
- Assist in optimizing joins. See "Inner Joins" on page 2-4 and "Outer Joins" on page 2-11.

The scope of a derived table is only visible to the level of the SELECT statement calling the subquery.

Derived tables are one of the options in the FROM clause, and can also be used in the ABORT, ROLLBACK, and DELETE statements.

Rules for Using Derived Tables

Follow these rules when using derived tables.

- A unique table name is required for the derived table.
- Qualified column names are mandatory for the select list used to build derived tables.

  This rule is parallel to, and consistent with, the rules for creating a view.

  The following query fails because the columns specified in the select list of the subquery that builds the derived table are not qualified:

```
SELECT *
FROM (SELECT *
      FROM tab1 AS t1, tab2 AS t2
      WHERE t1.col2 = t2.col3) AS derived_table;

*** Failure 3809 Column 'col1' is ambiguous.
Statement# 1, Info =0
*** Total elapsed time was 1 second.
```

  The query is correctly written as follows:

```
SELECT *
FROM (SELECT tab1.col1, tab1.col2, tab1.col3, tab1.col1,
             tab1.col2, tab1.col3
      FROM tab1 AS t1, tab1 AS t2
      WHERE t1.col2=t2.col3) AS derived_table;
```

- The number of fields in the column name list for the derived table must be the same as the number of fields in the select list of the derived table.

Derived Tables and Multilevel Aggregates

Aggregates are not allowed in a WHERE clause predicate; however, derived tables make such calculations easy to perform.

The following example shows how derived tables facilitate this.

```
SELECT Name, Salary, Average_Salary
FROM
 (SELECT AVG(Salary)
  FROM Employee) AS Workers (Average_Salary), Employee
WHERE Salary > Average_Salary
ORDER BY Salary DESC;
```

The query returns the following answer set.

Name	Salary	Average_Salary
Russel S	55 000.00	38 147.62
Watson L	56 000.00	38 147.62
Phan A	55 000.00	38 147.62
Aquilar J	45 000.00	38 147.62
Carter J	44 000.00	38 147.62

In this example, the WHERE condition compares values from rows in the base table Employee with the (in this case single) values of the field Average_Salary in the derived table Workers.

The following example is more elaborate, grouping the same information by DeptNo.

```
SELECT Name, Salary, DeptNo, Average_Salary
FROM
 (SELECT AVG(Salary), DeptNo
  FROM Employee
  GROUP BY DeptNo) AS Workers (Average_Salary, DeptNum), Employee
WHERE Salary > Average_Salary AND DeptNum = DeptNo
ORDER BY DeptNo,Salary DESC;
```

Here the request is to return salaries above the department averages.

The answer set might look like this.

Name	Salary	DeptNo	Average_Salary
Chin M	38 000.00	100	32 625.00
Moffit H	35 000.00	100	32 625,00
Russel S	55 000.00	300	47 666.67
Phan A	55 000.00	300	47 666.67
Watson L	56 000.00	500	38 285.71
Carter J	44 000.00	500	38 385.71
Smith T	42 000.00	500	38 285.71
Aguilar J	45 000.00	600	36 650.00

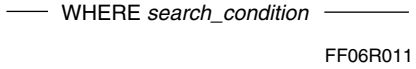
In this example, the derived table is a grouped table, providing a set of rows. The outer WHERE clause needs an equality join between the Department Number of the base table, Employee, and the derived table.

WHERE Clause

Purpose

Selects, or filters, rows that satisfy a conditional expression in SELECT, DELETE, INSERT, UPDATE, and ROLLBACK … ABORT statements.

Syntax



FF06R011

where:

Syntax Element …	Specifies …
search_condition	a search condition, also referred to as a conditional expression or predicate. The arguments can be individual values and or include subqueries, but the overall expression must be of a form that returns a single boolean (TRUE or FALSE) result. Logical expressions include comparisons of numeric values, character strings, and partial string comparisons.

ANSI Compliance

The WHERE clause is ANSI SQL-99-compliant.

Restricted Usage With Aggregates

Except when used in the ROLLBACK statement, the WHERE conditional cannot specify an aggregate operation. However, a predicate *can* be applied to an aggregate result returned by a subquery.

WHERE With AND And OR Operators

The WHERE clause can be used with the AND and OR operators. For example, the following statement selects employees who are in department 100 and who have either a college degree or at least 5 years of experience:

```
SELECT *
FROM Employee
WHERE Dept = 100
AND (EdLev >= 16
OR YrsExp >= 5);
```

WHERE In Character String Searches

The WHERE clause can be used to select data by searching for a character string.

For example, the following statement is processed by searching the Employee table for every row that satisfies the condition: the JobTitle field contains the character string “analyst”. The Name and DeptNo fields for those rows are then listed.

```
SELECT Name, DeptNo
FROM Employee
WHERE UPPER (JobTitle) LIKE '%ANALYST%' ;
```

WHERE In Joins

A WHERE clause can define a condition for joining table rows, typically for a situation in which the values in a common column must match in both tables.

Example

The following statement, which requests the name of each employee from the Employee table and the location the department for each employee from the Department table, is processed by joining the Employee and Department tables.

```
SELECT Name, Loc
FROM Employee, Department
WHERE Employee.DeptNo = Department.DeptNo ;
```

Join Efficiency and Indexes

The efficiency of a join operation depends on whether the WHERE condition uses values for columns on which primary or secondary indexes are defined.

Example

If indexes are defined on the DeptNo columns in both the Employee and Department tables, specifying an equality condition between the values in each indexed column, as in the preceding example, allows the rows in the two tables to be matched using the values in both indexed columns.

Efficiency is increased if a primary index is defined on one or both of these columns. For example, define DeptNo as the unique primary index for the Department table.

EXISTS Predicate

The EXISTS (logical ∃) predicate is supported for search conditions.

The EXISTS predicate tests the result of the subquery.

If performance of the subquery would return response rows, then the WHERE condition is considered to be satisfied. Specifying the NOT qualifier for the EXISTS predicate inverts the test.

Performance of the subquery does not return any response rows. It returns a boolean value to indicate whether responses would or would not be returned.

The subquery can be correlated with an outer query.

See "EXISTS/NOT EXISTS Predicates" in *Teradata RDBMS SQL Reference, Volume 5* for more detail on using EXISTS.

Unconstrained Joins

An unconstrained join is one in which a WHERE clause is not specified for the tables that are joined.

The result of an unconstrained join is a Cartesian product, which is almost always not the desired result. A Cartesian product is the product of the number of rows in each table that is joined.

An unconstrained join can produce a great many rows, returning a result that not only is not desired, but one that places a large performance burden on the system.

If a SELECT statement specifies both correlation and real names (for example, correlation names in a WHERE clause and real names in the SELECT), the Optimizer specifies an unconstrained join.

System-Derived PARTITION Column

You can specify a PPI partition number in a WHERE clause if the referenced table has a partitioned primary index but does not have a user-defined column named partition. See "Example 4: Delete Operation From a PPI Table Using PARTITION" on page 1-35 and "Example 5: INSERT … SELECT Operation From a PPI Table Using PARTITION" on page 1-36.

PARTITION is equivalent to a value expression where the that expression is identical to the partitioning expression defined for the primary index of the table with column references appropriately qualified as needed (or zero if no partitioning expression for the primary index is defined).

Therefore, a query against a PPI table that specifies the predicate `WHERE PARTITION <> partitioning_expression` should always return 0 rows. If any rows are returned, then they are not partitioned properly, and the table should be revalidated immediately.

Examples

The following examples illustrate the use of the WHERE clause.

Example 1: Simple Predicate

The following statement can be used to select the name and job title of every employee in Department 100. In this statement, the predicate DeptNo = 100 is the conditional expression.

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

Example 2: *table_name*.* Syntax

The following statement returns only those rows from the Employee table whose values in the EmpNo column match the values in the MgrNo column of the Department table.

Note the use of the *table_name*.* form of the select list.

```
SELECT Employee.*
FROM Employee, Department
WHERE Employee.EmpNo=Department.MgrNo;
```

Example 3: Ensure That Predicate Domains Are Consistent

If EmpNo is defined as CHARACTER(5), then EmpNo values should be compared with character values in the WHERE condition, as indicated in the following predicate.

```
…
WHERE EmpNo = '12345'
…
```

Example 4: Delete Operation From a PPI Table Using PARTITION

This example deletes all orders in the Order table from partition 1.

```
DELETE FROM Orders
WHERE Orders.PARTITION = 1;
```

Example 5: INSERT … SELECT Operation From a PPI Table Using PARTITION

This example performs an INSERT … SELECT to copy orders from partitions 1 and 2 of the Orders table into the Old_Orders table, then deletes them from Orders.

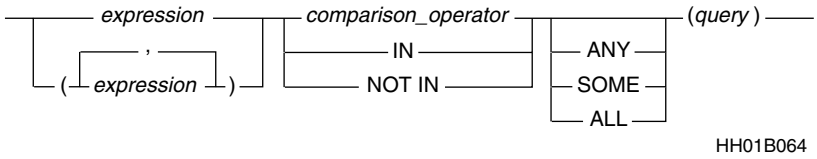
```
INSERT INTO Old_Orders
SELECT *
FROM Orders
WHERE Orders.PARTITION IN (1,2)
;DELETE FROM Orders
WHERE Orders.PARTITION IN (1,2);
```

Subqueries in Search Conditions

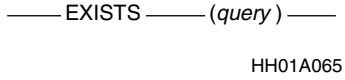
Purpose

Permits a more sophisticated and detailed query of a database through the use of nested SELECT statements.

Syntax 1



Syntax 2: Logical Expressions



ANSI Compliance

Subqueries are ANSI SQL-99-compliant.

Restriction on Number of Subqueries Permitted

You can specify no more than 64 subqueries per SELECT statement.

This limit is established using the DBSControl record field MaxParTreeSegs. The value for MaxParTreeSegs must be set to 128 or higher to define a limit of 64 subquery nesting levels.

Examples

The following examples illustrate the use of subqueries in search conditions.

Example 1

The following request selects the names and department locations of those employees who report to manager Aguilar (whose EmpNo is 10007).

```
SELECT Name, Loc
FROM Employee, Department
WHERE Employee.DeptNo = Department.DeptNo
AND Employee.DeptNo IN
 (SELECT DeptNo
  FROM Department
  WHERE MgrNo = 10007) ;
```

Example 2

The following SELECT statement finds every employee in the Employee table with a salary that is greater than the average salary of all employees in the table:

```
SELECT Name, DeptNo, JobTitle, Salary
FROM Employee
WHERE Salary >
 (SELECT AVG(Salary)
  FROM Employee)
  ORDER BY Name ;
```

The statement returns the following results table.

Name	DeptNo	JobTitle	Salary
Aguilar J	600	Manager	45 000.00
Carter J	500	Engineer	44 000.00
Omura H	500	Programmer	40 000.00
Phan A	300	Vice President	55 000.00
Regan R	600	Purchaser	44 000.00
Russell S	300	President	65 000.00
Smith T	500	Engineer	42 000.00
Smith T	700	Manager	45 000.00
Watson L	500	Vice President	56 000.00

Example 3

The following SELECT statement retrieves the employees with the highest salary and the most years of experience.

The query returns this many rows …	WHEN this condition evaluates to TRUE …
0	employees with the highest salary do not also have the most years experience.
1	there is only one employee with the highest salary and years experience.
multiple	there is more than one employee with the highest salary and years experience.

```
SELECT EmpNo,Name,JobTitle,Salary,YrsExp
FROM Employee
WHERE (Salary,YrsExp) >= ALL
 (SELECT Salary,YrsExp
  FROM Employee) ;
```

The result shows that only one employee has both the highest salary and the most years of experience.

EmpNo	Name	JobTitle	Salary	YrsExp
10018	Russell S	President	65 000.00	25

Example 4

The following statement uses a nested ordered analytical function and a HAVING clause to find those items that appear in the top 1 percent of profitability in more than 20 stores.

```
SELECT Item, COUNT(Store)
FROM
 (SELECT Store, Item, Profit, QUANTILE(100, Profit) AS Percentile
  FROM
   (SELECT Item, SUM(Sales) - COUNT(Sales) * Items.ItemCOST) AS Profit
    FROM DailySales, Items
    WHERE DailySales.Item = Items.Item
    GROUP BY Item) AS ItemProfit
  GROUP BY STORE
  QUALIFY Percentile = 0) AS TopOnePercent
GROUP BY Item
HAVING COUNT(Store) >= 20 ;
```

The results of this query might look something like the following table.

Item	Count(Store)
Chilean sea bass	99
Mackerel	149
Tako	121
Herring	120
Salmon	143

Example 5

The following example uses nested RANK functions and QUALIFY clauses to report the top 100 items by profitability and the top items by revenue, matching those that appear in both lists using an OUTER JOIN.

```
SELECT *
FROM
 (SELECT Item, Profit, RANK(Profit) AS ProfitRank
  FROM Item, Sales
  QUALIFY ProfitRank <= 100 AS P)
FULL OUTER JOIN
 (SELECT Item, Revenue, RANK(Revenue) AS RevenueRank
  FROM Item, Sales
  QUALIFY RevenueRank <= 100 AS R)
ON P.Item = R.Item;
```

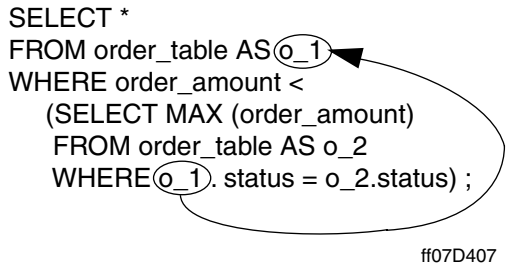
The results of this query might look something like the following table.

Item	Profit	ProfitRank	Item	Revenue	RevenueRank
Dress slacks	17804	74	Dress slacks	180211	56
Dress shirts	16319	68	?	?	?
Dresses	55888	82	Dresses	652312	77
Blouses	9849	48	Blouses	771849	92
?	?	?	Skirts	817811	55

Correlated Subqueries

Definitions

A subquery is said to be correlated when it references columns of outer tables in an enclosing, or containing, outer query. The reference from the inner query to its containing query is indicated explicitly in the following illustration.



The expression *correlated subquery* comes from the explicit requirement for the use of correlation names (table aliases) in any correlated subquery in which the same table is referenced in both the internal and external query. When the inner table references a foreign key column of a different table, then there is no need to use correlation names to distinguish the relevant columns, but they must be fully qualified within the interior WHERE clause.

The effect of a correlated subquery is to provide an implicit loop function within any standard SQL DML statement, including ABORT, DELETE, INSERT, and UPDATE (see "Correlated Subqueries and the ABORT, DELETE, INSERT, and UPDATE Statements" on page 1-46).

A reference to variables in an outer expression by a variable in an inner expression is called an outer reference.

IF a scalar subquery references tables in its containing query that are …	THEN it is said to be …
not referenced in its own FROM clause	correlated.
referenced in its own FROM clause	uncorrelated.

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

ANSI Compliance

Correlated subqueries are ANSI SQL-99-compliant.

Behavior of Outer References

Outer references behave as described in the following table. This process does not mirror the query plan generated by the Optimizer: it is meant only to describe how a correlated subquery works at a conceptual level.

Stage	Process
1	For each row of an outer query, the values of the outer references in that row are used to evaluate the result of the inner subquery.
2	The inner subquery expression result is joined with its associated row from the outer query based on the specified join constraint for the inner query.

The semantics of correlated subqueries imply that an inner subquery expression is executed once for *each* row of its immediate outer query expression. The semantics do *not* guarantee that each iteration of the subquery produces a unique result.

The following example of this behavior uses the following simple Employee table.

EmpNo	EmpName	Sex	Age
PI	NN		
101	Friedrich	F	23
102	Harvey	M	47
103	Agrawal	M	65
104	Valduriez	M	34
105	Cariño	F	39
106	Au	M	28
107	Takamoto	F	51
108	Ghazal	F	26

The following SELECT statement demonstrates the behavior of a simple correlated subquery. Because the same table is referenced in both the inner and outer queries, *both* references are given correlation names for clarity, though only one of them (it makes no difference which) *must* be aliased.

```
SELECT *
FROM employee AS e1
WHERE age <
 (SELECT MAX(age)
  FROM employee AS e2
  WHERE e1.sex = e2.sex);
```

Stage 1

Two copies of the table described earlier are generated, one as e1 and the other as e2.

Stage 2

Evaluation of the inner query requires data from the outer, containing, query.

The evaluation of the inner query becomes a set of virtual individual queries such as the following.

```
SELECT 101, 'Friedrich', 'F', 23
FROM employee AS e1
WHERE 23 <
 (SELECT MAX(age)
  FROM employee AS e2
  WHERE 'F' = e2.sex;
⋮

SELECT 108, 'Ghazal', 'F', 26
FROM employee as e1
WHERE 26 <
 (SELECT MAX(age)
  FROM employee AS e2
  WHERE 'F' = e2.sex;
```

Stage 3

The expanded individual queries once the subquery has been evaluated for each row in the inner query look like the following.

```
SELECT 101, 'Friedrich', F, 23 FROM employee AS e1 WHERE 23 < 51;
SELECT 102, 'Harvey', M, 47 FROM employee AS e1 WHERE 47 < 65;
SELECT 103, 'Agrawal', M, 65 FROM employee AS e1 WHERE 65 < 65;
SELECT 104, 'Valduriez', M, 34 FROM employee AS e1 WHERE 34 < 65;
SELECT 105, 'Cariño', F, 39 FROM employee AS e1 WHERE 39 < 51;
SELECT 106, 'Au', M, 28 FROM employee AS e1 WHERE 28 < 65;
SELECT 107, 'Takamoto', F, 51 FROM employee AS e1 WHERE 51 < 51;
SELECT 108, 'Ghazal', F, 26 FROM employee AS e1 WHERE 26 < 51;
```

Stage 4

The same evaluations are performed for each row in the outer query.

Stage 5

Employee 103, Agrawal, is eliminated from the result because his age is not less than the maximum male age in the table. Similarly, employee 107, Takamoto, is eliminated because her age is not less than the maximum female age in the table.

The final result is reported in the following table.

EmpNo	EmpName	Sex	Age
101	Friedrich	F	23
102	Harvey	M	47
104	Valduriez	M	34
105	Cariño	F	39
106	Au	M	28
108	Ghazal	F	26

Rules for Using Correlated Subqueries

The rules for using correlated subqueries are as follows:

- Correlated subqueries can specify combinations of equality and inequality between the tables of an outer query and the tables of a subquery.

  For example, the following SELECT statement specifies an equality condition in its subquery and specifies an inequality condition in the main query.

  ```
  SELECT *
  FROM table_1
  WHERE x < ALL
   (SELECT y
    FROM table_2
    WHERE table_2.N = table_1.N);
  ```

- Correlated subqueries can be nested to an arbitrary depth.

  For example, the following SELECT statement specifies three levels of depth.

  ```
  SELECT *
  FROM table_1
  WHERE x =
   (SELECT y
    FROM table_2
    WHERE table_2.N = table_1.N
    AND T2.M =
     (SELECT y
      FROM table_3
      WHERE table_3.N = table_2.M
      AND table_1.L = table_3.L));
  ```

- A correlated subquery can reference the same table as a table specified in the FROM clause of an outer query.

  To differentiate between the references to the inner and outer tables, one of them must to be renamed with a correlation name.

  In the first example, the outer table is aliased, while in the second example, the inner table is aliased.

```
SELECT *
FROM table_1 AS A
WHERE x <
 (SELECT AVG(table_1.X)
  FROM table_1
  WHERE table_1.N = A.N);

SELECT *
FROM table_1
WHERE x <
 (SELECT AVG(A.X)
  FROM table_1 AS A
  WHERE table_1.N = A.N);
```

Ensuring That Subqueries Are Uncorrelated

To ensure that a subquery is performed as an *un*correlated subquery and not as a correlated subquery, observe the following guidelines.

- For each main and subquery you code, ensure that its scope is local by specifying a FROM clause that includes all the tables it references.
- Instead of specifying join operations in the main queries of DELETE and UPDATE statements, restrict them to subqueries.

Correlated Subqueries and the ABORT, DELETE, INSERT, and UPDATE Statements

Correlated subqueries can be used with the ABORT, DELETE, INSERT, and UPDATE statements as well as with the SELECT statement.

For information about using correlated subqueries with this statement ...	See ...
ABORT	• "Correlated Subqueries and the ABORT/ROLLBACK Statements" on page 1-56. • "ABORT With a WHERE Clause" on page 3-4.
DELETE	• "Correlated Subqueries and the DELETE Statement" on page 1-54 • "Subqueries in DELETE Statements" on page 3-38.
INSERT	"Subqueries In INSERT Statements" on page 3-113.
UPDATE	• "Correlated Subqueries and the UPDATE Statement" on page 1-52. • "Subqueries in UPDATE Statements" on page 3-155.

Comparing the Behavior of Uncorrelated and Correlated Subqueries

In the following SELECT statement, no entry in predicate_2 of the subquery references any entry in table_list_1 of the main query; therefore, the subquery is completely self-contained and is uncorrelated. A self-contained subquery is also referred to as a local subquery.

```
SELECT column_list_1
FROM table_list_1
WHERE predicate_1
 (SELECT column_list_2
  FROM table_list_2
  WHERE predicate_2);
```

The relationship between inner and outer query predicates is referred to as their correlation. The subquery in the previous statement is said to be *un*correlated because it does not reference tables in the main query. Because its subquery is local, the statement restricts the number of iterations of its subquery to one. The results of the query are then joined with the results of the query made by the main SELECT statement.

Correlated subqueries perform the subquery in parentheses once for *each* result row of the main query. When it references tables in the main query that are not referenced within itself, a subquery is said to be correlated because its result is directly correlated with the main query. A correlated subquery performs once for each value from its containing query. It does *not* necessarily produce a unique result for each of those iterations.

The following example demonstrates the behavior of an SQL statement that contains a correlated subquery.

Assume that table_1 has columns field_1 and field_2, while table_2 has columns field_3 and field_4. The following four rows exist in the two tables.

field_1	field_2
100	1
50	1
20	2
40	2

field_3	field_4
100	1
50	1
20	2
40	2

- The following SELECT statement, containing a correlated subquery, is used to query the tables.

```
SELECT *
FROM table_1
WHERE field_1 IN
 (SELECT MAX(field_3)
  FROM table_2
  WHERE table_1.field_2=table_2.field_4);
```

  The statement returns two response rows because the subquery is performed four times: once for each row in table_1.

  The result contains only two response rows (and not four) because of the MAX(field_3) constraint, because two of the subquery executions return a response row where field_1 is *not* in the result.

  The two rows returned are these.

  - field_1 = 100, field_2 = 1
  - field_1= 40, field_2 = 2.

  The four executions of the subquery return the following response rows.

f3	f4
100	1
100	1
40	2
40	2

  Only the first and fourth rows of table_1 have a value for field_1 in this result set. If you had not specified the MAX function, then all four rows of table_1 would have been returned.

The FROM clause is . . .	In the main query of these SQL statements . . .
not required	• ABORT • DELETE • SELECT • UPDATE
optional	• ABORT • SELECT • UPDATE
optional depending on the syntax used	DELETE

Combining Equality and Inequality Constraints in a Correlated Subquery

You can use correlated subqueries to specify a combination of equality and inequality constraints with the subquery.

For example, to select the names of all students who are younger than all students at the same grade, you can perform the following query.

```
SELECT name
FROM student st1
WHERE age < ALL
 (SELECT age
  FROM student st2
  WHERE st1.grade = st2.grade
  AND st1.stno <> st2.stno);
```

Statement-Specific Behavior of Correlated Subqueries

Additional statement-specific information about correlated subqueries is described in:

- “Correlated Subqueries and the SELECT Statement” on page 1-49
- “Correlated Subqueries and the UPDATE Statement” on page 1-52
- “Correlated Subqueries and the DELETE Statement” on page 1-54
- “Correlated Subqueries and the ABORT/ROLLBACK Statements” on page 1-56

Correlated Subqueries and the SELECT Statement

Rules

The following rules apply to using SELECT statements with correlated subqueries:

- A SELECT statement can be used in an outer query, a main query, or a subquery.

IF a SELECT is specified in ...	THEN ...
the main query	any table referenced in the main query should be specified in the FROM clause of the main query SELECT.
a subquery	any table referenced in that subquery must be specified either in the FROM clause of that query or in some outer query. If the column reference is to the FROM clause of an outer query, then it must be fully qualified.

- Correlated subqueries can be used in a FROM clause under the following conditions:

IF a table referenced in a subquery is identified in the FROM clause of ...	THEN the table reference is ...
a subquery	local and the subquery is uncorrelated.
an outer query	not local and the subquery is correlated.

  - A FROM clause is required in any query specification.
  - The FROM clause is not required in the main query (see the bullet under "Rules for Using Correlated Subqueries" on page 1-44). However, you should always write your applications to specify a FROM clause because future releases might enforce the ANSI SQL-99 rule that all tables referenced in a SELECT must be specified in a FROM clause.

    A Warning message is returned if the referenced table does not appear in the FROM clause.

SELECT COUNT(*)

Use the following query to select the names of the publishers whose book count values in the library match the actual count of books.

```
SELECT pubname, BookCount
FROM library
WHERE (BookCount, pubnum) IN
 (SELECT COUNT(*), book.pubnum
  FROM book
  GROUP BY pubnum);
```

If the book count for a publisher in the library is zero, then the name of that publisher is not returned because no row is returned by the subquery for this publisher.

Another equivalent SELECT statement uses two uncorrelated subqueries to return the same answer set as follows.

```
SELECT pubname, BookCount
FROM library
WHERE (BookCount, pubnum) IN
 (SELECT COUNT(*), pubnum
  FROM book
  GROUP BY pubnum)
OR NOT IN
 (SELECT book.pubnum
  FROM book
  GROUP BY pubnum)
AND BookCount = 0;
```

The following SELECT statement, which is less complicated and more elegant, uses a correlated subquery to return the same correct answer as the previous query.

```
SELECT pubname, BookCount
FROM library
WHERE BookCount IN
 (SELECT count(*)
  FROM book
  WHERE book.pubnum = library.pubnum);
```

EXISTS Predicate and Correlated Subqueries

The EXISTS predicate is supported as the predicate of a search condition in a WHERE clause.

EXISTS tests the result of the subquery. The subquery is performed for each row of the table specified in the outer query when the subquery is correlated.

If the subquery returns response rows, then its WHERE condition is satisfied.

Examples

The following simple SELECT statements are syntactically correct in all releases.

```
SELECT CURRENT_DATE, CURRENT_TIME;

SELECT x1,x2
FROM table_1,table_2
WHERE table_1.x1 = table_2.x2;

SELECT x1
FROM table_1
WHERE x1 IN
 (SELECT x2
  FROM table_2);

SELECT table_1.x1;

SELECT table_1.x1
FROM table_1
WHERE table_1.x1 = table_2.x2;
```

The following SELECT statement returns the names of employees having the highest salary in each department.

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

Correlated Subqueries and the UPDATE Statement

Correlated Subqueries and the WHERE Clause

Correlated subqueries are valid in the search condition of the WHERE clause of an UPDATE statement.

Rules

Some forms of the UPDATE statement have an optional FROM clause.

- If a FROM clause is specified for the UPDATE syntax you are using, then any alias name used must be specified in the FROM clause.
- If an alias is specified for the updated table name in the FROM clause, this alias name, instead of the original name, must be used as the *table_name* that follows the UPDATE keyword. This FROM clause is optional if no joined tables are specified for an UPDATE.
- If an inner query column specification references an outer FROM clause table, then the column reference must be fully qualified.
- If the FROM clause is omitted, you can specify an alias name for the *table_name* that follows the UPDATE keyword.

Example Set 1

The following updates are syntactically correct.

```
UPDATE table_1
SET x1=1
WHERE x1 IN
(SELECT x2
FROM table_2);

UPDATE table_1 AS a
SET x1=1
WHERE a.x2=2;

UPDATE table_1
FROM table_1
SET x1=1
WHERE table_1.x1=2;
```

Example Set 2

The following update is syntactically correct, but it is not a recommended form because the table being updated is not specified explicitly.

```
UPDATE table_1
SET x1=1
WHERE table_1.x=table_2.x;
```

The recommended form for the preceding statement is the following syntax.

```
UPDATE table_1
FROM table_1, table_2
SET x1=1
WHERE table_1.x = table_2.x;
```

Correlated Subqueries and the DELETE Statement

Rules

The following rules apply to correlated subqueries used in a DELETE statement.

- The DELETE statement requires that if joined tables are specified for a delete, all tables referenced in the DELETE must be specified in the FROM clause, including the deleted table.

  A *table_name* must be added to specify the deleted table name in this case.
- All alias names must be defined in the FROM clause.

  ANSI calls table aliases correlation names. They are also referred to as range variables.
- If an alias is defined for the deleted table name in the FROM clause, this alias name (and *not* the original table name) must be used as the *table_name* that follows the DELETE keyword.
- If an inner query column specification references an outer FROM clause table, then the column reference must be fully qualified.
- The *table_name* preceding the FROM clause is optional if no joined tables are specified for deletion.

Example Set 1

The following DELETE statements are syntactically correct:

```
DELETE
FROM table_1
WHERE x1 IN
(SELECT x2
FROM table_2);

DELETE
FROM table_1 AS a
WHERE a.x1=2;
DELETE table_1
FROM table_1
WHERE table_1.x1=2;

DELETE table_1
FROM table_1, table_2
WHERE table_1.x1=table_2.x2;

DELETE a
FROM table_1 AS a, table_2
WHERE a.x1=table_2.x2;
```

Example Set 2

The following DELETE statement is syntactically correct, but it is not a recommended syntax because the table from which rows are to be deleted is not named explicitly.

```
DELETE
FROM table_1
WHERE table_1.x1=table_2.x2;
```

The recommended syntax for this DELETE statement is the following.

```
DELETE table_1
FROM table_1,table_2
WHERE table_1.x1 = table_2.x2 ;
```

Example Set 3

The following DELETE statements are not valid.

```
DELETE table_1
FROM table_1 AS a, table_2
WHERE a.x1=table_2.x2;

DELETE table_1 AS a
FROM table_1, table_2
WHERE a.x1=table_2.x2;
```

Correlated Subqueries and the ABORT/ROLLBACK Statements

Rules

The following rules apply to correlated subqueries used in an ABORT or ROLLBACK statement:

- A FROM clause is required for an ABORT statement if the ABORT references a table. All tables referenced in an ABORT statement *must* be defined in a FROM clause.
- If an inner query column specification references an outer FROM clause table, then the column reference must be fully qualified.
- The ABORT and ROLLBACK statements are supported as extensions to ANSI syntax.

A WHERE clause is supported as part of the ABORT and ROLLBACK statements.

Correlated subqueries and the EXISTS predicate are supported in this WHERE clause.

Examples

The following ABORT statements are syntactically correct.

```
ABORT
WHERE user = 'DBC';

ABORT FROM table_1
WHERE table_1.x1 > 1;

ABORT FROM table_1
WHERE table_1.x1 = table_2.x2;

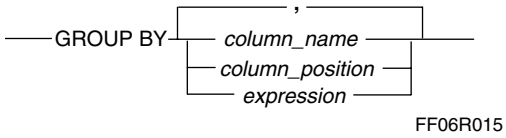
ABORT FROM table_1, table_2
WHERE table_1.x1 = table_2.x2;
```

GROUP BY Clause

Purpose

Groups result rows by the values in one or more columns.

Syntax



where:

Syntax Element …	Specifies …
GROUP BY	a reference to one or more expressions in the select expression list.
column_name	a set of column names drawn from the list of tables specified in the FROM clause of the SELECT statement that is used in the GROUP BY clause to specify the columns by which data is to be grouped.
column_position	the sequential numeric position of columns within the *column_list* clause of the SELECT statement that is used in the GROUP BY clause to specify the order by which data is to be grouped. This must be a positive integer. Use of *column_position* is a Teradata extension to the ANSI SQL-99 standard.
expression	any list of valid SQL expressions specified for the GROUP BY clause. *column_name*, *column_position*, and *expression* can comprise single entries or a list. Use of *expression* is a Teradata extension to the ANSI SQL-99 standard.

ANSI Compliance

The GROUP BY clause is ANSI SQL-99-compliant with extensions.

Aggregate Handling

When an aggregate operation (for example, SUM, AVERAGE, MAX, MIN, or COUNT) is specified, GROUP BY can be used to return a summary row for each group. Aggregate operators can be used only in the SELECT expression list or in the optional HAVING clause.

All nonaggregate groups in a SELECT expression list must be included in the GROUP BY clause.

Combining GROUP BY With ORDER BY

If you specify an ORDER BY clause, then any group contained in the ORDER BY clause must also be included in the GROUP BY clause.

Sorting Rows for Grouping

Before performing the sort operation that groups the rows to be returned to the requestor, Teradata creates a sort key and appends it to the rows to be sorted. If the length of this temporary data structure exceeds the system limit of 64K bytes, the operation fails.

The BYNET only looks at the first 4 096 bytes of the sort key created to sort the specified fields, so if the field the sort key is based on is greater than 4 096 bytes, the key is truncated and the data might or might not come back in the desired order.

Examples

The following examples illustrate the use of the GROUP BY clause.

Example 1

If the following statement were used to generate a report of salary totals by department, the result might look something like the report that follows.

```
SELECT DeptNo, SUM(Salary)
FROM Employee
GROUP BY DeptNo;
```

DeptNo	Sum(Salary)
100	180 500.00
300	143 000.00
500	268 000.00
600	146 600.00
700	113 000.00

Example 2

The following statement returns an error message because a nonaggregate group listed in the SELECT expression list (Department.DeptName) is not listed in the GROUP BY clause:

```
SELECT Employee.DeptNo, Department.DeptName, AVG(Salary)
FROM Employee, Department
WHERE Employee.DeptNo = Department.DeptNo
GROUP BY Employee.DeptNo;
```

Example 3

The following statement returns an error message because the nonaggregate group listed in the ORDER BY clause is not listed in the GROUP BY clause:

```
SELECT Employee.DeptNo, AVG(Salary)
FROM Employee, Department
WHERE Employee.DeptNo = Department.DeptNo
ORDER BY Department.DeptName
GROUP BY Employee.DeptNo;
```

Example 4

The following statement uses GROUP BY with an ordered analytical function to generate report breaks where the function resets and computes a new value for the next grouping.

The example groups all items into percentile by profitability for each store and then returns only the items of interest, which, in this case, are the lowest percentile for each store.

```
SELECT Store, Item, Profit, QUANTILE(100, Profit) AS
Percentile
FROM
 (SELECT Item, SUM(Sales) - (COUNT(Sales)*Items.ItemCost) AS
   Profit
  FROM DailySales, Items
  WHERE DailySales.Item = Items.Item
  GROUP BY Item) AS ItemProfit
GROUP BY Store
QUALIFY Percentile = 99;
```

The result of this query might look something like the following table.

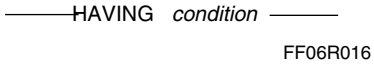
Store	Item	Profit	Percentile
Eastside	Golf balls	100.19	99
Westside	Tennis balls	- 110.00	99
Central	Bowling balls	- 986.81	99
South	Codfish balls	- 1 891.89	99
North	Fishing poles	1 180.88	99

HAVING Clause

Purpose

Specifies a conditional expression that must be satisfied for a group of rows to be included in the result data.

Syntax



FF06R016

where:

Syntax Element ...	Specifies ...
HAVING	the conditional clause in the SELECT statement.
condition	one or more conditional expressions that must be satisfied by the result rows. Aggregate operators can be used with HAVING. HAVING *condition* selects rows from a single group defined in the SELECT expression list that has only aggregate results, or it selects rows from the group or groups defined in a GROUP BY clause.

ANSI Compliance

The HAVING clause is ANSI SQL-99-compliant.

Usage Notes

- The conditional expression can define one or more aggregates (for example, MAX, MIN, AVG, SUM, COUNT) and can be applied to the rows of the following group conditions:
  - A single group defined in the SELECT expression list, which has only aggregate results
  - One or more groups defined in a GROUP BY clause
- When the WHERE, GROUP BY, and HAVING clauses are used together in a SELECT statement, the order of evaluation is:

  1 WHERE

  2 GROUP BY

  3 HAVING

- If the tables referenced in a HAVING clause meet any of the following criteria, an error is returned to the requestor:
  - Are not declared in the FROM clause
  - Do not appear in the SELECT expression list
  - Do not appear in a non-aggregate condition.

Examples

The following examples illustrate the use of the HAVING clause.

Example 1

The following statement can be used to display salary ranges for specified departments whose salaries average more than $37,000:

```
SELECT DeptNo, MIN(Salary), MAX(Salary), AVG(Salary)
FROM Employee
WHERE DeptNo IN (100,300,500,600)
GROUP BY DeptNo
HAVING AVG(Salary) > 37000;
```

The result is:

```
DeptNo  Minimum(Salary)  Maximum(Salary)  Average(Salary)
------  --------------- ---------------   ---------------
   300        23,000.00       65,000.00         47,666.67
   500        22,000.00       56,000.00         38,285.71
```

Example 2

The following example demonstrates the use of HAVING as applied to the aggregate results of a single group defined in the SELECT expression list. This application is particularly useful in a SELECT subquery.

```
SELECT COUNT(Employee)
FROM Department
WHERE DeptNo = 100
HAVING COUNT(Employee) > 10 ;
```

The following SELECT statements are additional examples of the correct use of the HAVING clause.

```
SELECT SUM(t.a)
FROM t,u
HAVING SUM(t.a)=SUM(u.a);

SELECT SUM(t.a), SUM(u.a)
FROM t,u
HAVING SUM(t.b)=SUM(u.b);

SELECT SUM(t.a)
FROM t,u
HAVING SUM(t.b)=SUM(u.b)
AND u.b = 1
GROUP BY u.b;
```

Aggregating a Join In a HAVING Clause

Teradata SQL supports the HAVING clause when referencing columns from two or more tables.

Example

The columns named price and sales_qty are from two different tables table_1, UnitPriceCost, and table_2, SalesHist. Use the following SELECT statement to find which category of items is sold for a profit margin greater than $1000.

```
SELECT table_1.category,
(table_2.price - table_2.cost) * SUM (table_1.sales_qty) AS
margin
FROM SalesHist table_1, UnitPriceCost table_2
WHERE table_1.prodno=table_2.prodno
GROUP BY table_1.category,table_2.price, table_2.cost
HAVING margin > 1000;
```

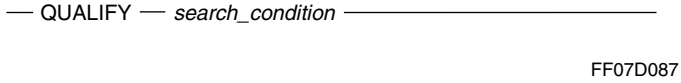
A subquery can have a join on a view with an aggregate operation and a HAVING clause that references more than one table.

QUALIFY Clause

Purpose

Filters results of a previously computed ordered analytical function according to user-specified conditions.

Syntax



FF07D087

where:

Syntax element ...	Specifies ...
QUALIFY	a conditional clause in the SELECT statement.
search_condition	one or more conditional expressions that must be satisfied by the result rows. You can use aggregate operators within a QUALIFY clause.

ANSI Compliance

The QUALIFY clause is a Teradata extension to the ANSI standard.

Usage Notes

- When you specify a QUALIFY clause in a query, you must also specify a statistical function in one of the following locations within the query.
  - Select list of the SELECT clause
  - Grouping key of the GROUP BY clause
  - Search condition of the QUALIFY clause
- If you do not specify a statistical function in one of these clauses, then the query fails and returns a 3610 error.
- When the WHERE, GROUP BY, and QUALIFY clauses are used together in a SELECT statement, the order of evaluation is:

  1 WHERE

  2 GROUP BY

  3 QUALIFY

In detail, the steps are these.

Stage	Process
1	The WHERE clause conditions are evaluated on the FROM clause tables.
2	The resulting rows are grouped using the GROUP BY columns.
3	The ordered analytical functions are evaluated on the grouped table.
4	The QUALIFY clause is applied to the resulting set.

The statistical functions invoked in both the select_list and in the search_condition of the QUALIFY clause are computed on the grouped rows without any rows eliminated and then the search_condition of the QUALIFY clause is applied.

- If the tables referenced in a QUALIFY clause meet any of the following criteria, an error is returned to the requestor:
  - Are not declared in the FROM clause
  - Do not appear in the SELECT expression list
  - Do not appear in a non-aggregate condition.

The spirit of QUALIFY is similar to that of HAVING in that rows are eliminated based on the value of a function computation. With QUALIFY, rows are eliminated based on the computation of the ordered analytical functions.

Example 1

The following statement displays each item in a Sales table, its total sales, and its rank within the top 100 selling items:

```
SELECT ItemID, sumPrice, RANK(sumPrice)
FROM
(SELECT a1.itemID, SUM(a1.Price)
FROM Sales a1
GROUP BY a1.itemID) AS T1 (itemID, sumPrice)
QUALIFY RANK(sumPrice) <=100;
```

Example 2

The following example reports the bottom percentile of items by profitability:

```
SELECT Item, Profit, QUANTILE(100, Profit) AS Percentile
FROM
(SELECT Item, SUM(Sales) - (COUNT(Sales) * Items.ItemCost) AS
Profit
FROM DailySales, Items
WHERE DailySales.Item = Items.Item
GROUP BY Item) AS ItemProfit
QUALIFY Percentile = 99;
```

The results of this query might look something like the following table.

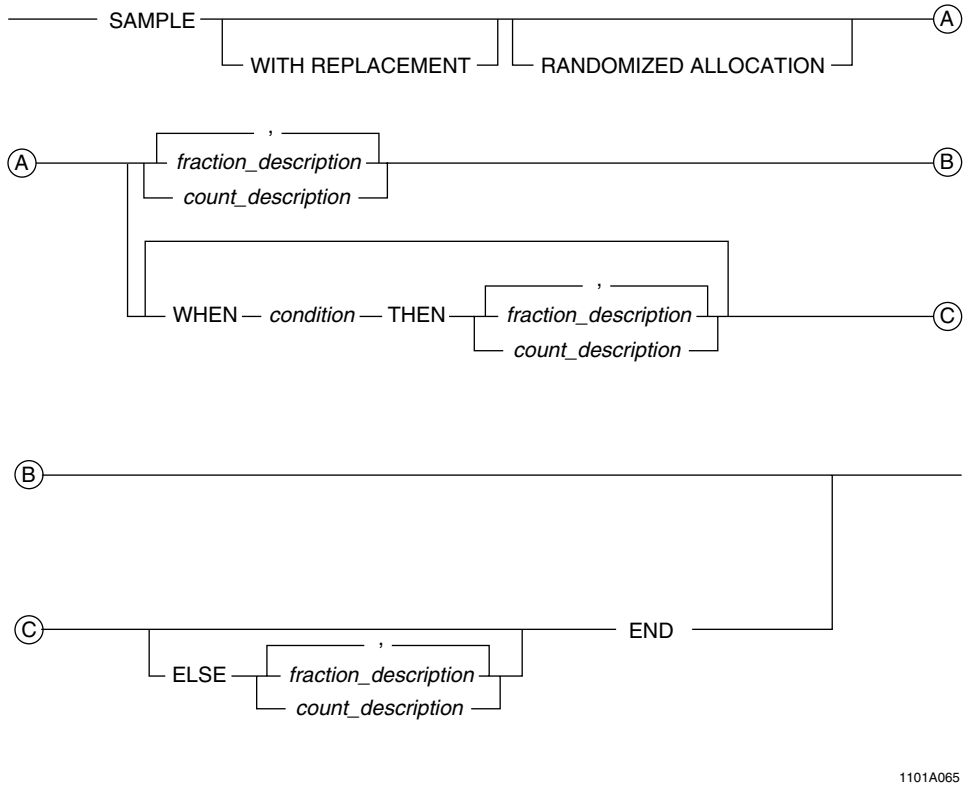
Item	Profit	Percentile
Carrot-flavored ice cream	10.79	99
Low fat carrot-flavored ice cream	- 100.55	99
Low fat sugar-free carrot-flavored ice cream	- 1 110.67	99
Regular broccoli-flavored ice cream	- 2 913.88	99
Low fat broccoli-flavored ice cream	- 4 492.12	99

SAMPLE Clause

Purpose

Reduces the number of rows to be considered for further processing by returning one or more samples of rows specified either as a list of fractions of the total number of rows or as a list of numbers of rows from the SELECT query.

Syntax



where:

Syntax element ...	Specifies ...
WITH REPLACEMENT	whether sampling is done by returning each sampled row to the table for possible redundant sampling or by withholding sampled rows from resampling. If you specify WITH REPLACEMENT, then it is possible to request more samples than there are rows in the table. Sampling without replacement is the default. You select it implicitly by not specifying WITH REPLACEMENT.

Syntax element ...	Specifies ...
RANDOMIZED ALLOCATION	whether rows are sampled randomly across AMPS (RANDOMIZED ALLOCATION) or proportionate to the number of qualified rows per AMP (proportional allocation). The proportional allocation option does not provide a simple random sample of the entire population. It provides a random sample stratified by AMPs, but it is much faster, especially for very large samples. Proportional is the default. You select it implicitly by not specifying RANDOMIZED ALLOCATION.
fraction_description	any set of unsigned floating point constant numbers in the closed interval (0,1) that specifies the percentage of rows to be sampled for a true search condition. This is a comma-separated list of fractions, the sum of which must not exceed 1. The value set specifies the percentage of the homogeneous subgroup defined by *search_condition* to be sampled for the report. No more than 16 samples can be requested per fraction description.
count_description	a set of positive integer constants that specifies the number of rows to be sampled for a true search condition. A warning is returned if there are not enough rows in the result to satisfy the sampling request completely. No more than 16 samples can be requested per count description.
WHEN	to test a set of conditions for truth.
search_condition	an evaluation predicate that defines each homogeneous subgroup in the sample set.
THEN	to apply the specified sampling function description or count description to the sample.
ELSE	to apply the specified sampling function description or count description to the sample if none of the WHEN *condition* predicates evaluates to true.
END	the termination of the WHEN … THEN … ELSE clause.

ANSI Compliance

The SAMPLE clause is a Teradata extension to the ANSI SQL-99 standard.

Usage Notes

- SAMPLE operates on the evaluated output of the table expression, which can include a WHERE clause and GROUP BY, HAVING, or QUALIFY clauses, sampling the result according to user specification.
- You can specify sampling either with or without replacement.
- You can specify sample allocation as either randomized or proportional.
- You can use the SAMPLEID expression to identify the samples to which the rows belong. See "SAMPLEID Expression" on page 1-75.

Rules

The following rules apply to SAMPLE.

- If a *fraction_description* results in no rows being returned, a warning is generated.
- If a *count_description* cannot be completely satisfied, a warning is generated and the sample size is reduced to the number of remaining rows.
- No more than 16 samples can be requested per fraction description or count description.
- A sampling request cannot be repeated. The identical sampling query run twice against the same data will report different rows in the result.
- Sampling can be used in a derived table, view, or INSERT … SELECT to reduce the number of rows to be considered for further computation.
- You cannot use a SAMPLE clause in a subquery.

Simple Random Sampling

Simple random sampling is a procedure in which every possible set of the requested size has an equal probability of being selected.

Stratified Random Sampling

Stratified random sampling, sometimes called proportional or quota random sampling, is a sampling method that divides a heterogeneous population of interest into homogeneous subgroups, or strata, and then takes a random sample from each of those subgroups.

The result of this homogeneous stratification of the population is that stratified random sampling represents not only the overall population, but also key subgroups. For example, a retail application might divide a customer population into subgroups composed of customers who pay for their purchases with cash, those who pay by check, and those who buy on credit.

Sampling With or Without Replacement

The WITH REPLACEMENT option specifies that sampling is to be done with replacement. The default is sampling *without* replacement. Sampling without replacement is assumed implicitly if you do not specify WITH REPLACEMENT explicitly.

When sampling with replacement, a sampled row, once sampled, is returned to the sampling pool. As a result, a row might be sampled multiple times. Because of this, it is possible to sample more rows than the number of rows in the input. When multiple samples are requested, all the samples are from the whole population and therefore not mutually exclusive.

Sampling without replacement is analogous to selecting rows from a SET table in that each row sampled is unique, and once a row is sampled, is not returned to the sampling pool. As a result, requesting a number of samples greater than the cardinality of the table returns an error or warning. Whenever multiple samples are requested, they are mutually exclusive.

The magnitude of the difference of the results obtained by these two methods varies with the size of the sample relative to the population. The smaller the sample relative to the population, the less the difference in the results of sampling with or without replacement.

Randomized and Proportional Row Allocation

Randomized allocation means that the requested rows are allocated across the AMPs by simulating random sampling. This is a slow process, especially for large sample sizes, but it provides a simple random sample for the system as a whole.

The default row allocation method is proportional. This means that the requested rows are allocated across the AMPs as a function of the number of rows on each AMP. This method is much faster than randomized allocation, especially for large sample sizes. Because proportional allocation does not include all possible sample sets, the resulting sample is not a simple random sample, but it has sufficient randomness to suit the needs of most applications.

Note that simple random sampling, meaning that each element in the population has an equal and independent probability of being sampled, is employed for each AMP in the system regardless of the specified allocation method.

One way to decide on the appropriate allocation method for your application is to determine whether it is acceptable to stratify the sampling input across the AMPs to achieve the corresponding performance gain, or whether you need to consider the table as a whole.

The SAMPLEID value is simply 1, 2, 3, … *n* across *n* specified samples regardless of stratification. That is, for the following SAMPLE clause,

```
SAMPLE WHEN state = 'CA' THEN 0.3, 0.2 ELSE 0.5, 0.2
```

the SAMPLEID correspondence would be: 1 2 3 4

Example 1: Using *fraction_description*

Suppose you want to generate three mutually exclusive sample sets of a customer table for a neural net analysis. The desired percentages are as follows:

- Train, 60%
- Test, 25%
- Validate, 15%

Note that the sum does not exceed 100%.

A SELECT statement to generate the desired result looks like the following:

```
SELECT customer_id, age, income, marital_status, SAMPLEID
FROM customer_table
SAMPLE 0.6, 0.25, 0.15;
```

The result might look something like the following table.

customer_id	age	income	marital_status	SAMPLEID
1362549	17	0	1	1
1362650	21	17 804	2	1
1362605	34	16 957	2	1
1362672	50	16 319	2	3
1362486	76	10 701	3	1
1362500	40	56 708	1	3
1362489	35	55 888	3	2
1362498	60	9 849	2	1
1362551	27	23 085	1	1
1362503	18	5 787	1	2

Sample 1 is the training group, Sample 2 is the test group, and Sample 3 is the validation group.

Example 2: Using *count_description*

Suppose you want to see if your customers are in at least 100 cities. The SELECT statement to do that is the following:

```
SELECT COUNT (DISTINCT city)
FROM (SELECT city FROM customer_table
SAMPLE 1000) TEMP;
```

If customer_table is large, the SAMPLE 1000 clause would not require a full scan of the table and the sort for DISTINCT would only handle 1000 rows.

A 1 000 row sample is more than 95 percent accurate for estimating if the number of distinct values is greater than 100.

If you were to make a similar query without including the SAMPLE clause, the query would first have to sort the large customer_table before performing the DISTINCT. For example.

```
SELECT COUNT (DISTINCT city)
FROM customer_table;
```

Examples Using Stratified Sampling

The following simple table provides the basis for the for the examples that follow.

```
SELECT *
FROM stores;

    storeid  city             state
----------  ---------------  -----
          2  Green Bay        WI
          7  San Diego        CA
          5  San Jose         CA
          8  Los Angeles      CA
          3  Madison          WI
          1  Racine           WI
          6  San Francisco    CA
          4  Milwaukee        WI
```

Example 3

The following query uses proportional allocation by default to sample, without replacement, 25 percent of the rows for WI and 50 percent of the rows for CA.

```
SELECT city, state, sampleid
FROM stores
SAMPLE WHEN state = 'WI' THEN 0.25
       WHEN state = 'CA' THEN 0.5
       END
ORDER BY 3;
```

```
city             state     SampleId
---------------  -----  -----------
Milwaukee        WI               1
San Diego        CA               2
San Jose         CA               2
```

Example 4

The following query uses proportional allocation by default with replacement to sample two non-mutually exclusive samples of 3 and 1 rows, respectively, for WI and two non-mutually exclusive samples of 2 rows each for CA.

```
SELECT city, state, sampleid
FROM stores
SAMPLE WITH REPLACEMENT
       WHEN state = 'WI' THEN 3, 1
       WHEN state = 'CA' THEN 2, 2
       END
ORDER BY 3;
```

```
city             state     SampleId
---------------  -----  -----------
Green Bay        WI               1
Madison          WI               1
Madison          WI               1
Racine           WI               2
San Diego        CA               3
San Jose         CA               3
San Diego        CA               4
San Jose         CA               4
```

Example 5

The following query uses randomized allocation without replacement to sample two mutually exclusive samples of 25% and 50%, respectively, of the rows from WI and two mutually exclusive samples of 25% each for CA.

```
SELECT city, state, sampleid
FROM stores
SAMPLE RANDOMIZED ALLOCATION
       WHEN state = 'WI' THEN 0.25, 0.5
       WHEN state = 'CA' THEN 0.25, 0.25
       END
ORDER BY 3;
```

```
city             state     SampleId
---------------  -----  -----------
Green Bay        WI               1
Milwaukee        WI               2
Madison          WI               2
San Diego        CA               3
San Francisco    CA               4
```

Example 6

The following query uses randomized allocation with replacement to sample 3 rows for WI, and 2 non-specific, non-WI rows.

```
SELECT city, state, sampleid
FROM stores
SAMPLE WITH REPLACEMENT RANDOMIZED ALLOCATION
       WHEN state = 'WI' THEN 3
       ELSE 2
       END
ORDER BY 3;
```

```
city             state     SampleId
---------------  -----  -----------
Racine           WI               1
Racine           WI               1
Madison          WI               1
San Diego        CA               2
San Diego        CA               2
```

Example 7

The following query samples 3 rows with replacement using randomized allocation.

```
SELECT city, state
FROM stores
SAMPLE WITH REPLACEMENT RANDOMIZED ALLOCATION 3;
```

```
city             state
---------------  -----
San Diego        CA
Madison          WI
San Jose         CA
```

SAMPLEID Expression

Purpose

Identifies the sample to which a row belongs, distinguishing rows belonging to different samples specified in the SAMPLE clause of a SELECT statement.

Syntax



ANSI Compliance

SAMPLEID is a Teradata extension to the ANSI SQL-99 standard.

Definition

The sample ID identifies the sample to which a row belongs in the left-to-right order of the SAMPLE clause specification, from 1 through *n* (where *n* is the number of samples requested in the SAMPLE clause).

Where to Specify SAMPLEID

SAMPLEID can only be specified with a SAMPLE clause and can appear either as part of a SELECT clause or in an ORDER BY clause.

Using SAMPLEID With Stratified Sampling

The SAMPLEID value for stratified sampling is simply 1, 2, 3, … *n* across *n* specified samples regardless of stratification. That is, for the following SAMPLE clause,

```
SAMPLE WHEN state = 'CA' THEN 0.3, 0.2 ELSE 0.5, 0.2
```

the SAMPLEID correspondence would be: 1 2 3 4

Example

The following SELECT statement provides three mutually exclusive sample sets (60% for sample category X, 25% for sample category Y, and 15% for sample category Z) from a customer table.

The results are returned in a table with three columns: cust_name, cust_addr, and SAMPLEID.

The integer in the SAMPLEID column identifies whether the row belongs to the 0.6 sample, the 0.25 sample, or the 0.15 sample.

The samples are identified as 1 through 3, in left-to-right order from the SAMPLE clause, so 0.6 is identified by 1, 0.25 by 2, and 0.15 by 3.

```
SELECT cust_name, cust_addr, SAMPLEID
FROM customer_table
SAMPLE 0.6, 0.25, 0.15;
```

A partial results table might look something like the following:

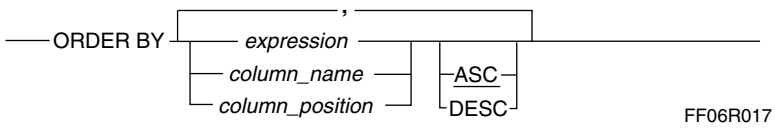
cust_name	cust_addr	SAMPLEID
Jones Pharmaceuticals	4235 Lawler Road Memphis, TN USA	1
Fong Furniture	310 East Highway 5 Hong Kong	2
Subramaniam Spice Exports	455 1/2 Gandhi Lane Hyderabad India	3
Forrester Property Management	1 West Broadway Syracuse, New York USA	1
Iwahashi Consulting	33 Korakuen Hall Tokyo Japan	1
Adler Music Publishing, Ltd.	5 East 245th Street Nashville, TN USA	
O'Brian Metals	83 Heatherington The Whithers Cobblestone-on-Treads United Kingdom	1
Irama Rice Importers	8562 Rhoma Lane Jakarta Indonesia	2
Abdelwahab Fine Egyptian Rugs	1723 Kulthum Avenue Cairo Egypt	1
Bachar Ouds	18 Rashied Diagonal Baghdad Iraq	1
…	…	…

ORDER BY Clause

Purpose

Specifies how results data are to be sorted.

Syntax



where:

Syntax Element ...	Specifies ...
ORDER BY	the order in which result rows are to be sorted.
expression	a valid SQL expression on which to sort. If the sort field is a character string, the expression used in the ORDER BY phrase can include a type modifier to force the sort to be either CASESPECIFIC or NOT CASESPECIFIC. See "Defining Case Sensitivity for Table Columns" in Chapter 4: "Character Data Types" of *Teradata RDBMS SQL Reference, Volume 3*.
column_name	the names of columns used in the ORDER BY clause in the SELECT statement. These can be ascending or descending.
column_position	the numeric position of the columns specified by the ORDER BY clause. This can be ascending or descending.
ASC	ascending sort order. The default order is ASC. If a sort option is not specified, result values are sorted in ascending order according to the client system collation sequence. If ORDER BY is not specified, rows are returned unsorted.
DESC	descending sort order.

ANSI Compliance

The ORDER BY clause is ANSI SQL-99-compliant with extensions.

Use of an ORDER BY clause in the definition of an updatable cursor is ANSI SQL-99-compliant.

Use of *expression* in ORDER BY is a Teradata extension to the ANSI SQL-99 standard.

Column Name

Each *column_name* in an ORDER BY clause is the name of a column in a table or view referenced in the SELECT expression list. The columns named do not have to match the columns in the SELECT expression list.

Column Position

The column position (*column_position*) is a positive integer that refers to the numeric position of a column in the SELECT expression list. The column position cannot be passed as a parameter; attempts to do so cause it to be interpreted as an expression, and the data to be sorted by a constant.

Use Parentheses for Required Sort Order

If a SELECT statement includes multiple parameters such as FORMAT or NAMED, or if it involves type conversion, and if an ORDER BY clause is part of the query, use parentheses to group the parameters so that they are processed in the proper sequence. Without parentheses, the parameters will be processed in left to right order.

Sorting and Default Sort Order

Before performing the sort operation that orders the rows to be returned to the requestor, Teradata creates a sort key and appends it to the rows to be sorted. If the length of this temporary data structure exceeds the system limit of 64K bytes, the operation fails.

The BYNET only looks at the first 4 096 bytes of the sort key created to sort the specified fields, so if the field the sort key is based on is greater than 4 096 bytes, the key is truncated and the data might or might not come back in the desired order.

Teradata sorts nulls *before* all other data in a column. In other words, nulls sort low. For example, suppose you want to sort the following field set: 5, 3, null, 1, 4, and 2.

These fields sort as follows:

```
null
1
2
3
4
5
```

By default, the result values of a character expression are sorted in ascending order, using the collating sequence in effect for the session.

Specifying Collation

You can specify the default collation for a user using the CREATE USER or MODIFY USER statement, or you can specify collation for the duration of a session using the SET SESSION COLLATION statement. Otherwise, collation for a session is determined by the logon client system. The direction of the sort can be controlled by including the DESC (descending) or ASC (ascending) sort option in the SQL request.

The Teradata RDBMS supports ASCII, EBCDIC, and MULTINATIONAL collating sequences. If MULTINATIONAL is in effect, your collation will be one of the European (diacritical) or Kanji sort sequences described in "International Sort Orders" on page 1-80.

The following subsections explain the results of ORDER BY as affected by whether character string expressions have the CASESPECIFIC or NOTCASESPECIFIC attribute.

Case Sensitivity

For details on defining case sensitivity for a column, see "Defining Case Sensitivity for Table Columns", CASESPECIFIC Phrase", and "UPPERCASE Phrase" in Chapter 4: "Character Data Types" of *Teradata RDBMS SQL Reference, Volume 3*.

NOT CASESPECIFIC Sorts With Japanese Character Data

On a Japanese character site in Teradata mode, if the character strings to be sorted have the NOT CASESPECIFIC attribute, only lowercase simple Latin letters (a...z) are converted to uppercase before a comparison or sorting operation is performed.

Any non-Latin single byte character, any multibyte character, and any byte indicating a transition between single byte characters and multibyte characters is excluded from this function.

If the character strings to be sorted have the CASESPECIFIC attribute, conversion to uppercase is not performed. (CASESPECIFIC is the default in ANSI mode). In this condition, simple Latin letters are considered matched only if they are the same letters and the same case.

International Sort Orders

MULTINATIONAL collation is enabled at the user level via the COLLATION option of the CREATE USER or MODIFY USER statement. If a collation is not specified, the default is HOST; that is, the standard sort ordering of the client system (EBCDIC for IBM clients, ASCII for all others).

You can override the default at the session level by issuing the SET SESSION COLLATION statement.

When MULTINATIONAL collation is in effect, the default collation sequence is determined by the collation setting installed at system start-up. Also see Chapter 4: "International Character Support" of *Teradata RDBMS SQL Reference, Volume 1*.

Each international sort sequence is defined by program constants and no check is made to ensure that collation is compatible with the character set of the current session. The choice of sequence is controlled by your database administrator. The programmed character sequences cannot be changed.

European Sort Order

The sort order uses a two-level comparison that involves the following rules:

- All lowercase letters are first mapped to their uppercase equivalents unless CASESPECIFIC is specified in the query or was defined for the column being accessed.

  In ANSI mode, the default is CASESPECIFIC and you must explicitly specify NOT CASESPECIFIC to change this.
- All diacritical forms of the same letter are given the value of the base letter; that is, Ä is given the value of A, Ö is given the value of O, and so forth.
- If two strings produce the same value, the characters are further ordered according to their sort position in their diacritical equivalence class (defined in the table of European sort orders, below).
- Unless the query specifies the DESC sort option, collation is in ascending order.

When these rules are applied, the words "abbey," "Active," and "adage" are returned in this order,

```
abbey       Active       adage
```

and the names Muller, Handl, Böckh, Mueller, Händl, Bohr, Bock, and Müller are ordered as:

```
Bock        Böckh        Bohr       Handl
Händl       Mueller      Muller     Müller
```

Equivalence classes and the ordering of diacritical characters in each class are shown in the following table of European sort orders.

The listed classes are those with characters that have diacritical forms.

A	C	E	I	N	O
a A à À á Á â Â ã Ã ä Ä	c C ç Ç	e E è È é É ê Ê ë Ë	i I ì Ì í Í î Î ï Ï	n N ñ Ñ	o O ò Ò ó Ó ô Ô õ Õ ö Ö o O
S	**U**	**Y**	**AE**	**O slash**	**A ring**
s S β	u U ù Ù ú Ú û Û (U tilde) ü Ü	y Y ÿ Ÿ	æ Æ	ø Ø	å Å

Japanese Character Sort Order Considerations

To ensure that sorting of character data is identical with that of the client, users at a Japanese character site should set their collation as follows:

- KanjiEUC or KanjiShift-JIS character set users should use the ASCII collation.
- Users under the KATAKANAEBCDIC, KANJIEBCDIC5026_0I, or KANJIEBCDIC5035_0I character set, should install either KATAKANAEBCDIC, KANJIEBCDIC5026_0I, or KANJIEBCDIC5035_0I, respectively, at start-up time, and use MULTINATIONAL collation.

  These sort orders use a one-level binary ordering.

Character data collation is handled as follows:

- Unless the CASESPECIFIC qualifier is specified in the query or was defined for the column being accessed, simple Latin letters are converted to their uppercase equivalents.
- Characters identified as single byte characters under the current character set are converted according to the collation sequence in effect for the session.

  Note that under the KanjiEUC character set, the ss3 0x8F is converted to 0xFF. This means that a user-defined KanjiEUC codeset 3 are not ordered properly with respect to other KanjiEUC code sets. The order of other KanjiEUC code sets *is* proper (that is, ordering is the same as the binary ordering on the client system).
- Characters identified as multibyte characters remain in the client encoding and are collated based on their binary values.
- Graphic data string collation is based on logical characters in the binary value on the client system.

Examples

The following examples illustrate the use of the ORDER BY clause:

Example 1

The following example produces a list of employees, sorted by years of work experience.

```
SELECT Name, JobTitle, YrsExp
FROM Employee
ORDER BY YrsExp;
```

Example 2

The following example substitutes a positive integer (3) for YrsExp in the ORDER BY clause. The 3 refers to the numeric position of YrsExp in the select expression list.

```
SELECT Name, JobTitle, YrsExp
FROM Employee
ORDER BY 3;
```

Example 3

The following example produces a list of employees, sorted by DeptNo and Salary. Note that Salary is not contained in the expression list.

```
SELECT Name, DeptNo
FROM Employee
ORDER BY DeptNo, Salary;
```

Example 4

The following example returns a list of each department number and its total population, in the order of lowest population first:

```
SELECT COUNT(Name), DeptNo
FROM Employee
GROUP BY DeptNo,
ORDER BY COUNT(Name) ;
```

Example 5

Each of the following statements can be used to list employees by department number, with the highest paid employee listed first and the lowest paid last:

```
SELECT Name, DeptNo, Salary
FROM Employee
ORDER BY 2 ASC, 3 DESC;

SELECT Name, DeptNo, Salary
FROM Employee
ORDER BY DeptNo, Salary DESC;
```

Example 6: Case Sensitivity

Consider the following statements that create and populate table T:

```
CREATE TABLE T
 (A CHAR(4) NOT CASESPECIFIC,
  B BYTEINT)
PRIMARY INDEX (A,B);

   INSERT INTO T VALUES ('AAAA', 1);
   INSERT INTO T VALUES ('aaaa', 2);
   INSERT INTO T VALUES ('BBBB', 3);
   INSERT INTO T VALUES ('bbbb', 4);
```

If the default handling of case is allowed, the following request produces the results table immediately following.

```
SELECT *
FROM T
ORDER BY A ;

A         B
----     ---
AAAA      1
aaaa      2
BBBB      3
bbbb      4
```

On the other hand, when you specify CASESPECIFIC (CS) for the query, the results are one of the results tables immediately following the example SELECT statements, depending on the collation sequence in effect.

```
SELECT *
FROM T
ORDER BY CAST(A AS CASESPECIFIC);
```

or

```
SELECT CAST(A AS CASESPECIFIC), B
FROM T
ORDER BY 1;
```

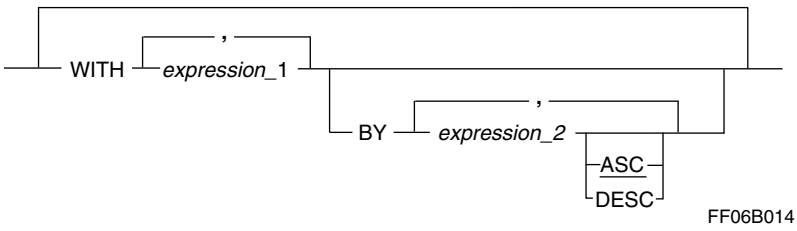
EBCDIC		ASCII		MULTINATIONAL	
A	B	A	B	A	B
----	----	----	----	----	----
aaaa	2	AAAA	1	aaaa	2
bbbb	4	BBBB	3	AAAA	1
AAAA	1	aaaa	2	bbbb	4
BBBB	3	bbbb	4	BBBB	3

WITH Clause

Purpose

Specifies summary lines and breaks (grouping conditions) that determine how selected results are returned (typically used for subtotals).

Syntax



where:

Syntax Element …	Specifies …
WITH	the keyword introducing the condition to be fulfilled by the SELECT statement.
expression_1	a summary line (such as a total) for the values in a column (expression) of the select result. *expression_1* can contain one or more aggregate and arithmetic operators that are applied to column values.
BY *expression_2*	one or more result columns (expressions) for which *expression_1* is provided. BY is valid only when used with WITH. *expression_2* can be considered as a group condition. *expression_2* can refer to an expression in the select expression list either by name or by means of a constant that specifies the numeric position of the expression in the expression list.
ASC DESC	the sort order. ASC specifies ascending order and is the default. DESC specifies descending order. If a sort option is not given, result values are sorted in ascending order according to the client system's collating sequence. If ORDER BY is not specified, rows are returned unsorted.

ANSI Compliance

The WITH clause is a Teradata extension to the ANSI SQL-99 standard.

Expressions

The following expressions can be used in a WITH clause.

- Expressions operated on by aggregate operators (for example, SUM, AVERAGE, COUNT, MIN, or MAX).

  An aggregate operator must be specified directly before each column to which the operator applies, for example, WITH SUM(Salary) or MAX(YrsExp).
- Expressions associated with the field values of an expression contained in the BY phrase, for example, WITH DeptNo, SUM(Salary) BY DeptNo.

TITLE Phrases

A TITLE phrase can be used to specify a title for any expression contained in *expression_1* and the SELECT expression list. The TITLE phrase must be enclosed by parentheses and follow the entire expression to which it applies.

Title is relevant only for FieldMode output for report generation and normally done only via BTEQ.

In the following statement, the title “Subtotal” is listed at each summary row:

```
WITH SUM(Salary)(Title 'Subtotal')
```

In the following statement, a blank title is specified:

```
WITH SUM(Salary)(Title ' ')
```

expression_2 is an expression that determines where summary lines are generated. For example, BY DeptNo specifies a summary for each value in the DeptNo column; a summary line is generated following a listing of the values for each DeptNo.

If the BY phrase is not used, the summary line applies to the entire result, as specified by the SELECT expression list.

As in the ORDER BY clause, the values of any expression specified by *expression_2* can be sorted in ascending or descending order. For example:

```
WITH SUM(Salary) BY DivNo ASC, DeptNo DESC
```

Likewise, *expression_2* can specify a constant that references an expression by its position in the SELECT expression list. For example:

```
WITH SUM(Salary) BY 2 ASC, 3 DESC
```

However, an expression that is specified in *expression_1* or *expression_2* need not appear in the SELECT expression list.

Example

The following statement can be used to generate a departmental salary report that contains a detail line for each employee and a subtotal line for each department:

```
SELECT Name, DeptNo, Salary
FROM Employee
WITH SUM(Salary) BY DeptNo;
```

The result returned is:

```
Name         DeptNo       Salary
----         ------       ----------
Peterson J     100         25,000.00
Moffit H       100         35,000.00
Jones M        100         50,000.00
Chin M         100         38,000.00
Greene W       100         32,500.00
                           ----------
              Sum(Salary) 180,500.00

Leidner P      300         34,000.00
Phan A         300         55,000.00
Russell S      300         65,000.00
                           ----------
              Sum(Salary) 154,000.00
```

Multiple WITH Clauses

More than one WITH clause can be used in a SELECT statement to specify different kinds of summaries. Each succeeding WITH clause refers to an ever broader grouping of rows.

- The BY phrase in the first-specified WITH clause defines the least important sort key.
- The BY phrase in the next-specified WITH clause defines the next-to-least important sort key.
- The final WITH clause defines the major sort key.

Example

The following statement generates a report of employee salaries ordered by department, with a summary line of total salaries for each department and a final summary line of total salaries for the entire organization:

```
SELECT Name, DeptNo, Salary
FROM Employee
WITH SUM(Salary) BY DeptNo
WITH SUM(Salary);
```

The result returned is as follows.

```
Name          DeptNo         Salary
----------  -------    -----------
Peterson J      100      25,000.00
Moffit H        100      35,000.00
Jones M         100      50,000.00
Chin M          100      38,000.00
Greene W        100      32,000.00
                         ----------
           Sum(Salary) 180,000.00


Leidner P       300      34,000.00
Phan A          300      55,000.00
Russell S       300      65,000.00
                        ----------
           Sum(Salary) 154,000.00

Smith T         500      42,000.00
Clements D      700      38,000.00
                         ----------
            Sum(Salary) 113,000.00
                         ----------
            Sum(Salary) 851,000.00
```

Combined WITH and ORDER BY Clauses

An ORDER BY clause can be specified before or after any WITH clause. Also see "ORDER BY Clause" on page 1-77.

When WITH and ORDER BY are used together, the WITH clause defines the major sort key and the ORDER BY clause defines the minor sort key. This is true regardless of the structure of the query or the number of WITH clauses.

Example

Both of the following statements use an ORDER BY clause to sort employee names in ascending order in each department group:

```
SELECT Name, DeptNo, Salary
FROM Employee
ORDER BY Name
WITH SUM(Salary) BY DeptNo
WITH SUM(Salary);

SELECT Name, DeptNo, Salary
FROM Employee
WITH SUM(Salary) BY DeptNo
WITH SUM(Salary)
ORDER BY Name;
```

The result returned is as follows.

```
Name         DeptNo        Salary
----------  -------   -----------
Chin M          100     38,000.00
Greene W        100     32,500.00
Jones M         100     50,000.00
Moffit H        100     35,000.00
Peterson J      100     25,000.00
                        ----------
            Sum(Salary) 180,500.00
   .             .         .
   .             .         .
   .             .         .
Brangle B       700     30,000.00
Clements D      700     38,000.00
Smith T         700     45,000.00
                        ----------
            Sum(Salary) 113,000.00
                        ----------
            Sum(Salary) 851,100.00
```

If any sort key column contains character data that was entered in mixed case, the results produced from WITH...BY or ORDER BY can be unexpected, depending on whether the CASESPECIFIC option was defined on the column and the collation in effect for the session (see "ORDER BY Clause" on page 1-77, and the heading "SET SESSION COLLATION" under "SET SESSION" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*).

Combining WITH and GROUP BY Clauses

Using WITH and GROUP BY clauses in the same SELECT statement can produce unintended results. Also see "GROUP BY Clause" on page 1-57.

Example

The following statement is intended to generate a report of salary totals by department with a grand total of employee salaries:

```
SELECT DeptNo, SUM(Salary)
FROM Employee
GROUP BY DeptNo
WITH SUM(Salary);
```

The result returned is as follows.

```
DeptNo     Sum(Salary)
------    -----------
   100     180,500.00
   300     143,000.00
   500     268,000.00
   600     146,600.00
   700     113,000.00
           -----------
 Sum(Sa    851,100.00
```

As would be expected, the WITH clause produces a summary line of total salaries for all departments. Note that the summary title is truncated because of the narrow width of the DeptNo column.

If the WITH clause contains an unnecessary BY phrase, then a useless summary line is generated following each department salary total as seen in the report following this example query.

```
SELECT DeptNo, SUM(Salary)
FROM Employee
GROUP BY DeptNo
WITH SUM(Salary) BY DeptNo;

DeptNo   Sum(Salary)
------   ----------
100       180,500.00
         ----------
Sum(Sa    180,500.00
          .
          .
          .
700        113,000.00
          -----------
Sum(Sa     113,000.00
```

Chapter 2:

Join Expressions

This chapter describes joins of tables and views, including the following:

- "Inner Joins" on page 2-4
  - "Ordinary Inner Join" on page 2-5
  - "Cross Join" on page 2-8
  - "Self-Join" on page 2-10
- "Outer Joins" on page 2-11
  - "Left Outer Join" on page 2-20
  - "Right Outer Join" on page 2-22
  - "Full Outer Join" on page 2-24

The chapter also presents an outer join case study ("Outer Join Case Study" on page 2-38) that demonstrates how to write correct outer joins that behave the way you expect them to.

Information on the various join algorithms used by the Teradata optimizer to perform these various joins is provided in the chapter "Join Optimizations" in *Teradata RDBMS SQL Reference, Volume 2*.

Joins

Definition

A join is an action that projects columns from two or more tables into a new virtual table. Teradata supports joins of as many as 64 tables per query.

Less formally, a join is an action that retrieves column values from more than one table.

Join Varieties

The various types of joins that SQL supports are distinguished by the conditions specified in the query that joins the tables. There are four basic types of join:

- Natural
- Theta (represented symbolically by Θ)
- Inner
- Outer

These are not mutually exclusive types: you can make natural inner and outer joins as well as Θ inner and outer joins.

Natural and Theta Joins

At the highest level of abstraction, the two basic types of join are the natural join and the Θ join.

The natural join is an equality join made over a common column set such as a primary index or primary key-foreign key relationship that is expressed in a WHERE clause equality condition. For example,

```
... WHERE a.custnum = b.custnum ...
```

The Θ join is similar except that it is made over an inequality condition. For example,

```
... WHERE a.custnum > b.custnum ...
```

SQL also supports two other special join cases: the cross join, or Cartesian product and the self-join.

For more detailed information about these join types, see the following topics:

- "Inner Joins" on page 2-4
- "Ordinary Inner Join" on page 2-5
- "Cross Join" on page 2-8
- "Self-Join" on page 2-10

Inner and Outer Joins

The inner join projects only those rows that have specific commonality between the joined tables. Because the inner join does not include rows that have no counterpart in the other table, it is sometimes said to lose that information.

For further information about inner joins, see the following topics:

- "Inner Joins" on page 2-4
- "Ordinary Inner Join" on page 2-5

The outer join is meant to provide a method for regaining the "lost" information the inner join does not report. The outer join is an extended inner join that projects not only those rows that have commonality between the joined tables, but also those rows that have no counterpart in the other relation. Depending on how you write an outer join, it can project the inner join rows plus any of the following: the nonmatching rows of the left table, the nonmatching rows of the right table, or the nonmatching rows from both. The attributes of the complementary row sets of an outer join are represented in the result by nulls.

For further information about the outer join and its various types, see the following topics:

- "Outer Joins" on page 2-11
- "Left Outer Join" on page 2-20
- "Right Outer Join" on page 2-22
- "Full Outer Join" on page 2-24

Outer joins are somewhat controversial for several reasons:

- There are some formal complications involved in deriving outer joins. For example, while it is true that the inner natural join is a projection of the inner equijoin, it is not true that the outer natural join is a projection of the outer equijoin.
- The result of an outer join can be very difficult to interpret, if only because nulls are used to represent two very different things: the standard "value unknown" meaning and the empty set, representing the attributes of the outer row set.

  See "Nulls and the Outer Join" in the chapter "Designing for Missing Information" in *Teradata RDBMS Database Design* for further information about issues with nulls and the outer join.
- It is extremely difficult to code an outer join correctly.

  For further information about coding outer joins, see the following topics:
  - "Coding ON Clauses for Outer Joins" on page 2-30
  - "Coding ON Clauses With WHERE Clauses for Outer Joins" on page 2-33
  - "Rules And Recommendations for Coding ON and WHERE Clauses for Outer Joins" on page 2-37
  - "Outer Join Case Study" beginning on page 2-38.

Inner Joins

This section describes inner joins. The inner join is more commonly referred to simply as a join.

Topics include:

- "Ordinary Inner Join" on page 2-5
- "Cross Join" on page 2-8
- "Self-Join" on page 2-10

Ordinary Inner Join

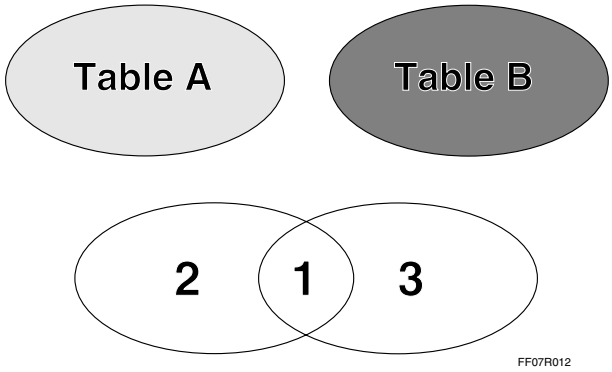
Definitions: Join and Inner Join

A join allows you to select columns and rows from 2 or more tables and views. You can join as many as 64 tables and views.

An *inner* join projects data from 2 or more tables or views that meet specific join conditions. Each source must be named and the join condition, the common relationship among the tables or views to be joined, must be specified explicitly in a WHERE clause.

Components of an Inner Join

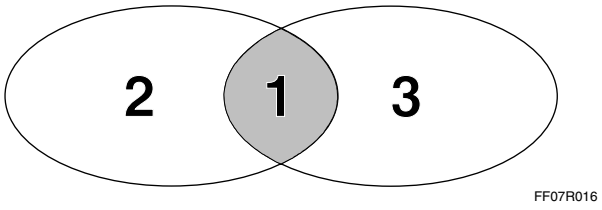
Refer to the following table abstractions and Venn diagram.



Define table_A as the row set containing sections 1 and 2. Refer to table_A as *left_table*.

Define table_B as the row set containing sections 1 and 3. Refer to table_B as *right_table*.

The inner join of table_A and table_B is section 1, as indicated by the following Venn diagram.



The Default Join

Unless explicitly declared as outer joins, all joins are inner joins by default. You can, however, specify an explicit inner join using the reserved word INNER.

The following SELECT statement is an inner join of the 2 tables table_A and table_B.

```
SELECT …
FROM table_A
INNER JOIN table_B …
```

Because the keyword sequence INNER JOIN is optional (but if you use the keyword INNER, you must use JOIN with it), the following SELECT statements are also correct examples of inner joins:

```
SELECT …
FROM table_A
JOIN table_B …

SELECT …
FROM table_A, table_B …
```

You can specify the join condition using either the ON clause or the WHERE clause for inner joins. Note that an ON clause is mandatory if the keyword JOIN is specified for an inner join.

Example

You can determine the department location of employee Marston by joining the Employee and Department tables on the column in which there are values common to both. In this case, that column is DeptNo.

```
SELECT Loc
FROM Department, Employee
WHERE Employee.Name = 'Marston A'
AND Employee.DeptNo = Department.DeptNo;
```

This query asks two questions:

- What is the number of the department in the Employee file that contains Marston?
- What is the location code for that department in the Department file?

The key to answering both questions is the DeptNo column, which has the same values in both tables and thus can be used to make the join relationship (the actual name of the column is irrelevant).

The result of this inner join is the following report.

```
Loc
ATL
```

Joins on Views Containing an Aggregate

You can join views containing an aggregate column.

In the following example, the CustFile table is joined with the CustProdSales view (which contains a SUM operation) in order to determine which companies purchased more than $10,000 worth of item 123.

```
CREATE VIEW CustProdSales (custno, pcode, sales) AS
SELECT custno, pcode, SUM(sales)
FROM SalesHist
GROUP BY custno, pcode;

SELECT company_name, sales
FROM CustProdSales a, CustFile b
WHERE a.custno = b.custno
AND a.pcode = 123
AND a.sales > 10000;
```

Cross Join

Definition

SQL supports unconstrained joins (or cross joins)—joins in which a WHERE relationship between the joined tables is not specified.

The result of an unconstrained join is also referred to as a Cartesian product (more formally, an extended Cartesian product) after the proof by the philosopher and mathematician Rene Descartes that an ordered pair of elements, in this case the concatenation of two rows, can be represented in a two dimensional plane as a single point in two-dimensional space.

The collection of all such ordered pairs formed by multiplying each element of one relation with each element of a second relation is the same result obtained by performing a cross join between two tables: the join is the product of the number of rows in each table that is joined.

Why Perform a Cross Join?

Concatenating each row of one table with every row of another table rarely produces a useful result. Before performing a cross join, you should address the following considerations:

- Why you need to execute such a wide ranging and uninformative operation
- How expensive (with respect to resource consumption) a cross join can be

For example, a cross join of two tables, A and B, each with 1,000 rows, returns a joined table of 1 million rows. Cross joins can easily abort transactions by exceeding user spool space limits.

Generally speaking, the only reason to request a cross join is to write performance benchmarks. Real world applications of cross joins are essentially nonexistent.

How to Write a Cross Join

If you want to return a Cartesian product that is not an outer join of two or more tables, you can write the following SELECT statement.

```
SELECT …
FROM table_a
CROSS JOIN table_b;
```

Because the reserved word sequence CROSS JOIN is optional, you could more concisely write:

```
SELECT …
FROM table_a,table_b;
```

For example, if table_a contains five rows and table_b contains three rows, then the Cartesian product is 3 x 5, or 15. An unconstrained join on these tables results in 15 rows being returned.

Self-Join

Definition

A self-join combines the information from two or more rows of the same table into a single result row, effectively joining the table with itself.

Example

For example, the following query asks the names of employees who have more years of experience than their managers:

```
SELECT Workers.Name, Workers.YrsExp, Workers.DeptNo,
 Managers.Name, Managers.YrsExp
FROM Employee Workers, Employee Managers
WHERE Managers.DeptNo = Workers.DeptNo
AND Managers.JobTitle IN ('Manager', 'Vice Pres')
AND Workers.YrsExp > Managers.YrsExp ;
```

The operation treats the Employee table as if it were two tables; one named Workers, the other named Managers. This is accomplished by using table name aliases in the FROM clause.

ANSI calls table aliases correlation names. They are also referred to as range variables.

Because each of these fictitious tables has the same columns (Name, YrsExp, and DeptNo) each column name must be qualified with the alias name of its table as defined in the FROM clause (for example, "Workers.DeptNo").

The WHERE clause establishes the following things:

- A key to both tables (DeptNo)
- Which employees belong in the Managers table (first AND)
- Which workers and managers should be listed in the tables (second AND)

A possible result of this self-join is as follows:

Name	YrsExp	DeptNo	Name	YrsExp
Greene W	15	100	Jones M	13
Carter J	20	500	Watson L	8
Smith T	10	700	Watson L	8
Aguilar J	11	600	Regan R	10
Leidner P	13	300	Phan A	12
Russell S	25	300	Phan A	12

Outer Joins

The outer join is an extension of the inner join that includes both rows that qualify for a simple inner join as well as a specified set of rows that do not match the join conditions expressed by the query.

Because outer joins are as complicated to code correctly as they are to interpret, this section studies them in detail, including a carefully thought out case study and series of recommendations contributed by a Teradata database administrator.

There are three types of outer joins:

- LEFT OUTER
  See "Left Outer Join" on page 2-20 for further information.
- RIGHT OUTER
  See "Right Outer Join" on page 2-22 for further information.
- FULL OUTER
  See "Full Outer Join" on page 2-24 for further information.

Definition of the Outer Join

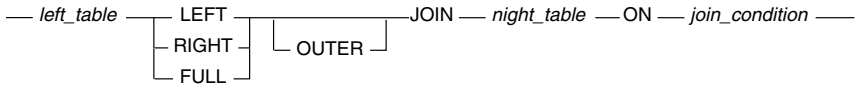
Definition: Outer Join

The *outer* join is an extension of the *inner* join.

Outer joins of two or more tables perform an inner join of those tables according to a join condition and also return rows from the left join table or the right join table (or both) that are not returned in the result of the inner join, extending the results rows with nulls for their nonmatching fields.

The outer join is loosely defined as the relational algebraic UNION ALL of its components.

Syntax



where:

Syntax element . . .	Specifies . . .
left_table	the table reference that appears to the left of the join type keywords.
right_table	the table reference that appears to the right of the join type keywords.
join_condition	the columns on which the join is made separated by the comparison operator that specifies the comparison type for the join. The join conditions of an ON clause define the rows in the outer table that take part in the match to the inner table. At *least* one join condition is required in the ON clause for each table in the outer join. You can include multiple join conditions in an ON clause by using the Boolean AND, OR, and NOT operators.

For example, consider the following SELECT statement.

```
SELECT Offerings.CourseNo, Offerings.Location, Enrollment.EmpNo
FROM Offerings
LEFT OUTER JOIN Enrollment ON Offerings.CourseNo = Employee.CourseNo;
```

where:

THIS variable . . .	CORRESPONDS to this syntax element . . .
Offerings	*left_table*
Enrollment	*right_table*
Offerings.CourseNo=Employee.CourseNo	*join_condition*

Using Views in an Outer Join

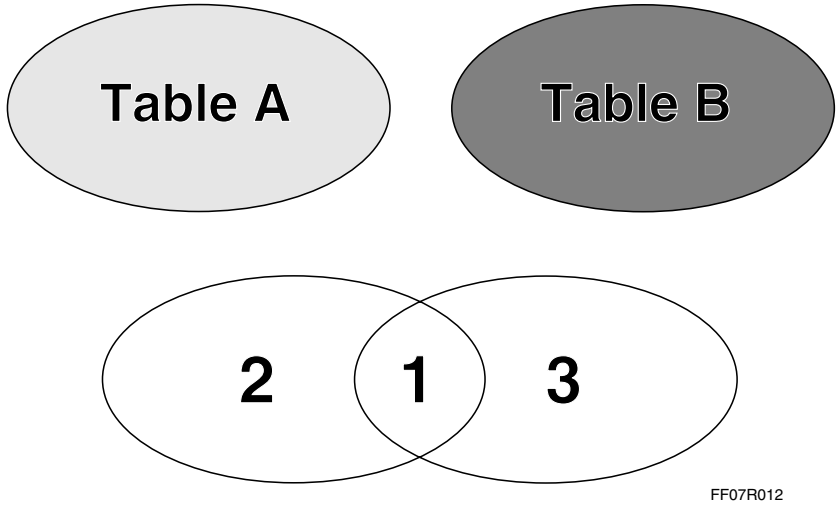
Outer join syntax can be used with views *only* in the following pairings:

- Two base tables
- A base table and a view
- Two views

No other pairings of views with base tables are permitted.

Components of an Outer Join

Refer to the following table abstractions and Venn diagram.



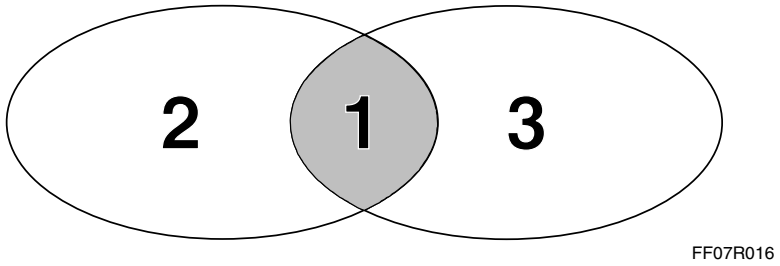
This diagram is identical to the Venn diagram used for "Components of an Inner Join" on page 2-5; however, the way the components of this diagram are combined for an outer join is very different.

Define table_A as the row set containing sections 1 and 2. Refer to table_A as *left_table*.

Define table_B as the row set containing sections 1 and 3. Refer to table_B as *right_table*.

Define the components of a 2-table outer join as follows:

This component . . .	Is defined as . . .
1	the inner join of the 2 tables as defined by an ON clause with all join conditions applied.
2	all rows from *left_table* not included in section 1 extended with nulls for each column selected from *right_table*.
3	all rows from *right_table* not included in section 1 extended with nulls for each column selected from *left_table*.



Types of Outer Join

The sections to be unioned are defined by the type of outer join:

This outer join type . . .	Is defined as . . .
LEFT	Section 1UNION ALL Section 2
RIGHT	Section 1UNION ALL Section 3
FULL	Section 1UNION ALL Section 2 UNION ALL Section 3

For each type of outer join, you can think of the operation as assembling the proper components using the UNION ALL operator.

Note the following equality.

FULL OUTER JOIN = LEFT OUTER JOIN ∪ RIGHT OUTER JOIN

Terminology

The terms “inner table” and “outer table” used frequently when talking about outer joins, are defined in the following table.

Term	Definition
Inner table	The inner table is the table that contributes only matched rows to the outer join result. For an inner join, *both* tables are inner tables.
Outer table	The outer table is the table that contributes unmatched rows to the outer join result. In this description, the word *unmatched* refers to the rows in the left or right (or both) table that are not part of the inner join rows because they have no matching columns, so they are extended with nulls in the results table.

This terminology also applies to the result of nested joins and spool files.

Inner/Outer Table Example

Consider the following nested outer join:

```
(table_A
LEFT OUTER JOIN
 (table_B
  RIGHT OUTER JOIN table_C ON join_condition
 )
ON join_condition
)
FULL OUTER JOIN table_D ON join_condition
```

What are the inner and outer tables of this outer join?

Beginning with the most deeply nested join and working outward, the following relationships apply:

1. `table_C` is an outer table with respect to `table_B`
2. `table_A` is an outer table with respect to the nested join `(table_B RIGHT OUTER JOIN table_C ON join_condition)`
3. `(table_A LEFT OUTER JOIN (table_B RIGHT OUTER JOIN table_C ON join_condition) ON join_condition` is an outer table for the full outer join
4. `table_D` is an outer table for the full outer join

Projected Column List

When you construct an outer join, the projected column list has to be chosen carefully to make the results both useful and interpretable. In general, you should project the column from the same side as the outer join. "Practical Example" on page 2-23 right outer joins on CourseNo; therefore, you needed to project out the right join column, which is Enrollment.CourseNo.

Similarly, in the full outer join in "Practical Example" on page 2-25, the example projects both CourseNo columns for this same reason.

Nulls and the Outer Join

Nulls are a fundamental component of the report produced by an outer join query.

The key feature of the outer join is that in returning rows from the outer table or tables, the report extends the rows that have no matching values with nulls, as if these unknown "values" came from the corresponding table.

Suppose, for example, you wish to list courses offered by customer education for which employees have registered and also to include in that list those courses for which no one signed up.

The practical outer join examples in "Left Outer Join" on page 2-20, "Right Outer Join" on page 2-22, and "Full Outer Join" on page 2-24 use these three tables.

- Offerings shows customer education courses currently being offered and their location. With respect to courses being offered, you can think of Offerings as a subset of Courses.

Offerings		
CourseNo	**Beginning Dates**	**Location**
C100	01/05/1999	El Segundo
C200	07/02/1999	Dayton
C400	10/07/1999	El Segundo

- Enrollment shows employees who have registered for courses, some of which may not be offered.

Enrollment	
EmpNo	**CourseNo**
236	C100
236	C300

- Courses lists all courses developed by customer education, some of which are not currently being offered (for example, C300).

Courses	
CourseNo	**Name**
C100	Recovery Planning
C200	Software Architecture
C300	Introduction to Teradata
C400	Introduction to Java Programming

Example

The following SELECT statement is an example of a left outer join of two tables, table_A and table_B:

```
SELECT column_expression
FROM table_A
LEFT OUTER JOIN table_B ON join_conditions
WHERE join_condition_exclusions;
```

The left table in the above example means the table specified to the left of the keywords OUTER JOIN in the FROM clause. The right table means the table specified to the right of the keywords OUTER JOIN.

The keyword LEFT indicates the source of the rows not returned in the result of the inner join. These are called nonmatching rows.

In addition to performing an inner join of two or more tables according to a join condition, a left outer join, as in the example above, returns nonmatching rows from its left table (table_A) and extends them with nulls.

A right outer join returns nonmatching rows from its right table and extends them with nulls.

A full outer join returns nonmatching rows from both its tables and extends them with nulls.

The reserved word OUTER is optional so that the above SELECT statement could also be written as follows:

```
SELECT...
FROM table_A
LEFT JOIN table_B ON join_condition;
```

Review of Relevant Relational Algebra for the Outer Join

Basic SQL Query

The relational algebra and the relational calculus (more formally, the first order predicate calculus) are two different, but precisely equivalent formal languages for manipulating relations. They differ in that the algebra is procedural, lending itself to internal representations of queries that can be manipulated by query optimizers and database managers, while the calculus is nonprocedural, lending itself to providing a foundation for user-malleable query languages.

The basic query SELECT-FROM-WHERE, which is the SQL dialect of the generalized nonprocedural relational calculus for a query, can be restated in terms of relational algebra as follows.

```
projection ( restriction ( product ) )
```

where:

Syntax element …	Specifies …
projection	the result of applying a PROJECT operator that extracts one or more attributes from one or more relations. In concrete terms, this defines an SQL SELECT … FROM. Projection selects columns from tables.
restriction	the result of applying a RESTRICT (or SELECT) operator that extracts one or more tuples from the projection. Note that the SELECT of relational algebra is different than the SELECT of SQL, which is essentially a PROJECT operator. In concrete terms, this defines an SQL WHERE clause. Restriction selects qualified rows from tables. When a join condition exists, restriction selects qualified rows from the intermediate results table produced by the join.
product	the result of applying a PRODUCT operator that builds all possible combinations of tuples, one from each specified relation. In concrete terms, this defines a join. More rigorously, the term defines an *inner* join.

For clarity of presentation, restate the original expression as projection (inner join).

Relational Algebra for the Outer Join

Now consider the outer join. Informally, the outer join merges the result of an inner join with the remaining tuples in one (or both) of the joined relations that do not share commonality with the tuples of the inner join result table.

The outer join can be restated in terms of the relational algebra developed here as follows:

```
projection ( inner_join ∪ extension )
```

where:

Syntax element . . .	Specifies . . .
projection	the result of applying a PROJECT operator that extracts one or more attributes from one or more relations.
inner_join	the result of applying the RESTRICT (or SELECT) operator to the PRODUCT of the relations in the projection.
∪	the UNION ALL operator.
extension	the complement of the result of applying the RESTRICT operator to the PRODUCT of the relations in the projection. Depending on how the query is stated, extension can refer either to the excluded tuples from the left table, the right table, or both. These are, respectively, left, right, and full outer joins.

Left Outer Join

Definition

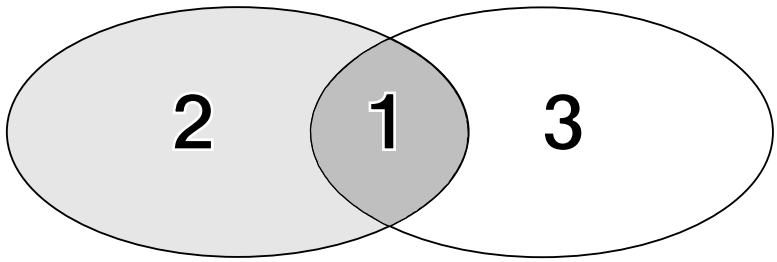
When you perform a left outer join on the Offerings and Enrollment tables, the rows from the left table that are not returned in the result of the inner join of these two tables *are* returned in the outer join result and extended with nulls.

Inner/Outer Table Example

The following example uses the explicit table names *inner_table* and *outer_table* (see "Terminology" on page 2-15) to indicate how these terms relate to the way a simple left outer join is constructed in the FROM clause of a SELECT statement.

The example demonstrates the semantics of inner and outer table references for a left outer join.

```
outer_table LEFT OUTER JOIN inner_table
```



FF07R013

Section 1 represents the inner join (intersection) of *outer_table* and *inner_table*. Section 2 represents the unmatched rows from the outer table.

The outer join result contains the matching rows from Sections 2 and 3, indicated in the diagram as Section 1, plus *un*matched rows from Section 2, noted in the graphic by the more lightly shaded component of the Venn diagram.

In terms of the algebra of sets, the result is

```
(Table_A ∩ Table_B) + (Table_A - Table_B)
```

where:

`Table_A ∩ Table_B` is the set of matched rows from the inner join of Table_A and Table_B.

`Table_A - Table_B` is the set of unmatched rows from Table_A.

Practical Example

The following SELECT statement yields the results in the table that follows.

```
SELECT Offerings.CourseNo, Offerings.Location, Enrollment.EmpNo
FROM Offerings
LEFT OUTER JOIN Enrollment ON Offerings.CourseNo = Enrollment.CourseNo;
```

O.CourseNo	O.Location	E.EmpNo
C100	El Segundo	236
C100	El Segundo	668
C200	Dayton	?
C400	El Segundo	?

These results show that course C100 has two employees enrolled in it and that course C200 and C400 have no employees enrolled in them. In this case, the nulls returned by the outer join of the Offerings and Enrollment table provide meaningful information.

The use of the keyword OUTER in the FROM clause is optional so the above SELECT statement could just as well have been written like the following SQL code:

```
SELECT Offerings.CourseNo, Offerings.Location, Enrollment.EmpNo
FROM Offerings
LEFT JOIN Enrollment ON Offerings.CourseNo = Employee.CourseNo;
```

Right Outer Join

Definition

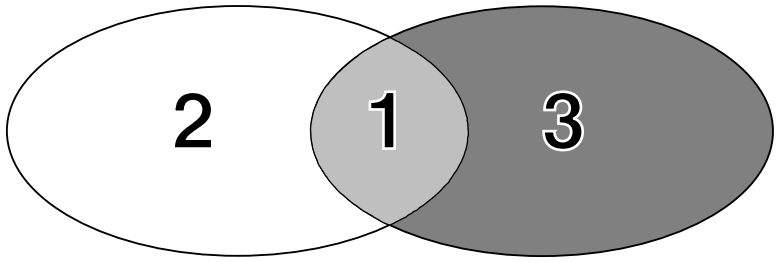
In a right outer join, the rows from the right table that are returned in the result of the inner join are returned in the outer join result and extended with nulls.

Inner/Outer Table Example

The following example uses the explicit table names *inner_table* and *outer_table* (see "Terminology" on page 2-15) to indicate how these terms relate to the way a simple right outer join is constructed in the FROM clause of a SELECT statement.

The example demonstrates the semantics of inner and outer table references for a right outer join.

```
inner_table RIGHT OUTER JOIN outer_table
```



FF07R014

Section 1 represents the inner join (intersection) of *outer_table* and *inner_table*. Section 3 represents the unmatched rows from the outer table.

The outer join result contains the matching rows from Sections 2 and 3, indicated in the diagram as Section 1, plus unmatched rows from Section 3, noted in the graphic by the more darkly shaded component of the Venn diagram.

In terms of the algebra of sets, the result is

```
(Table_A ∩ Table_B) + (Table_B - Table_A)
```

where:

`Table_A ∩ Table_B` is the set of matched rows from the inner join of Table_A and Table_B.

`Table_B - Table_A` is the set of unmatched rows from Table_B.

Practical Example

When you perform a right outer join on the Offerings and Enrollment tables, the rows from the right table that are not returned in the result of the inner join are returned in the outer join result and extended with nulls.

The following SELECT statement yields the results in the following table:

```
SELECT Offerings.CourseNo, Offerings.Location, Enrollment.EmpNo
FROM Offerings
RIGHT OUTER JOIN Enrollment ON Offerings.CourseNo = Enrollment.CourseNo;
```

O.CourseNo	O.Location	E.EmpNo
C100	El Segundo	236
C100	El Segundo	668
?	?	236

These results show that course C100 has two employees enrolled in it and that employee 236 has not enrolled in another class. But in this case the nulls returned by the right outer join of the Offerings and Enrollment tables are deceptive, because we know by inspection of the Enrollment table that employee 236 has enrolled for course C300. We also know by inspection of the Offerings table that course C300 is not currently being offered.

To get the more useful results you probably want, write the following right outer join:

```
SELECT Enrollment.CourseNo, Offerings.Location, Enrollment.EmpNo
FROM Offerings
RIGHT OUTER JOIN Enrollment ON Offerings.CourseNo = Enrollment.CourseNo;
```

Instead of the row (?, ?, 236), this query returns the row (C300, ?, 236).

Full Outer Join

Definition

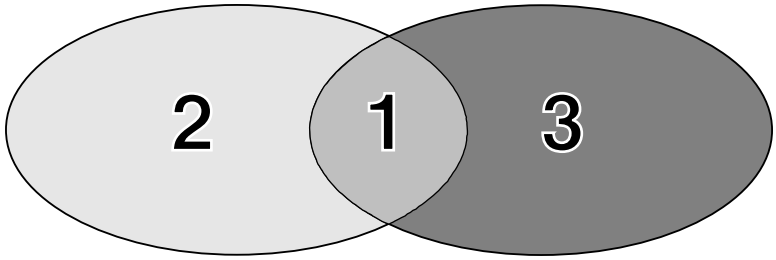
In a full outer join, rows from both tables that have not been returned in the result of the inner join are returned in the outer join result and extended with nulls.

Inner/Outer Table Example

The following example uses the explicit table names inner table and outer table (see "Terminology" on page 2-15) to indicate how these terms relate to the way a simple full outer join is constructed in the FROM clause of a SELECT statement.

The example demonstrates the semantics of inner and outer table references for a full outer join.

```
outer_table_1 FULL OUTER JOIN outer_table_2
```



FF07R015

Section 1 represents the inner join (intersection) of *outer_table_1* and *outer_table_2*. Sections 2 and 3 represent the unmatched rows from both outer tables.

The outer join result contains matched rows from the inner join of table_A with table_B plus the unmatched rows from table_A *and* table_B.

In terms of the algebra of sets, the result is

```
(Table_A ∩ Table_B) + (Table_A - Table_B) + (Table_B - Table_A)
```

where:

(`Table_A ∩ Table_B`) is the set of matched rows from the inner join of Table_A and Table_B.

(`Table_A - Table_B`) is the set of unmatched rows from Table_A.

(`Table_B - Table_A`) is the set of unmatched rows from Table_B.

Practical Example

Suppose you wish to find out if the current course offering by customer education satisfied the employee requests for courses, and include the courses offered and their locations, as well as the classes requested (but not offered) and the employees who requested them.

The example uses the same two tables as the prior example, Offerings and Enrollment.

Offerings

CourseNo	Beginning Dates	Location
C100	01/05/94	El Segundo
C200	07/02/94	Dayton
C400	10/07/94	El Segundo

Enrollment

EmpNo	CourseNo
236	C100
236	C300
668	C100

Performing a full outer join on these two tables results in the rows from both tables not returned in the result of the inner join being returned in the outer join and extended with nulls.

The following SELECT statement yields the desired results, as indicated in the results table that follows:

```
SELECT Offerings.CourseNo, Offerings.Location, Enrollment.CourseNo,
 Enrollment.EmpNo,
FROM Offerings
FULL OUTER JOIN Enrollment ON Offerings.CourseNo = Enrollment.CourseNo;
```

O.CourseNo	O.Location	E.CourseNo	E.EmpNo
C100	El Segundo	C100	236
C100	El Segundo	C100	668
C200	Dayton	?	?
C400	El Segundo	?	?
?	?	C300	236

These results show that course C100 has two employees enrolled and that courses C200 and C400 have no employees enrolled.

In this case, the nulls returned by the outer join of the Offerings and ENROLLMENT tables are meaningful information.

You can see that course C300, for which employee 236 registered, is not among those courses offered by customer education, nor does it have a location. Notice the null in the last row of the O.CourseNo and the O.Location columns.

Multitable Joins

Introduction

This topic describes some of the operations that are unique to joins of more than two tables.

Definition: Temporary Derived Table

When joining three or more tables, the join process creates an intermediate file called a temporary derived table that is defined only for the duration of the SELECT operation.

Such a temporary table is also called a joined table. Teradata refers to this kind of temporary table as a spool file.

Consider the following example:

```
table_R
LEFT OUTER JOIN table_S ON join_condition
RIGHT OUTER JOIN table_T ON join_condition
```

when table_R is left outer joined to table_S according to the first join condition, a derived table is created. It is that derived table, not a real table, that is then right outer joined (as a new derived left table) to table_T according to the second join condition.

ON Clause Evaluation Order

The result of an inner join of two tables is not changed if rows from table_A are joined to rows from table_B, or if rows from Table_B are joined to rows from table_A. In terms of set algebra, inner joins are both commutative and associative. This is true no matter how many tables are (inner) joined.

Because inner joins are both commutative and associative, the Optimizer can select the best join order for inner joins arbitrarily and the end result is always identical.

Outer joins, on the other hand, are rarely either commutative or associative. The Optimizer cannot select an arbitrary best join order for outer joins because neither commutativity nor associativity can be assumed, nor does it have any way to know what specific result you intended to produce with the query you presented to it.

If you wish to outer join three tables, then you must specify their join order explicitly by placing the ON clause in an appropriate position within the FROM clause to ensure that the join is evaluated correctly.

The rules the Optimizer follows to generate join orders for outer joins are these.

- The first ON clause in the query (reading from left to right) is evaluated first.
- Any ON clause applies to its immediately preceding join operation.

Example 1

Note that the two following outer joins evaluate differently because of the order in which their ON clauses are expressed.

Example 1a

Consider the following outer join.

```
table_R
LEFT OUTER JOIN table_S ON join_condition
RIGHT OUTER JOIN table_T ON join_condition
```

This statement is evaluated according to the steps in the following table:

Stage	Process
1	table_R is left outer joined to table_S according to the first join condition.
2	The derived table (the table resulting from the first join operation, a newly derived “left” table with respect to what follows) is right outer joined to table T according to the next join condition.

Using parentheses to indicate order of evaluation, you could rewrite the example in this way:

```
(
table_R
LEFT OUTER JOIN table_S ON join_condition
)
RIGHT OUTER JOIN table_T ON join_condition
```

You can use parentheses to write a SELECT operation that performs an outer join on three or more tables.

Example 1b

The next example places the ON clause differently, resulting in an entirely different result.

```
table_R
LEFT OUTER JOIN table_S
  RIGHT JOIN table_T ON join_condition
ON join_condition
```

This statement is evaluated according to the steps in the following table:

Stage	Process
1	Table S is right outer joined to table T according to the first join condition.
2	The derived table (a newly derived “right” table) is left outer joined to table R according to the next join condition.

Using parentheses to indicate order of evaluation, you could rewrite the example as in the following.

```
table_R
LEFT OUTER JOIN
(
 table_S
 RIGHT JOIN table_T ON join_condition
)
ON join_condition
```

Example 2

Suppose you add a fourth table, table_U to the example, as in the following join operation:

```
table_R
LEFT OUTER JOIN table_S ON join_condition
JOIN table_T
LEFT OUTER JOIN table_U ON join_condition
ON join_condition
```

This statement is evaluated according to the steps in the following table:

Stage	Process
1	table_R is left outer joined to table_S according to the first join condition.
2	table_T is then left outer joined to table_U according to the second join condition.
3	The two derived tables are inner joined according to the third join condition.

Using parentheses to indicate order of evaluation, you could rewrite the example as follows:

```
(
 table_R
 LEFT OUTER JOIN table_S ON join_condition
)
JOIN
(
 table_T
 LEFT OUTER JOIN table_U ON join_condition
)
ON join_condition
```

Notice that the join condition specified in the ON clause for the inner join (JOIN in the example) is separated from that operation by the join condition that specifies how the left outer join of table_T and table_U is to be performed.

Coding ON Clauses for Outer Joins

Introduction

Different join conditions in a FROM clause can yield very different results.

For example, suppose you wish to find out, using the three tables Courses, Offerings, and Enrollment, whether the current course offerings satisfy the employee requests for courses, including the following information in the query.

- Courses offered
- Course locations
- Courses requested but not offered
- Employees who requested courses

Example 1

In the SELECT statement that follows, the Offerings table is full outer joined to the Enrollment table according to the first ON condition (`Offerings.CourseNo = Enrollment.CourseNo`).

```
SELECT Courses.CourseNo, Offerings.CourseNo, Offerings.Location,
 Enrollment.CourseNo, Enrollment.EmpNo
FROM Offerings
FULL OUTER JOIN Enrollment ON Offerings.CourseNo = Enrollment.CourseNO
RIGHT OUTER JOIN Courses ON Courses.CourseNo = Offerings.CourseNo;
```

The result of this intermediate join is shown in the following table:

O.CourseNo	E.CourseNo	E.EmpNo
C100	C100	236
C100	C100	668
C200	?	?
C400	?	?
?	C300	236

This intermediate derived table is next right outer joined to the Courses table, according to the second join condition (`Courses.CourseNo = Offerings.CourseNo`).

The final results are shown in the following table.

C.CourseNo	O.CourseNo	E.CourseNo	E.EmpNo
C100	C100	C100	236
C100	C100	C100	668
C200	C200	?	?
C400	C400	?	?
C300	?	?	?

Example 2

If the same SELECT statement is written with a different second join condition, as illustrated in the next SELECT statement example, then the Offerings table is full outer joined to the Enrollment table, according to the first JOIN condition (`Offerings.CourseNo = Enrollment.CourseNo`)

```
SELECT Courses.CourseNo, Offerings.CourseNo, Enrollment.CourseNo,
 Enrollment.EmpNo
FROM Offerings
FULL OUTER JOIN Enrollment ON Offerings.CourseNo = Enrollment.CourseNO
RIGHT OUTER JOIN Courses ON Courses.CourseNo = Enrollment.CourseNo;
```

The result of this intermediate join is shown in the following table.

O.CourseNo	E.CourseNo	E.EmpNo
C100	C100	236
C100	C100	668
C200	?	?
C400	?	?
?	C300	236

This intermediate derived table is next right outer joined to the Courses table, according to the second join condition (`Courses.CourseNo = Enrollment.CourseNo`).

The final results are shown in the following table.

C.CourseNo	O.CourseNo	E.CourseNo	E.EmpNo
C100	C100	C100	236
C100	C100	C100	668
C300	?	C300	236
C200	?	?	?
C400	?	?	?

Comparing the ON Clause Join Conditions of Examples 1 and 2

Although the following second join conditions might intuitively appear to be equivalent, the results of a SELECT operation are different using the two different join conditions.

Join Condition	Example
`Courses.CourseNo = Offerings.CourseNo`	"Example 1" on page 2-30
`Courses.CourseNo = Enrollment.CourseNo`	"Example 2" on page 2-31

Thus, using `Courses.CourseNo = Offerings.CourseNo` as the second join condition, you do not see in the results of the outer join, for example, that employee 236 registered for Course C300.

But if you use `Courses.CourseNo = Enrollment.CourseNo` as the second join condition, you do not see in the results of the outer join, for example, that Courses C200 and C400 are, in fact, offered.

Further Information

For information on using WHERE clauses with ON clauses, see "Coding ON Clauses With WHERE Clauses for Outer Joins" on page 2-33.

For guidelines on using ON and WHERE clauses with outer joins, see "Rules And Recommendations for Coding ON and WHERE Clauses for Outer Joins" on page 2-37.

Coding ON Clauses With WHERE Clauses for Outer Joins

Introduction

Although an ON clause is required for each operation in an outer join in a FROM clause, an outer join can itself also include a WHERE clause.

Any restriction in the WHERE clause is applied only to the table that is the final result of the outer join. The WHERE clause does not define the join condition of the outer join.

Example 1

Consider the following SELECT statement.

```
SELECT Offerings.CourseNo, Enrollment.EmpNo
FROM Offerings
LEFT JOIN Enrollment ON Offerings.CourseNo = Enrollment.CourseNo
WHERE Location = 'El Segundo';
```

This query yields the intermediate derived results in the following table.

O.CourseNo	E.EmpNo
C100	236
C100	668
C200	?
C400	?

The query next applies the location restriction. After the restriction is applied (course C200 is given in Dayton), the results are as seen in the following table.

O.CourseNo	E.EmpNo
C100	236
C100	668
C400	?

Example 2

Suppose you were to put the location restriction in the ON clause, as in the following SELECT statement.

```
SELECT Offerings.CourseNo, Enrollment.EmpNo
FROM Offerings
LEFT OUTER JOIN Enrollment ON (Location = 'El Segundo')
AND (Offerings.CourseNo = Enrollment.CourseNo);
```

Such a restriction is not, strictly speaking, a join condition. Instead, it is a search condition.

The location restriction is applied as part of the inner join (of the left outer join) rather than *after* the left outer join has been performed.

When formed this way, the query produces confusing results, as the following table illustrates.

O.CourseNo	E.EmpNo
C100	236
C100	668
C200	?
C400	?

Although it was applied as part of the inner join (of the left outer join), the location restriction did not eliminate course C200, given in Dayton, from being returned. C200 was returned as part of the rows that were not returned as a result of the inner join.

Use Join Conditions in ON Clauses, Not Search Conditions

To avoid obtaining unexpected results, you should generally use only ON clauses that reference columns from two tables that are to be joined—that is, use join conditions in ON clauses, and not search conditions. Also see "A Guide to Placing Search Conditions in a Join" on page 2-36.

Example 1: WHERE Restriction Not Part of the Join Condition

Consider the following two tables, table_A and table_B.

table_A.a	table_A.b
3	1
6	6

table_B.a
3
6

The following SELECT statement yields these results. The left outer join returns the data in the table that follows.

```
SELECT *
FROM table_A
LEFT OUTER JOIN table_B ON table_A.a = table_B.a
WHERE table_A.b > 5;
```

The query yields the intermediate derived results in the following table.

table_A.a	table_A.b
3	1
6	6

table_B.a
3
6

When the WHERE restriction is applied, the final results are reported as seen in the following table.

table_A.a	table_A.b
6	6

table_B.a
6

Example 2: WHERE Restriction Part of the Join Condition

What happens if you include the WHERE restriction as part of the join condition? Consider the following SELECT statement.

```
SELECT *
FROM table_A
LEFT OUTER JOIN table_B ON (table_A.b >5)
AND (table_A.a = table_B.a);
```

The results of this query are confusing, as the following results table demonstrates.

table_A.a	table_A.b
6	6
3	1

table_B.a
6
?

Notice that the join condition, which specified the inner join of the left outer join, restricts the results to 6, 6, 6. But then the rows that were not returned as a result of the inner join are returned and extended with nulls. Thus, for our second row, you get 3, 1, null, which is not the desired result.

Using Search Conditions in ON Clauses

Some scenarios require a search condition in the ON clause. For example, to list all courses offered as well as the course requested by employee 236, you might use the following query.

```
SELECT Offerings.CourseNo, Enrollment.EmpNo
FROM Offerings
LEFT OUTER JOIN Enrollment ON Offerings.CourseNo = Enrollment.CourseNo
AND Enrollment.EMpNo = 236;
```

A Guide to Placing Search Conditions in a Join

FOR this join type . . .	Put the search condition in this clause . . .
outer	WHERE
inner	ON

In a left outer join, the outer table is the left table, and the inner table is the right table.

Further Information

For information on coding ON clauses, see “Coding ON Clauses for Outer Joins” on page 2-30.

For guidelines on using ON and WHERE clauses with outer joins, see “Rules And Recommendations for Coding ON and WHERE Clauses for Outer Joins” on page 2-37.

Rules And Recommendations for Coding ON and WHERE Clauses for Outer Joins

- One or more join conditions, or connecting terms, are required in the ON clause for each table in an outer join.

  These join conditions define the rows in the outer table that take part in the match to the inner table.
- The best practice is to use *only* join conditions in ON clauses.

  However, when a search condition is applied to the *inner* table in a WHERE clause, it should be applied in the ON clause as well.

  A search condition in the ON clause of the inner table does *not* limit the number of rows in the answer set. Instead, it defines the rows that are eligible to take part in the match to the outer table.
- An outer join can also include a WHERE clause; however, the results you get with a WHERE clause may be surprising and not obvious or even intuitive. This is explained in more detail in the case study (see "Outer Join Case Study" on page 2-38 and the topics that follow) using specific examples.

  To limit the number of qualifying rows in the outer table (and therefore the answer set), the search condition for the outer table must be in the WHERE clause. Note that the Optimizer applies the WHERE clause condition only *after* the outer join has been produced.

  If a search condition on the inner table is placed in the WHERE clause, the join is logically equivalent to an inner join, even if you explicitly specify the keywords LEFT/RIGHT/FULL OUTER JOIN in the query. The Optimizer always treats such a join as an inner join to simplify the query, rewriting it to roll the entire complex process into a single step.

Outer Join Case Study

Acknowledgement

This case study was developed by Rolf Hanusa, lead DBA at Southwestern Bell, and is used with his permission.

The text has been expanded, reformatted, and rewritten by NCR Information Engineering to accommodate the look and feel of NCR user documentation.

Purpose of the Case Study

The outer join is a powerful tool, and its output can be as difficult to interpret as it is to produce. A lack of understanding can produce unexpected, not to mention costly, undesired results. For example, you might erroneously mail promotional fliers to 17 million customers instead of the 17,000 customers that you intended to target.

This case study demonstrates some common pitfalls of the outer join, including how a carelessly worded outer join request can result in a simple inner join and how an improperly constructed outer join can return millions of rows that do not answer the business question you are trying to ask.

The study also provides guidelines to help you get a feeling for whether you are answering the business question you think you are asking with your outer joins. The study helps you to understand the complexity of outer joins by using realistic examples and explanations.

As this case study demonstrates, many results are possible, and the correct solution is not necessarily intuitive, especially in a more complex query.

Topics for the Case Study

This case study contains:

- "Case Study Examples" on page 2-39
- "Heuristics for Determining a Reasonable Answer Set" on page 2-41
- "First Attempt" on page 2-43
- "Second Attempt" on page 2-46
- "Third Attempt" on page 2-48
- "Final Attempt: The Correct Answer" on page 2-51
- "Guidelines for Getting the Desired Answer Using Outer Joins" on page 2-53

Case Study Examples

Introduction

These examples represent actual cases from a Teradata customer site. The examples are changed slightly to avoid disclosing any business critical information, but the basic syntax and counts remain accurate.

The example EXPLAIN reports are altered slightly for clarity, replacing site-specific aliases with generic database names.

Note, too, that the EXPLAIN reports do not reflect the current EXPLAIN text features. This does not detract from their value as an instructional tool.

Defining the Business Question

Before writing a complex query, you must understand the business question it is supposed to answer. The following text is a simple explanation of the business question being asked in the case study.

- What are the Customer Numbers for all the customers in the Customer table (which contains over 18 million rows)

  AND

  - Who reside in District K

    AND

  - Who have a Service_Type of ABC

    OR

  - Who have a Service_Type of XYZ

  AND
- What is their Monthly_Revenue for July 1997 (using the Revenue table, which contains over 234 million rows)

  AND
- If the Monthly_Revenue for a customer is unknown (that is, if no revenue rows are found), then report Customer Number with a null (represented by a ? character) to represent its Monthly_Revenue.

This sounds like a fairly simple task, but when you analyze your answer sets carefully, you might find them to be incorrect; in some cases, surprisingly so. As the following missteps (see "First Attempt" on page 2-43, "Second Attempt" on page 2-46, and "Third Attempt" on page 2-48) demonstrate, the results are correct, but the outer join queries that produce them are not asking the proper questions. This becomes more apparent as the examples are developed and the results analyzed.

The study presents two example queries for determining the reasonableness of the answers of this study, three missteps along the path to getting the correct answer, and a properly coded query that returns the correct answer for the original question that was asked.

Heuristics for Determining a Reasonable Answer Set

Introduction

Before you can judge the accuracy of the results returned from an outer join query, you need to estimate a reasonable guess about the likelihood that an answer set is correct.

This topic provides 2 simple methods for obtaining the approximate numbers of rows you should expect to see returned from a correctly structured outer join query.

The first example is a single table SELECT that provides the number of customer rows that you should expect to see in the result of the example outer join used for this study.

The second example is an inner join that helps to explain the remaining queries and results in the case study. It starts with the same base of customer rows but matches them with revenue rows for a particular month.

Single Table Cardinality Estimate

This query is a single table SELECT that provides the cardinality of the customer table as restricted by the conditions used for the example outer join queries.

From the report returned by this query, we know how many rows should be returned by the outer join.

```
SELECT  c.custnum
FROM sampdb.customer c
WHERE c.district = 'K'
AND  (c.service_type = 'ABC'
OR c.service_type = 'XYZ')
ORDER BY 1;
```

This query returns 18 034 rows.

Inner Join Cardinality Estimate

This query is an inner join that helps to explain the example outer join queries and their results.

The query starts with the same customer rows found in "Single Table Cardinality Estimate" on page 2-41, but then matches them with revenue rows for the month of July, 1997.

```
SELECT c.custnum, b.monthly_revenue
FROM sampdb.customer c, sampdb2.revenue b
WHERE c.custnum = b.custnum
AND  c.district = 'K'
AND  b.data_year_month = 199707
AND (c.service_type = 'ABC'
OR c.service_type = 'XYZ')
ORDER BY 1;
```

The query returns 13 010 rows.

Note that all Customer table rows are matched by a Monthly_Revenue table row.

First Attempt

Example 1: Outer Join That Is Really an Inner Join

Example 1 makes an explicit outer join request, but its result might surprise you.

```
SELECT c.custnum, b.monthly_revenue
FROM sampdb.customer c
LEFT OUTER JOIN sampdb2.revenue b ON c.custnum = b.custnum
WHERE c.district ='K'
AND b.data_date = 199707
AND (c.service_type = 'ABC'
OR c.service_type = 'XYZ')
ORDER BY 1;
```

This query returns 13 010 rows, the same as the simple inner join of "Inner Join Cardinality Estimate" on page 2-42.

Brief Analysis of the Result

Although the statement requests a LEFT OUTER JOIN, the Optimizer treats it as a simple inner join. Why?

This result occurs because all the selection criteria are in the WHERE clause, so they are logically applied only *after* the outer join processing has been completed. This means that Examples 2 and 3 are logically identical, so they produce the same result.

EXPLAIN Text for Example 1

An EXPLAIN for the query in Example 1 generates the following text.

Notice in step 2 of the EXPLAIN text that the Optimizer recognizes this query to be an *inner* join and executes it as such ("...joined using a merge join..." rather than "...*left outer* joined using a merge join..."). Therefore, it executes with the speed of an inner join.

```
1. First, we lock SAMPDB.CUSTOMER for access, and we lock
   SAMPDB2.REVENUE for access.
2. Next, we do an all-AMPs JOIN step from SAMPDB.CUSTOMER by
   way of a RowHash match scan with a condition of ("(SAMPDB.
   T1.DISTRICT = 'K') and ((SAMPDB. T1.SERVICE_TYPE= 'ABC ')
   or (SAMPDB. T1.SERVICE_ TYPE = 'XYZ '))"), which is joined
   to SAMPDB.CUSTOMER with a condition of
   ("SAMPDB2.REVENUE.DATA_YEAR_MONTH = 199707").
   SAMPDB.CUSTOMER and SAMPDB2.REVENUE are joined using a
   merge join, with a join condition of
   ("(SAMPDB.CUSTOMER.CUSTNUM = SAMPDB2. REVENUE.CUSTNUM)").
   The input table SAMPDB.CUSTOMER will not be cached in
   memory. The result goes into Spool 1, which is built locally
   on the AMPs. Then we do a SORT to order Spool 1 by the sort
   key in spool field1. The size of Spool 1 is estimated to be
   1,328,513 rows. The estimated time for this step is 6
   minutes and 2 seconds.
3. Finally, we send out an END TRANSACTION step to all AMPs
   involved in processing the request.
-> The contents of Spool 1 are sent back to the user as the
   result of statement 1. The total estimated time is 0 hours
   and 6 minutes and 2 seconds.
```

Analysis of the EXPLAIN Text

Stage	Description
1	The Lock Manager applies the appropriate ACCESS locks for the query.
2	The Optimizer selects the best join algorithm for this *inner* join (a merge join) and applies the conditions from the WHERE clause. Logically, the WHERE clause terms are applied after the terms in the ON clause of the LEFT OUTER JOIN have been applied. That would result in an intermediate step of 18 034 rows. However, the term B.DATA_YEAR_MONTH=199707 from the WHERE clause would then applied, eliminating all rows where the value for B.DATA_YEAR_MONTH is null and returning the final, incorrect, result of 13 010 rows. These steps are actually headed off by the Optimizer, which recognizes that the WHERE clause references an inner table that does not evaluate to TRUE for nulls and is therefore equivalent to an inner join. As a result, it generates a plan for a simple inner join, ignoring the request for a left outer join. You can see this in the EXPLAIN text where it says "...SAMPDB.CUSTOMER and SAMPDB2.REVENUE are joined using a merge join..." Had the plan used an outer join, the EXPLAIN text would have said "...SAMPDB.CUSTOMER and SAMPDB2.REVENUE are left outer joined using a merge join..."
3	The result of the query, 13 010 rows, is returned to the requestor. This is *not* the correct answer to our business question.

The Coding Error

The following error was made in coding the SELECT statement used in this example:

- The selection criteria for the outer table in the query (Customer) are placed in the WHERE clause rather than the ON clause, resulting in the query being processed as a simple inner join rather than the intended left outer join.

You can examine this from 2 different perspectives, both of which explain why the expected result was not returned.

1. Logical explanation. The WHERE clause appears later in the query than the ON clause. Because of this, the WHERE clause predicates are evaluated after the ON clause predicates.

   In this particular case, the result table for the ON clause has 18 034 rows. When the WHERE clause predicate b.data_year_month=199707 is applied, you should expect the ensuing result table to have fewer rows because this condition eliminates any results rows where b.data_year_month is null, and an outer join, by definition, produces results having nulls on selection predicates.

2. Physical explanation. Note that the EXPLAIN text for this query does not follow the logical explanation.

   One of the fundamental tasks of the Optimizer is to reorganize queries into an algebraic form that can be analyzed quickly. In this case, the Optimizer recognizes that a predicate referencing the inner table of an outer join does not evaluate to TRUE when a column referenced in that predicate expression contains nulls.

   Because of this, the Optimizer recognizes the requested join to be an inner, not an outer, join and so evaluates the query in that light. As a result, it generates a plan to perform an inner join (you can see this in the second step of the EXPLAIN text where it says “...are joined using a merge join...” The wording would have been “...are joined using an *outer* merge join...” if the query had been formulated correctly as an outer join.

   While this is the correct interpretation of the SELECT statement as written, the query does *not* formulate the business question correctly.

Second Attempt

Example 2: Outer Join But Does Not Return the Correct Answer

Example 2 is a correctly formed outer join, but it neither addresses the question posed nor returns the correct answer.

```
SELECT c.custnum, b.monthly_revenue
FROM sampdb.customer c
LEFT OUTER JOIN sampdb2.revenue b ON c.custnum = b.custnum
AND c.district ='K'
AND b.data_year_month = 199707
AND (c.service_type ='ABC'
OR c.service_type ='XYZ')
ORDER BY 1;
```

This query returns 17 713 502 rows.

Brief Analysis of the Result

The statement requests a left outer join without qualifying the results table with a WHERE clause. The Optimizer constructs a plan for a left outer join exactly as specified by the query. The result is a join of *every* row in the outer table (17 713 502 rows) with the qualified rows of the inner table, those having service types of either ABC or XYZ in district K in the month of July, 1997 (13 010 rows).

The cardinality of this result, 17 713 502 rows, is three orders of magnitude larger than the correct answer to the question we hope to answer.

Without a WHERE clause to qualify the result, an outer join result always contains at least 1 row for every row in the outer table.

Qualifying predicates placed in the ON clause do not eliminate rows from the result. Rather, they are treated as rows that are not matched with the inner table irrespective of whether join terms that reference both the inner and outer tables evaluate to TRUE. In other words, the selection criteria of an ON clause only define the rows to which nulls are to be appended for nonmatching rows. They do *not* restrict the result to rows matching the conditions they specify.

Note that the Optimizer does qualify the result to a degree, returning only those rows for which the predicate b.data_year_month = 199707 evaluates to TRUE in the second step (see “EXPLAIN Text for Example 2” on page 2-47).

EXPLAIN Text for Example 2

The following EXPLAIN text is reported on the query in Example 2.

```
1) First, we lock SAMPDB.CUSTOMER for access, and we lock
SAMPDB2.REVENUE for access.
2) Next, we do an all-AMPs JOIN step from SAMPDB.CUSTOMER by
way of a RowHash match scan with no residual conditions, which
is joined to SAMPDB.CUSTOMER with a condition of
("SAMPDB2.REVENUE.DATA_YEAR_MONTH = 199707"). SAMPDB.CUSTOMER
and SAMPDB2.REVENUE are left outer joined using a merge join,
with condition(s) used for nonmatching on left table
("((SAMPDB.T1.SERVICE_TYPE='ABC') or
(SAMPDB.T1.SERVICE_TYPE='XYZ')) and (SAMPDB. T1.DISTRICT =
'K')"), with a join condition of (" (SAMPDB.T1. CUSTNUM =
SAMPDB2.REVENUE.CUSTNUM)"). The input table SAMPDB.CUSTOMER
will not be cached in memory. The result goes into Spool 1,
which is built locally on the AMPs. Then we do a SORT to order
Spool 1 by the sort key in spool field1. The size of Spool 1
is estimated to be 17,713,502 rows. The estimated time for this
step is 7 minutes and 15 seconds.
3) Finally, we send out an END TRANSACTION step to all AMPs
involved in processing the request.
--> The contents of Spool 1 are sent back to the user as the
result of statement 1. The total estimated time is 0 hours and
7 minutes and 15 seconds.
```

Analysis of the EXPLAIN Text

Stage	Description
1	The Lock Manager applies the appropriate ACCESS locks for the query.
2	The Optimizer selects the best join algorithm for this left outer join (a merge join) and applies the conditions from the ON clause.
3	The result of the query, 17 713 502 rows, is returned to the requestor. This is *not* the correct answer to our business question.

The Coding Error

The following error was made in phrasing the SELECT statement used in this example:

- All selection criteria for the query were placed in the ON clause.

  Only selection criteria for the *inner* table in the outer join, those that define the nullable nonmatching rows for the outer join, should be placed in the ON clause.

  Selection criteria for the *outer* table (the Customer table in this case) in the outer join must be placed in the WHERE clause.

Third Attempt

Example 3: Outer Join That Is Really an Inner Join

Example 3 is another query where an explicit outer join is requested, but that join is logically an inner join, so the Optimizer transforms it into a simple inner join before performing the query.

```
SELECT c.custnum, b.monthly_revenue
FROM sampdb.customer c
LEFT OUTER JOIN sampdb2.revenue b ON c.custnum = b.custnum
AND c.district ='K'
AND (c.service_type = 'ABC'
OR c.service_type = 'XYZ')
WHERE b.data_year_month = 199707
ORDER BY 1;
```

This query returns 13,010 rows.

Brief Analysis of the Result

Note the similarity to Example 1 (see "EXPLAIN Text for Example 1" on page 2-44).

Again, the Optimizer treats this query as an inner join even though the statement explicitly requested an outer join. The WHERE clause on the inner table logically changes this query from an outer join to an inner join (see step 2 in "EXPLAIN Text for Example 3").

As in previous examples, the WHERE clause is logically applied *after* the outer join processing completes, removing all rows that were nulled in the process: nonmatching rows between the left and right tables. Like "First Attempt" on page 2-43, the Optimizer knows to execute this as an inner join to improve the performance of the query.

EXPLAIN Text for Example 3

The following EXPLAIN text is reported on the query in Example 3.

As you can see, this EXPLAIN text output is identical to that of the EXPLAIN text for Example 1 (see "EXPLAIN Text for Example 1" on page 2-44) and, as expected, so is the answer set.

```
1. First, we lock SAMPDB.CUSTOMER for access, and we lock
SAMPDB2.REVENUE or access.
2. Next, we do an all-AMPs JOIN step from SAMPDB.CUSTOMER by
way of a RowHash match scan with a condition of ("(SAMPDB.
T1.DISTRICT = 'K') and ((SAMPDB. T1.SERVICE_TYPE= 'ABC') or
(SAMPDB. T1.SERVICE_ TYPE='XYZ'))"), which is joined to
SAMPDB.CUSTOMER with a condition of
("SAMPDB2.REVENUE.DATA_YEAR_MONTH =199707"). SAMPDB.CUSTOMER
and SAMPDB2.REVENUE are joined using a merge join, with a join
condition of (" (SAMPDB.T1.CUSTNUM = SAMPDB2.
REVENUE.CUSTNUM)"). The input table SAMPDB.CUSTOMER will not
be cached in memory. The result goes into Spool 1, which is
built locally on the AMPs. Then we do a SORT to order Spool 1
by the sort key in spool field1. The size of Spool 1 is
estimated to be 1,328,513 rows. The estimated time for this
step is 6 minutes and 2 seconds.
3. Finally, we send out an END TRANSACTION step to all AMPs
involved in processing the request.
--> The contents of Spool 1 are sent back to the user as the
result of statement 1. The total estimated time is 0 hours and
6 minutes and 2 seconds.
```

Analysis of the EXPLAIN Text

Stage	Description
1	The Lock Manager applies the appropriate ACCESS locks for the query.
2	The Optimizer selects the best join algorithm for this *inner* join (a merge join) and applies the conditions from the WHERE clause. Logically, the WHERE clause term is applied after the terms in the ON clause of the LEFT OUTER JOIN have been applied. That would result in an intermediate step of 18 034 rows; however, the term B.DATA_YEAR_MONTH=199707 from the WHERE clause would then be applied, eliminating all rows where the value for B.DATA_YEAR_MONTH is null, and returning the final, incorrect, result of 13 010 rows. These steps are headed off by the Optimizer, which recognizes that the WHERE clause references an inner table that does not evaluate to TRUE for nulls and is therefore equivalent to an inner join. As a result, it generates a plan for a simple inner join, ignoring the request for a left outer join. You can see this in the EXPLAIN text where it says "...SAMPDB.CUSTOMER and SAMPDB2.REVENUE are joined using a merge join..." Had the plan used an outer join, the EXPLAIN text would have said "...SAMPDB.CUSTOMER and SAMPDB2.REVENUE are left outer joined using a merge join..."
3	The result of the query, 13 010 rows, is returned to the requestor. This is *not* the correct answer to our business question.

The Coding Error

The following error was made in phrasing the SELECT statement used in this example:

- The selection criterion of B.DATA_DATE = 199707 was placed in the WHERE clause rather than the ON clause, resulting in the query being processed as a simple inner join rather than the intended left outer join.

Final Attempt: The Correct Answer

Example 4: Outer Join

Finally, we have the correct answer. This example is an outer join that provides the desired answer to the original business question.

```
SELECT c.custnum, b.monthly_revenue
FROM sampdb.customer c
LEFT OUTER JOIN sampdb2.revenue b ON c.custnum = b.custnum
AND b.data_year_month = 199707
WHERE c.district ='K'
AND (c.service_type = 'ABC'
OR c.service_type = 'XYZ')
ORDER BY 1;
```

This query returns 18 034 rows.

Brief Analysis of the Result

The cardinality of this query result, 18 034 rows, reconciles with the expected number of returned rows (see "Single Table Cardinality Estimate" on page 2-41).

13 010 rows have values (non-nulls) for Monthly_Revenue.

EXPLAIN Text for Example 4

The following EXPLAIN text is reported on the query in Example 4.

```
1. First, we lock SAMPDB.CUSTOMER for access, and we lock
SAMPDB2.REVENUE for access.
2. Next, we do an all-AMPs JOIN step from SAMPDB.CUSTOMER by
way of a RowHash match scan with a condition of ("((SAMPDB.
T1.SERVICE_TYPE= 'ABC') or (SAMPDB. T1.SERVICE_ TYPE='XYZ'))
and (SAMPDB.T1. DISTRICT = 'K')"), which is joined to
SAMPDB.CUSTOMER with a condition of
("SAMPDB2.REVENUE.DATA_YEAR_MONTH = 199707"). SAMPDB.CUSTOMER
and SAMPDB2.REVENUE are left outer joined using a merge join,
with a join condition of ("(SAMPDB.T1.CUSTNUM =
SAMPDB2.REVENUE.CUSTNUM )"). The input table SAMPDB.CUSTOMER
will not be cached in memory. The result goes into Spool 1,
which is built locally on the AMPs. Then we do a SORT to order
Spool 1 by the sort key in spool field1. The size of Spool 1
is estimated to be 1,328,513 rows. The estimated time for this
step is 6 minutes and 2 seconds.
3. Finally, we send out an END TRANSACTION step to all AMPs
involved in processing the request.
--> The contents of Spool 1 are sent back to the user as the
result of statement 1. The total estimated time is 0 hours and
6 minutes and 2 seconds.
```

Analysis of the EXPLAIN Text

Stage	Description
1	The Lock Manager applies the appropriate ACCESS locks for the query.
2	The Optimizer selects the best join algorithm for this outer join (a merge join) and applies the conditions from the WHERE and ON clauses, respectively. The left (outer) table is limited by the search conditions in the WHERE clause, and the search condition in the ON clause for the right (inner) table defines the nullable nonmatching rows. The EXPLAIN text confirms that this is a true outer join ("...SAMPDB.CUSTOMER and SAMPDB2.REVENUE are left outer joined...").
3	The result of the query, 18 034 rows, is returned to the requestor. This is the correct answer to our business question.

Guidelines for Getting the Desired Answer Using Outer Joins

Introduction

As the examples in this case study show, outer joins, when used properly, provide additional information from a single query that would otherwise require multiple queries and steps to achieve. However, the proper use of outer joins requires training and experience because "common sense" reasoning does not always apply to formulating an outer join query in a manner that is not only *syntactically* correct, but also returns an *answer* that is correct for the business question posed.

Guidelines

Follow these steps to ensure that you always get the right answers to your business questions using outer joins.

1. Understand the question you are trying to answer and know the demographics of your tables.

   You should have a good idea of what the answer set should look like before you begin (see "Single Table Cardinality Estimate" on page 2-41 and "Inner Join Cardinality Estimate" on page 2-42).

2. Write the query, keeping in mind the proper placement of join and search conditions.

PLACE these conditions . . .	IN this clause . . .
join	ON
search condition predicates for inner table	ON
search condition predicates for outer table	WHERE

3. Always EXPLAIN the query before performing it.

   Look for the words *outer join* in the EXPLAIN text. If they are not there, then the Optimizer did not produce an outer join plan.

4. Perform the query and compare the result with your expectations.

IF the answer set . . .	THEN . . .
matches your expectations	it is probably correct.
does not match your expectations	check the placement of the selection criteria predicates in the ON and WHERE clauses of your outer join (see step 2 on page 2-53).

Chapter 3:

SQL Data Manipulation Language Statement Syntax

This chapter describes SQL data manipulation language (DML) statements excluding those used for declaring and manipulating cursors, multisession programming, client-server connectivity, dynamic SQL, embedded SQL only, and stored procedures only.

- ABORT
- BEGIN TRANSACTION
- CALL
- CHECKPOINT
- COMMENT (Placing Form)
- COMMIT
- DELETE
- DELETE (Basic/Searched Form)
- DELETE (Join Condition Form)
- DELETE (Positioned Form)
- ECHO
- END TRANSATION
- EXECUTE (Macro Form)
- EXPLAIN Modifier
  - EXPLAIN: Examples of Complex Queries
  - EXPLAIN and Join Processing
  - EXPLAIN and Standard Indexed Access
  - EXPLAIN and Parallel Steps
  - EXPLAIN and MERGE Conditional Steps
  - EXPLAIN and UPDATE (Upsert Form) Conditional Steps
- GROUP BY Clause
- HAVING Clause
- INCLUDE
- INSERT
- LOCKING Modifier
- MERGE
- ORDER BY Clause
- QUALIFY Clause

- ROLLBACK
- SAMPLE Clause
- SELECT
- SELECT INTO
- UPDATE (Searched Form)
- UPDATE (Positioned Form)
- UPDATE (Upsert Form)
- USING Row Descriptor
- WHERE Clause
- WITH Clause

ABORT

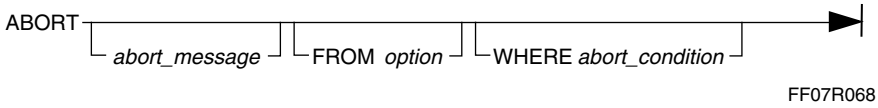
Purpose

Terminates the current transaction.

Invocation

Executable.

Syntax



where:

Syntax Element . . .	Specifies . . .
abort_message	the text of the message to be returned when the transaction is terminated. If *abort_message* is not specified, the message defaults to "user-generated transaction ABORT"
FROM *option*	an optional clause that might be required if the WHERE clause includes subqueries. The contents of the FROM option are described in "FROM Clause" on page 1-22.
WHERE *abort_condition*	an optional clause that introduces a conditional expression. The expression can specify an aggregate operation. If the WHERE clause is omitted, termination is unconditional. See "WHERE Clause" on page 1-32. The WHERE condition specifies an expression whose result must be true if termination is to occur. If the result is false, transaction processing continues.

ANSI Compliance

ABORT is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

ABORT and ROLLBACK Are Synonyms

ABORT is a Teradata synonym for the ANSI-compliant ROLLBACK statement.

ABORT With Correlated Subqueries

See "Correlated Subqueries and the ABORT/ROLLBACK Statements" on page 1-56 for information about using correlated subqueries with ABORT statements.

ABORT With a WHERE Clause

ABORT tests each value separately; therefore, the WHERE clause should not introduce both an aggregate and a nonaggregate value. The aggregate value becomes, in effect, a GROUP BY value, and the mathematical computation is performed on the group aggregate results.

For example, assuming that the following things are true, then the ABORT statement that follows incorrectly terminates the transaction.

- The table Test contains several rows,
- The sum of Test.colA is 188, and
- Only one row contains the value 125 in Test.colB

```
ABORT WHERE (SUM(Test.colA) <> 188)
AND (Test.colb = 125);
```

The preceding statement is processed first by performing an all-rows scan with the condition (colb = 125), which selects a single row and then computes intermediate aggregate results with the condition (SUM(cola) <> 188).

The condition tests true because the value of cola in the selected row is less than 188.

If ABORT … WHERE is used and it requires READ access to an object for execution, the user executing this DML statement must have SELECT right to the data being accessed.

The WHERE condition of an ABORT can include subqueries. If so, then the subqueries require FROM clauses, and the ABORT should have a FROM if it is desired that the scope of a reference in a subquery is the ABORT condition.

Also see "Correlated Subqueries" on page 1-41 and "ROLLBACK" on page 3-141.

Multiple ABORT Statements

If a macro or multistatement request contains multiple ABORT statements, those ABORT statements are processed in order, even if the expressions could be evaluated immediately.

Two Types of ABORTS

There are two categories of ABORT statements, those that can be evaluated by the parser and do not require access to a table and those that require table access.

If ABORT expressions are used that do not reference tables, and if their order of execution is not important relative to the other statements, then they should be placed ahead of any statements that reference tables so that the abort can be done at minimum cost.

In the following example, the first two ABORT statements can be evaluated by the parser and do not require access to tables. The third ABORT requires access to a table.

```
REPLACE MACRO macro_name (P1 INTEGER, P2 INTEGER)
AS. . .
ABORT 'error' WHERE :P1 < 0;
ABORT 'error' WHERE :P2 < 0;
SELECT. . .
ABORT 'error' WHERE tab.C1 = :P1;
```

Rules for Using ABORT in Embedded SQL

The following rules apply to using ABORT within an embedded SQL program.

- ABORT cannot be performed as a dynamic SQL statement.
- ABORT is not valid when you specify the TRANSACT(2PC) to the preprocessor.

Example 1

In the following example, the ABORT statement terminates macro execution if the row for the employee being deleted is not present in the Employee table.

Statements in the body of the macro are entered as one multistatement request. Therefore, if the WHERE condition of the ABORT statement is met, the entire request is aborted and the accuracy of the count in the Department table is preserved.

```
CREATE MACRO DelEmp (num SMALLINT FORMAT '9(5)',
   dname VARCHAR(12),
   dept SMALLINT FORMAT '999')
AS ( ABORT 'Name does not exist' WHERE :num NOT IN
   (SELECT EmpNo
   FROM Employee
   WHERE UPPER(Name) = UPPER(:dname))
   ; DELETE FROM Employee
   WHERE UPPER(Name) = UPPER(:dname)
   ; UPDATE Department
   SET EmpCount = EmpCount - 1
   WHERE DeptNo = :dept; ) ;
```

Example 2: Valid ABORT Statements

The following queries are syntactically correct.

```
ABORT WHERE user= 'DBC';

ABORT FROM table_1 WHERE table_1.x1 = 1;

ABORT FROM table_11 WHERE table_1.x1 > 1;

ABORT
FROM table_1,table_2
WHERE table_1.x1 = table_2.x2;
```

Example 3

The following example uses the ABORT statement in macro NewEmp. NewEmp adds a row to the Employee table for each new employee and then executes the SELECT statement to verify that the new information was entered correctly. The ABORT statement ensures that no new employee is inadvertently added to the executive office department (300).

```
CREATE MACRO NewEmp (
   number   INTEGER,
   name     VARCHAR(12),
   dept     INTEGER 100 TO 900,
   position VARCHAR(12),
   sex      CHAR,
   edlev    BYTEINT )
   AS ( ABORT 'Department number 300 not valid'
   WHERE :dept = 300 ;
   INSERT INTO Employee (EmpNo, Name, DeptNo,
   JobTitle, Sex, EdLev)
   VALUES (:number, :name, :dept, :position, :sex, :edlev) ;
   SELECT *
   FROM Employee
   WHERE EmpNo = :number ; ) ;
```

Related Information

See "ROLLBACK" on page 3-141.

BEGIN TRANSACTION

Purpose

Defines the beginning of a single logical transaction.

Invocation

Executable.

Syntax



ANSI Compliance

BEGIN TRANSACTION is a Teradata extension to the ANSI SQL-99 standard and is valid only in Teradata mode.

For ANSI mode transaction control statements, see "COMMIT" on page 3-33 and "ROLLBACK" on page 3-141.

Authorization

None.

Rules for Embedded SQL

The following rules apply to the use of BEGIN TRANSACTION in embedded SQL:

- BEGIN TRANSACTION is valid only when you specify the TRANSACT(BTET) or -tr(BTET) options to the preprocessor. Otherwise, an error is returned and the precompilation fails.
- BEGIN TRANSACTION cannot be performed as a dynamic SQL statement.

Explicit Transactions

An explicit, or user-generated, transaction is a single set of BEGIN TRANSACTION/END TRANSACTION statements surrounding one or more requests.

A LOGOFF command following a BEGIN TRANSACTION statement and one or more requests also forms an explicit transaction.

Note that this construct aborts the transaction and rolls back the processing results of all the requests between BEGIN TRANSACTION and LOGOFF.

All other transactions are implicit.

Implicit Transactions

An implicit, or system-generated, transaction, is typically one of the following:

- macro
- data manipulation statement that affects one or more table rows
- multistatement request that is not part of an explicit transaction and for which Teradata automatically supplies BEGIN TRANSACTION and END TRANSACTION statements.

Rules for Transactions Containing DDL Statements

A transaction can be any solitary SQL statement, including a DDL statement.

Note that when a request contains multiple statements, a DDL statement is allowed only if the following things are true:

- The transaction is explicit (bracketed by BEGIN TRANSACTION and END TRANSACTION statements).
- The DDL statement is the last statement in the transaction (immediately followed by the END TRANSACTION statement).

Teradata Transaction Handling Protocol

Teradata manages transactions to maintain valid, consistent, and available data for all users. When a transaction is received, various locks are placed on its associated database objects according to the types of requests contained in the transaction. Then, either all the requests in the transaction are processed, or none of them are processed.

If for some reason, such as a statement error, deadlock, access rights violation, table constraint violation, or premature logoff, a request does not complete successfully or causes processing to time out, Teradata performs the following transaction management steps to handle the problem:

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>The entire transaction is aborted.
Abort processing performs the following actions.
<table><tr><th>Stage</th><th>Process</th></tr><tr><td>1</td><td>Performs an implicit END TRANSACTION.</td></tr><tr><td>2</td><td>Backs out any changes made to the database by the transaction.</td></tr><tr><td>3</td><td>Releases any usage locks associated with the transaction.</td></tr><tr><td>4</td><td>Discards any partially accumulated results (spool files) generated by the transaction.</td></tr></table></td></tr>
<tr><td>2</td><td>A failure or time-out response is returned to requestor.</td></tr>
</table>

Using Statement Modifiers With BEGIN TRANSACTION

When Teradata receives a BEGIN TRANSACTION statement, it looks for an SQL statement keyword. Keep this in mind when determining the organization of statement modifiers.

For example, if the first statement in an explicit transaction is associated with a USING clause, the USING clause must precede the BEGIN TRANSACTION statement.

Nested BT/ET Pairs

When BT/ET pairs are nested, Teradata checks to ensure that each BT has a matching ET.

The outermost BT/ET pair defines the *explicit transaction*; the inner BT/ET pairs have no effect on the transaction.

Any embedded multistatement requests and macro executions are considered part of the outermost BT/ET explicit transaction and are not considered to be implicit transactions in this context.

For example, the following series of statements is treated as one explicit transaction:

```
BEGIN TRANSACTION;
   SELECT …;
   UPDATE …
   EXEC A(3,4);
   BEGIN TRANSACTION;
      UPDATE …;
      INSERT …
      ;INSERT …;
   END TRANSACTION;
   INSERT …;
END TRANSACTION;
```

If an error occurs in the middle of a nested BT/ET, everything rolls back to the initial BT.

Scenarios

The following scenarios illustrate the use of BEGIN/END TRANSACTION:

Scenario 1

Assuming that the Department table contains an EmpCount column, the following explicit transaction could be used to remove a row from the Employee table and then decrement a departmental head count in the Department table:

```
BEGIN TRANSACTION;
   DELETE FROM Employee
   WHERE Name = 'Reed C';
   UPDATE Department
   SET EmpCount = EmpCount -1
   WHERE DeptNo = 500;
END TRANSACTION;
```

To prevent the UPDATE statement from being executed in the event that the DELETE statement either fails or completes without removing a row, you must check the DELETE return code.

If a "No rows removed" condition is indicated, you must take action to bypass the update. Otherwise, the count result for department 500 is incorrect. Alternatively, the statements could be defined in a macro containing a conditional ABORT to protect the count.

Scenario 2

The following example illustrates an explicit transaction in which each statement, including the first, is associated with a USING clause:

```
USING ssnumfile (INTEGER)
BEGIN TRANSACTION ;
   INSERT INTO Employee (SocSecNo) VALUES (:ssnumfile) ;
      USING ssnumfile (INTEGER)
   INSERT INTO Employee (SocSecNo) VALUES (:ssnumfile) ;
      USING ssnumfile (INTEGER)
   INSERT INTO Employee (SocSecNo) VALUES (:ssnumfile) ;
END TRANSACTION ;
```

Scenario 3

The following examples illustrate the use a DDL statement in an explicit transaction. These transactions create a temporary table, perform an aggregate operation on the result of another aggregate operation, and then drop the temporary table.

Two transactions are used because a DDL statement must be either the only statement in a transaction, or the last statement in a transaction

```
BEGIN TRANSACTION;
CREATE TABLE DeptSumSal
(DeptNo SMALLINT FORMAT '999' BETWEEN 100 AND 900
      NOT NULL,
 SumSal DECIMAL(8,2) FORMAT 'ZZZ,ZZ9.99')
 PRIMARY INDEX(DeptNo);
END TRANSACTION;

BEGIN TRANSACTION;
INSERT INTO DeptSumSal
   SELECT DeptNo, SUM(Salary)
    FROM Employee
   GROUP BY DeptNo;
   SELECT AVG(SumSal)
    FROM DeptSumSal;
   DROP TABLE DeptSumSal;
END TRANSACTION;
```

Scenario 4: Implicit Transaction (BTEQ)

The following example is structured as a BTEQ multistatement request and therefore, is processed as a single implicit transaction.

Note the placement of the USING modifier and the semicolons. With this construct, the failure of any WHERE conditions causes the transaction to abort and all completed inserts and updates to be rolled back.

```
USING var1(CHAR), var2(CHAR), var3(CHAR)
   INSERT INTO TestTabU (C1) VALUES (:var1)
   ; INSERT INTO TestTabU (C1) VALUES (:var2)
   ; INSERT INTO TestTabU (C1) VALUES (:var3)
   ; UPDATE TestTabU SET C2 = C1 + 1
      WHERE C1 = :var1
   ; UPDATE TestTabU SET C2 = C1 + 1
      WHERE C1 = :var2
```

Related Information

See "END TRANSACTION" on page 3-50.

CALL

Purpose

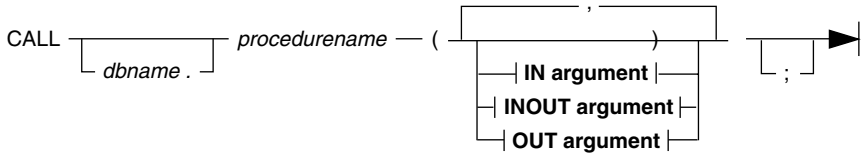
Invokes a stored procedure.

Invocation

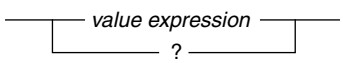
Executable.

Interactive SQL, embedded SQL, stored procedures, and macros.

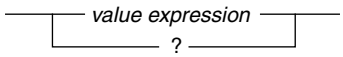
Syntax



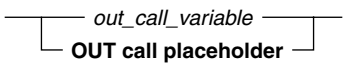
IN argument



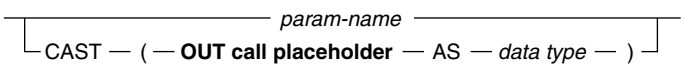
INOUT argument



OUT argument



OUT call placeholder



YSCALL001

where:

Syntax Element …	Specifies …
database_name	optional qualifier for the stored procedure to be executed. If *database_name* is not specified, the current default database is assumed.
procedure_name	the name of the stored procedure to be executed.
value_expression	Teradata-supported arithmetic and string expressions. The following can be specified in a value expression, subject to client-specific restrictions: • stored procedure local variables • stored procedure status variables • IN or INOUT parameters • for loop columns and aliases • host variables and macro parameters • FORMAT, TITLE, and NAMED phrases A value expression as a call argument must not reference tables, as it cannot contain subqueries. See "Rules for Call Arguments in Preprocessor2" later in this topic and the blocks following that for client-specific rules.
?	a call parameter argument. A QUESTION MARK character as an input call argument is valid only in ODBC and JDBC client applications.
out_call_variable	an identifier prefixed with the colon (:). Depending on the calling utility, the *out_call_variable* can be one of these: • host variable • local variable • IN or INOUT parameter
out_call_placeholder	a parameter name. The placeholder provides for nesting of placeholders (parameters or CAST … AS clauses)
param_name	the name of the OUT parameter as defined in the stored procedure.

Syntax Element . . .	Specifies . . .
CAST … AS	the request to convert the data definition of a parameter or another CAST clause to the required type. CAST clauses can be nested. FORMAT, NAMED and TITLE clauses can be used with the SQL CAST operator.
data_type	the desired data definition for the parameter set. All Teradata-supported data types are valid. See the chapter "SQL Lexicon" in *Teradata RDBMS SQL Reference, Volume 1* for more details on data types.

ANSI Compliance

CALL is ANSI SQL-99-compliant.

Authorization

You must have EXECUTE PROCEDURE right either on the stored procedure or on the database containing it.

Access Rights on Referenced Objects

As a user you need not have any privileges on the database objects referenced by the called stored procedure. The privileges are checked for the owner of the called stored procedure.

The owner must have the rights on the referenced objects with a WITH GRANT OPTION to allow other users to access or manipulate the object.

Rules for Executing Stored Procedures

The following rules apply to executing stored procedures:

- Stored procedures are platform-specific. A procedure created in UNIX (MP-RAS) or Windows can be executed only on that platform.
- CALL can be performed only in Teradata (BTET) or ANSI mode. CALL cannot be used in 2PC (Two Phase Commit) mode.

  If a stored procedure is created in Teradata mode, it cannot be executed in ANSI mode and vice versa.
- Additionally, CALL can be issued in Prepare or Execute mode using CLI applications.
- LOCKING modifiers cannot be used with a CALL statement.
- A CALL statement is not allowed in triggers, and in multi-statement SQL requests.
- A CALL statement may be submitted from a macro if it is the only statement inside the macro.

Delimiting the CALL Statement

Delimiting a CALL request with a semicolon is mandatory when CALL is invoked from BTEQ or from another stored procedure. It is not mandatory in other utilities.

General Rules for Specifying Input and Output Parameters

Call arguments consisting of input and output parameters must be submitted with the CALL statement. No default values can be defined for parameters at stored procedure creation time; so the CALL returns an error if the required call arguments are not specified.

The following rules apply to the parameter arguments:

- The number of call arguments in a CALL statement must be equal to the number of parameters in the called procedure.

  If the called procedure has no parameters, you cannot specify any call arguments.
- An IN, INOUT or OUT argument must correspond to an IN, INOUT or OUT parameter respectively in the called stored procedure
- IN parameters can only have input, and OUT parameters can only have output. INOUT parameters can have both input and output.
- The data type of a call argument must be compatible with the data type of the corresponding parameter in the called stored procedure.
- A value expression of NULL can be used to assign NULL value to the corresponding parameter in the stored procedure.
- On successful completion of the procedure execution, the ACTIVITY_COUNT in the Success or OK response is set to the following values:
  - 1, if the procedure has any output (INOUT or OUT) parameters.
  - 0, if the procedure has no output parameters.

Some additional rules and restrictions apply when CALL is invoked from different client utilities.

Rules for Call Arguments in BTEQ and CLIv2

The following additional rules apply to call arguments submitted from applications in BTEQ or CLIv2:

- An IN or INOUT argument must be a value expression.
- In a value expression used as IN or INOUT argument, identifiers prefixed by the colon (:), if any, must refer to USING variables associated with a USING clause for the request containing the CALL. The value of the expression is treated as the input value for the corresponding parameter in the called stored procedure.
- An OUT argument must be an “OUT call placeholder”. See syntax table for definition of the call placeholder.

Rules for Call Arguments in Preprocessor2

The following additional rules apply to a call argument when the SQL CALL statement is submitted from a PP2 application:

- An IN argument must be a value expression.

  Identifiers prefixed by the colon (:), if any, in the value expression must refer to host variables. The value of the expression is treated as the input value for the corresponding parameter.
- An INOUT argument must be a value expression that is limited to the form of an OUT-call-variable. The identifier must refer to a host variable.
- An OUT argument must be an OUT-call-variable. The identifier must refer to a host variable.

Rules for Call Arguments in ODBC and JDBC

The following additional rules apply to a call argument when the SQL CALL statement is submitted from an ODBC or JDBC application:

- An IN or INOUT argument must be one of the following:
  - a value expression. A value expression must not contain identifiers prefixed by the COLON character. It must be a constant expression.
  - a QUESTION MARK character used as an input placeholder.
  - If ? is specified, the value for the corresponding IN or INOUT parameter of the called procedure must be set using ODBC or JDBC specific calls prior to calling the stored procedure.
- An OUT argument must be an OUT call placeholder.

Rules for Call Arguments in Nested Procedures

The following additional rules apply to a call argument when the SQL CALL statement is submitted from another stored procedure:

- An IN argument must be a value expression.

  Identifiers prefixed by the colon (:), if any, in the value expression must refer to local variables, status variables, IN and INOUT parameters, or for-loop variable qualified columns and aliases of the calling stored procedure.
- An INOUT argument must be a value expression that is limited to the form of an OUT-call-variable. The identifier must refer to a local variable or an INOUT parameter of the calling stored procedure.
- An OUT argument must be an OUT-call-variable. The identifier must refer to a local variable or an INOUT or OUT parameter.

Session Dateform and Called Procedures

Stored procedures retain the dateform (date and time formats) that was in effect when they were created. They do not reflect the dateform in effect for the session in which they are called unless it is the same as the dateform in effect when the procedure was created.

See "SET SESSION DATEFORM" in *Teradata RDBMS SQL Reference, Volume 4*.

Retrieving Values of Output Parameters

The output values of INOUT and OUT parameters are returned after completion of the CALL operation in the following forms:

When CALL is issued from this type of application …	The output parameter values are …
BTEQ	a result row.
CLIv2	The values for all the INOUT and OUT parameters are returned in one row, irrespective of the number of parameters. The value of each parameter is a field in the output.
embedded SQL	stored in the corresponding host variable.
another stored procedure	stored in the corresponding local variable or parameter of the calling stored procedure.
ODBC	to be retrieved using an ODBC API.
JDBC	to be retrieved using a JDBC API.

Status of Unqualified Referenced Objects

If the objects referenced in the called stored procedure are not qualified by a database name, then they are qualified by the name of the default database at the time the stored procedure was created.

Dropped, Renamed, or Replaced Objects

Consider the scenario when a stored procedure references another procedure that has been dropped, renamed or replaced.

If the called stored procedure X references another stored procedure Y, and Y is dropped, renamed or replaced before executing X, the following happens.

In this mode …	This condition is generated when X submits an internal CALL to execute Y …
ANSI	Error
Teradata	Failure

The execution of Y generates an error or failure if the arguments specified are incompatible with the changed definition of Y as in these cases:

- the number of call arguments is not equal to the number of parameters in the changed definition of Y
- the data type of the call arguments is incompatible with the changed parameters of Y

These conditions also apply to the attributes of any referenced database objects.

CALL Request Priority

A stored procedure executes at the priority the user session is running at the time the CALL statement is submitted. If you specify a SET SESSION ACCOUNT at a REQUEST level and the request submitted is a CALL statement, the stored procedure along with the SQL statements within the procedure execute at the specified priority.

The account is set to the previous account after the CALL statement is completed.

If the user priority is changed asynchronously from the Performance Monitor during stored procedure execution, the new priority takes effect for all the subsequent statements that are submitted as part of stored procedure execution.

If the asynchronous change is at the request level, the priority change applies to all the statements executed as part of the CALL operation.

Consider the following stored procedure:

```
CREATE PROCEDURE prio2()
BEGIN
   INSERT INTO temp(1, 'stored procedure before prio1') /* STMT1 */;
   CALL prio1()                                         /* STMT2 */;
   INSERT INTO temp(2, 'stored procedure after prio1')  /* STMT3 */;
END;
```

The following scenarios explain the priority scheduling for the stored procedure execution:

Scenario 1

```
LOGON user1/user1, acct1;
CALL prio2() /* this, along with the SQL statements inside the stored
procedure are
             ** executed at the priority associated with acct1 */;
```

Scenario 2

```
LOGON user1/user1, acct1;
CALL prio2() /* this, along with the SQL statements inside the stored
procedure are
             ** executed at the priority associated with acct1 */;
SET SESSION ACCOUNT acct2 FOR REQUEST;
Call prio2() /* this, along with the SQL statements inside the stored
procedure are
             ** executed at the priority associated with acct2 */;
SELECT * FROM temp /* this is executed at the priority associated with
acct1 */;
```

Scenario 3

Assume that the priority is changed for user1 to acct2, after executing the STMT 1. The STMT 2 and STMT 3 (along with the SQL statements inside prio1) execute in the changed priority. If the priority is changed at request level, the session priority is restored to that corresponding to acct1 at the end of the execution of prio2.

```
LOGON user1/user1, acct1;
CALL prio2() /* this is executed at the priority associated with
acct1*/;
```

Errors and Failures in Procedure Execution

An error (in ANSI mode) or failure (in Teradata mode) is reported during stored procedure execution in any of the following situations:

- the database objects referenced in the stored procedure have been dropped or renamed after the stored procedure creation
- the attributes of the referenced objects or of the parameters of the objects have been changed
- translation error or failure occurs when the value of an IN or INOUT argument is converted implicitly to the corresponding parameter data type and assigned to the corresponding parameter.

See "Dropped, Renamed, or Replaced Objects" on page 3-18.

Errors and Failures in Nested Procedures

If an error or failure occurs in case of nested stored procedures, the error or failure message text is preceded by the name of the stored procedure in which the error has occurred.

Assume that stored procedure X references another stored procedure, Y.

If Y returns an error or failure during the execution of X, the error or failure text contains “Y” as the stored procedure name.

If the error or failure occurs in the calling stored procedure X, the error or failure text contains “X” as the stored procedure name.

Aborting a CALL Statement

You can use the standard abort facility provided by the client utility or interface to abort a CALL request. The following action takes place:

IF the abort request is specified . . .	THEN this action is taken. . .
During stored procedure execution	the execution stops. The transaction being processed at that time is rolled back, and a failure response is returned.
After the execution completes	the response of the CALL statement is returned to the client.

Examples

The following cases are examples of correct usage of the CALL statement and arguments.

Example 1: Input Arguments in BTEQ and CLIv2

Consider the following stored procedure spSample1 that has three IN parameters, p1, p2, and p3 of INTEGER data type. The argument list can be specified in either of the following formats:

```
CALL spSample1 (1, 2, 3);

CALL spSample1(1, CAST(((2 + 4) * 2) AS FORMAT 'ZZZ9'), 3);
```

The rule of specifying the arguments as a positional list applies to all arguments (IN, INOUT, or OUT).

Example 2: Input & Output Arguments in BTEQ and CLIv2

Consider the following stored procedure. The stored procedure definition can be viewed using an SQL SHOW PROCEDURE statement to verify the parameter types:

```
CREATE PROCEDURE spSample2(OUT p1 INTEGER, INOUT p2 INTEGER,
IN p3 INTEGER)
BEGIN
   SET p1 = p3;
   SET p2 = p2 * p3;
END;
```

The call arguments can be specified in any of the following formats:

Format 1

```
CALL spSample2(p1, 20, 3);
```

where, the parameter name p1 is supplied as an argument because it is a placeholder for an OUT parameter. The values 20 and 3 are passed as arguments to the INOUT and IN parameters, p2 and p3 respectively.

You get the following response from the CALL statement:

```
*** Procedure has been executed..
 *** Total elapsed time was 1 second.

         p1           p2
-----------  -----------
          3           60
```

Format 2

```
CALL spSample2(p1, 20 * 2, 30 + 40);
```

where the expressions (20 * 2) and (30 + 40) are passed as arguments to the INOUT and IN parameters, p2 and p3.

You get the following response from the CALL statement:

```
*** Procedure has been executed..
 *** Total elapsed time was 1 second.

         p1           p2
-----------  -----------
         70         2800
```

Format 3

```
CALL spSample2(CAST(CAST(p1 AS CHAR(10)) AS TITLE
'OutputValue'), CAST(20 AS SMALLINT), 30 + 40);
```

where the expressions CAST(20 AS SMALLINT) and (30 + 40) are passed as arguments to the INOUT and IN parameters, p2 and p3.

You get the following response from the CALL statement:

```
*** Procedure has been executed..
 *** Total elapsed time was 1 second.

OutputValue      20
-----------  ----------
70               1400
```

Format 4

This format is used with BTEQ and assumes that data is being imported from a data file.

```
USING (a INTEGER) CALL spSample1(p1, :a, 3);
```

where 3 is the value passed as an argument to the IN parameter p3. The value read from the input data file is assigned to the variable a and is passed as an argument to the INOUT parameter p2 via a USING clause.

BTEQ receives the following response from the server:

```
*** Procedure has been executed..
 *** Total elapsed time was 1 second.

         p1           p2
-----------  -----------
          3           30
```

Format 5 (Using NAMED and TITLE phrases in the SQL CALL statement)

```
CALL spSample1(CAST (p1 AS NAMED AA TITLE 'OUT VALUE'),
CAST (((20 * 2) + 3) AS TITLE 'INOUT VALUE'), 1);
```

The response looks like this:

```
*** Procedure has been executed..
 *** Total elapsed time was 1 second.

  OUT VALUE  INOUT VALUE
-----------  -----------
          1           43
```

Example 3: Stored Procedure or Embedded SQL Input Arguments

Consider the stored procedure spSample2 defined in Example 2. The arguments can be specified in any of the formats shown:

Format 1

Only literals or constant expressions are specified as arguments for the parameters:

- In a stored procedure:

  ```
  CALL spSample2(1, 2, 3 + 4);
  ```

- In a C program using embedded SQL:

  ```
  EXEC SQL CALL spSample2(1, 2, 3 + 4);
  ```

Format 2

The application variables contain the values that are passed as arguments:

- In a stored procedure:

```
SET AppVar1 = 10;
SET AppVar2 = 30;
SET AppVar3 = 40;
CALL spSample2(:AppVar1, :AppVar1 + :AppVar2, CAST(:AppVar3 AS
FORMAT 'Z,ZZ9'));
```

- In a C program using embedded SQL:

```
AppVar1 = 10;
AppVar2 = 30;
AppVar3 = 40;
EXEC SQL CALL spSample2(:AppVar1, :AppVar1 + :AppVar2,
CAST(:AppVar3 AS FORMAT 'Z,ZZ9'));
```

Format 3

The combination of the application variables (AppVar1, AppVar2, and AppVar3) and values/expressions are specified as arguments:

- In a stored procedure:

```
SET AppVar1 = 10;
SET AppVar2 = 30;
SET AppVar3 = 40;
CALL spSample2(:AppVar1, 3 + :AppVar2, 3 + 4 + :AppVar3);
```

- In a C program using embedded SQL:

```
AppVar1 = 10;
AppVar2 = 30;
AppVar3 = 40;
EXEC SQL CALL spSample2(:AppVar1, 3 + :AppVar2, 3 + 4 +
:AppVar3);
```

Note that no output parameters are returned from the stored procedure using this format, so the ACTIVITY_COUNT is returned as 0 in the success response.

Example 4: Stored Procedure and Embedded SQL Input and Output Arguments

Consider the stored procedure spSample2. The arguments can be specified in the following format:

Format 1

- In a stored procedure:

```
SET AppVar2 = 30 + AppVar3;
SET AppVar3 = 40;
CALL spSample1(:AppVar1, :AppVar2, :AppVar3);
```

- In a C program using embedded SQL:

```
AppVar2 = 30 + AppVar3;
AppVar3 = 40;
EXEC SQL CALL spSample1(:AppVar1, :AppVar2, :AppVar3);
```

The values specified for AppVar2 and AppVar3 are passed as arguments to p2 and p3, respectively. When the stored procedure execution completes, the output parameter values are returned in AppVar1 and AppVar2. ACTIVITY_COUNT is set to 1.

- Format 2
- In a stored procedure:

```
SET AppVar2 = 30 + AppVar3;
SET AppVar3 = 40;
CALL spSample1(:AppVar1, :AppVar2, :AppVar3 + 3);
```

- In a C program using embedded SQL:

```
AppVar2 = 30 + AppVar3;
AppVar3 = 40;
EXEC SQL CALL spSample1(:AppVar1, :AppVar2, :AppVar3 + 3);
```

The values for p2 and p3 are AppVar2 and (3 + AppVar3), respectively. When the stored procedure execution completes, the output parameter values are returned in AppVar1 and AppVar2. ACTIVITY_COUNT is set to 1.

Example 5: Input and Output Arguments in ODBC

The following example describes the usage of the stored procedures specifying input and output arguments in an ODBC application.

Consider the stored procedure spSample2 defined as follows:

```
CREATE PROCEDURE spSample2(OUT p1 INTEGER, INOUT p2 INTEGER,
IN p3 INTEGER)
BEGIN
   SET p1 = p3;
   SET p2 = p2 * p3;
END;
```

The arguments can be specified in the following format:

```
SQLBindParameter(..., 1, SQL_PARAM_INPUT_OUTPUT, ...,
SQLINTEGER, ..., ..., AppVar2, sizeof(AppVar2), ...);
SQLBindParameter(..., 2, SQL_PARAM_INPUT, ..., SQLINTEGER,
..., ..., AppVar3, sizeof(AppVar3), ...);
```

where the second argument in the `SQLBindParameter()` is the question mark number ordered sequentially from left to right, starting at 1.

Executing the stored procedure:

```
{
constchar *request = "CALL spSample2(p1, ?, ?)";
...
...
SQLExecDirect(hstmt, request);
...
...
}
```

Retrieving the output parameter values:

```
SQLBindCol(..., 1, ..., AppVar1, ..., ...);
SQLBindCol(..., 2, ..., AppVar2, ..., ...);
```

where the second argument in the `SQLBindCol()` is the parameter number of result data, ordered sequentially left to right, starting at 1.

Example 6: Input and Output Arguments in JDBC

Consider the stored procedure spSample2. It can be executed using the following JDBC API calls:

```
CallableStatement cstmt = con.prepareCall("CALL spSample2(p1,
?, ?)");
ResultSet rs = cstmt.executeQuery();
```

The following alternatives can be used for the second line (`ResultSet rs =` ...):

```
boolean bool = cstmt.execute();
```

or

```
int count = cstmt.executeUpdate();
```

The QUESTION MARK character indicates an argument and acts as a placeholder for IN or INOUT parameters in the `prepareCall()` statement. The question mark placeholder arguments need to be bound with the application local variables/literals using the `CallableStatement.setXXX()` JDBC API calls.

Alternatively, an IN or INOUT argument can be a constant expression.

The INOUT and OUT arguments need to be registered using the following JDBC API call:

```
CallableStatement.registerOutParameter()
```

After execution the parameter values can be retrieved from the response using the `CallableStatement.getXXX()` JDBC API calls.

CHECKPOINT

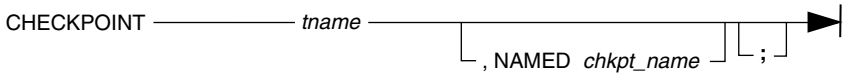
Purpose

Places a flag in a journal table that can be used to coordinate transaction recovery.

Invocation

Executable.

Interactive Syntax

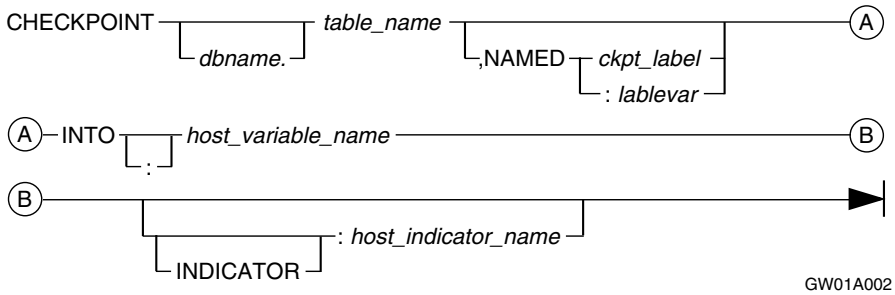


FF07A007

where:

Syntax Element . . .	Specifies . . .
table_name	the journal table that is to be marked with the checkpoint entry.
checkpoint_name	a label that can be used to reference the checkpoint entry in database recovery activities. *checkpoint_name* should be unique. If *checkpoint_name* duplicates an existing entry in the journal, then you can qualify it with the system-assigned event number. If you do not specify *checkpoint_name*, then the checkpoint entry must be referenced in recovery activities by its event number.

Embedded SQL and Stored Procedure Syntax



GW01A002

where:

Syntax element ...	Specifies ...
database_name	the name of the database for which a checkpoint flag is to be created. The period after the name is required.
table_name	the name of the table for which a checkpoint flag is to be created.
checkpoint_label	an SQL identifier that labels the checkpoint.
label_variable	a host variable that contains a label for the checkpoint. The colon is required.
host_variable_name	the name of a host variable to contain the checkpoint flag. The preceding colon is optional.
host_indicator_name	the name of an optional host indicator variable to receive nulls.

ANSI Compliance

CHECKPOINT is a Teradata extension to the ANSI SQL-99 standard.

Authorization

To checkpoint a journal table, you must meet at least one of the following criteria.

- Have CHECKPOINT privilege on the journal table.
- Have CHECKPOINT privilege on the database containing the journal table.
- Be an owner of the database containing the journal table.
- Be an immediate or indirect owner of the journal table.

General Usage Notes

CHECKPOINT is a data returning statement.

If an explicit transaction or a multistatement request contains a CHECKPOINT statement, then that CHECKPOINT statement must precede any INSERT, UPDATE, or DELETE statements in the transaction or request.

The CHECKPOINT statement causes Teradata to place a read lock on all data tables that write journal images to the table named in the CHECKPOINT statement. This lock causes any new transactions to wait until the checkpoint operation is complete. It also causes the checkpoint operation to await the completion of outstanding update transactions.

This action guarantees that the checkpoint saved in the journal represents a clean point in the transaction environment. When the checkpoint operation completes, the locks are released.

Teradata assigns an event number to each checkpoint entry. This number can be returned as a result of CHECKPOINT processing; it is also stored, along with other information about statement execution, in a data dictionary table. You can review the table data through the DBC.Events system view.

Usage Notes for Embedded SQL

The following rules apply to CHECKPOINT:

- Whether specified as *ckpt_label* or as *labelvar*, the checkpoint label must be a valid SQL identifier.
- If *labelvar* is specified, the host variable must follow the rules for SQL strings for the client language and must be preceded by a colon. See *"Input Host Variables" on page 4-12* for details.
- The main host variable identified by host variable name must be of a type that conforms to INTEGER.
- The CHECKPOINT statement, although a data returning statement, cannot be associated with a selection cursor.
- CHECKPOINT cannot be performed as a dynamic SQL statement.

CHECKPOINT causes a synchronization point to be generated and recorded in the journal table specified by *table_name*.

If you specify the NAMED clause, the checkpoint label is associated with the synchronization point. A 32-bit integer that uniquely identifies the synchronization point is returned into the main host variable defined in the host variable specification.

CHECKPOINT is not valid when you specify the TRANSACT(2PC) option to the SQL precompiler.

Example

The following statement can be used to place a checkpoint entry into a journal table:

```
CHECKPOINT AccountDB.JnlTable, NAMED DailyPayMaint;
```

When this statement is executed, all update activity is ceased on tables that write journal images to the journal table, (JnlTable).

A record containing the DailyPayMaint label is placed into the journal table. The label can then be used in rollforward or rollback operations.

COMMENT (Placing Form)

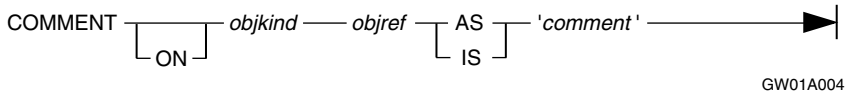
Purpose

Adds or replaces an object comment.

Invocation

Executable.

Syntax



where:

Syntax element …	Specifies …
object_kind	One of these objects: • COLUMN • DATABASE • MACRO • PROCEDURE • PROFILE • ROLE • TABLE • TRIGGER • USER • VIEW
object_reference	One of these object references: • column name • database name • macro name • procedure name • profile name • role name • table name • trigger name • user name • view name
'comment'	the text of the comment, enclosed in quotes.

ANSI Compliance

COMMENT is a Teradata extension to the ANSI SQL-99 standard.

Authorization

To place a comment in a database object definition, you must have the DROP privilege on the object.

For example, to place a comment in a MACRO definition, you must have the DROP MACRO privilege on that macro.

COMMIT

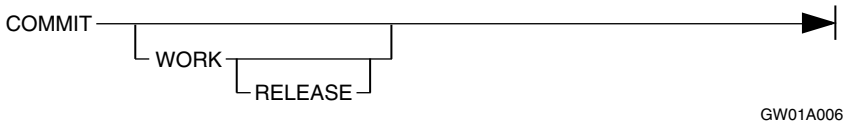
Purpose

Terminates the current ANSI SQL transaction and commits all changes made within it.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
WORK	that if the SQL Flagger is enabled, the absence of WORK causes the statement to be flagged.
RELEASE	the connection between the client application program and Teradata, whether implicit or explicit, is to be terminated. This option applies to embedded SQL only and is not ANSI-compliant.

ANSI Compliance

COMMIT is ANSI SQL-99-compliant with an extension.

COMMIT is not valid in Teradata mode. If used, it returns a failure and aborts the transaction.

Authorization

None.

How ANSI Transactions Are Defined and Terminated

In ANSI mode, the first SQL statement in a session initiates a transaction. The transaction is terminated by sending either a COMMIT, ROLLBACK, or ABORT statement, or failure causes a rollback of the transaction.

COMMIT Is Explicit

There are no *implicit* transactions in ANSI mode. The COMMIT must always be explicitly stated and be the last statement of a transaction in order for the transaction to terminate successfully.

Rules for Embedded SQL

The following rules apply to the COMMIT statement within an embedded SQL application:

- COMMIT cannot be performed as a dynamic SQL statement.
- COMMIT discards dynamic SQL statements prepared within the current transaction.
- COMMIT closes open cursors.
- COMMIT is valid only when you specify the TRANSACT(COMMIT), -tr(COMMIT), TRANSACT(ANSI), or -tr(ANSI) options to the preprocessor.

  Its use causes an error if the program is precompiled with the TRANSACT(BTET), -tr(BTET), or the TRANSACT(2PC) preprocessor options.
- If you specify the RELEASE option, then the application program connection to Teradata (whether explicit or implicit) is terminated.

  If additional SQL statements (other than CONNECT or LOGON) are then performed, an implicit connection is attempted.
- RELEASE is not valid when you specify the TRANSACT(ANSI) or -tr(ANSI) options to the preprocessor.

Relation to ABORT and ROLLBACK

The ABORT and ROLLBACK statements (see "ABORT" on page 3-3 and "ROLLBACK" on page 3-141) also cause the current transaction to be terminated, but with rollback rather than commit.

COMMIT and BTEQ

If you use COMMIT in a BTEQ script with either the .SESSION or the .REPEAT command, you must send the COMMIT statement along with the repeated SQL statement as one request.

If you send the repeated request without the COMMIT, one of the requests is eventually blocked by other sessions and the job hangs because of a deadlock.

BTEQ Example

The following dummy example illustrates how this should be done:

```
.session trans ansi
.sessions 10
.logon TDPID/USER,PASSWD

.import data file = <data file name>
.repeat I
USING i(INTEGER), j(INTEGER)
INSERT INTO <table name> (col1, col2)
VALUES (:1, :j); COMMIT WORK;

.quit
```

Examples

The following examples illustrate the use of COMMIT:

Example 1

The INSERT statement in the following example opens the transaction. COMMIT closes the transaction.

```
INSERT INTO employee (name, empno)
VALUES ('Sinclair P', 101)
WHERE dept = '400';
COMMIT;
```

Example 2

The following UPDATE initiates the transaction and COMMIT WORK terminates it:

```
UPDATE parts SET part_num = 20555
WHERE location = 'seoul';
COMMIT WORK;
```

Related Information

See "ROLLBACK" on page 3-141.

DELETE

Purpose

Removes rows from a table.

There are three forms of DELETE.

- Basic/Searched (see "DELETE (Basic/Searched Form)" on page 3-42)
- Join Condition (see "DELETE (Join Condition Form)" on page 3-44)
- Positioned (see "DELETE (Positioned Form)" on page 3-46)

This topic describes properties common to all 3 forms.

Locks

A DELETE operation sets a WRITE lock for the table or row.

Rules for Using DELETE

The following rules apply to using DELETE.

- All alias names must be defined in the FROM clause.
- The activity count in the success response for a DELETE statement reflects the total number of rows deleted.
- Deletes that reference objects in multiple databases should use fully qualified names. Name resolution problems may occur if referenced databases contain tables or views with identical names and these objects are not fully qualified. Name resolution problems may even occur if the identically named objects are not themselves referenced.

DELETE With Correlated Subqueries

See "Correlated Subqueries and the DELETE Statement" on page 1-54 for information about using correlated subqueries with DELETE statements.

Unconstrained (Fast Path) DELETE Processing

When submitted by itself as a transaction, an unconstrained delete (DELETE...ALL) is processed very quickly.

An unconstrained DELETE is one where no conditions are attached, that is, all rows are deleted. When an unconstrained delete is submitted as part of a multistatement transaction, the speed with which it is processed depends on its position in the transaction.

To enable fast path processing, which defers processing until a transaction ends and allows rollback processing to be almost instantaneous, positioning must be as follows, depending on the mode being used.

IN this mode . . .	FOR this type of transaction . . .	Positioning must be as follows . . .
ANSI	All	The unconditional DELETE must be immediately followed by a COMMIT. For example, DELETE FROM tname ALL;COMMIT;
Teradata	Implicit (such as a multistatement macro or BTEQ request)	DELETE...ALL must be the last statement in the request.
	Explicit	DELETE...ALL must be immediately followed by the END TRANSACTION statement that terminates the currently open transaction. This implementation is not available to a Teradata embedded SQL application because the DELETE and END TRANSACTION statements must fall into the same request packet.

Constrained (Slow Path) DELETE Processing

Processing time for a constrained delete (DELETE...WHERE) can be longer when any of the following conditions are true.

- The FALLBACK option is specified for the table, because the rows in the secondary copy of the table must also be deleted.
- Secondary indexes are defined for the table, because the secondary index subtable also must be updated to account for the rows deleted.

Processing time for a constrained delete can be shortened by specifying a primary index value or a unique secondary index value as the conditional expression of the WHERE clause.

Duplicate Rows and DELETE Processing

It is not possible to distinguish among duplicate rows in a MULTISET table. Because of this, when a WHERE condition identifies a duplicate row, then all of the duplicate rows are deleted.

Deleting Rows Using Views

To delete table rows using a view through which it is accessed, the following things must be true.

- You must have the DELETE privilege on the view. Also, the immediate owner of the view (that is, the database in which the view resides) must have the DELETE privilege on the underlying object (view or base table) whose rows are to be deleted, and the SELECT privilege on all tables that are specified in the WHERE clause.
- Each column of the view must correspond to a column in the underlying table (that is, none of the columns in the view can be derived using an expression).
- The data type definitions for an index column should match in both the view definition and in the base table to which the view refers.

  While it is true that you can generally convert the type of a view column (for example, from VARCHAR to CHAR), if that converted column is a component of an index, then that index is not used to delete rows from the base table because the data type of the recast column no longer matches the data type definition for that column in the index.

  The resulting all-AMP, all-row behavior of the delete operation circumvents the performance advantages for which the index was designed.
- No two view columns can reference the same table column.
- The view cannot include a column that contains a range constraint.
- The expression used to define a view cannot have a data type specified for any column in the view.

Subqueries in DELETE Statements

DELETE statement predicates can include subqueries that reference the delete table as well as other tables. The following DELETE statement is an example:

```
DELETE FROM Publisher
WHERE 0 =
 (SELECT COUNT(*)
  FROM book
  WHERE book.PubNum=Publisher.PubNum);
```

There are two publishers who have books in the library and two publishers who do not.

The subquery executes once for each row of the outer reference, the Publisher table. Because two publishers have no books in the library, the two rows that represent those publishers are deleted from the Publisher table.

To modify this DELETE to use a noncorrelated subquery, change the subquery code to include all tables it references in its FROM clause.

```
DELETE FROM Publisher
WHERE 0 =
 (SELECT COUNT(*)
  FROM book, publisher
  WHERE book.PubNum=Publisher.pubnum);
```

When coded this way, the subquery predicate has a local defining reference, so the statement does not contain a correlated subquery. The count, determined once, is nonzero, so no rows are deleted.

The sections on correlated subqueries in Chapter 1: "The SELECT Statement" describes additional examples of correlated and noncorrelated subqueries.

Examples

The following examples illustrate the use of DELETE:

Example 1

To delete rows from the Employee table for employees in Department 500:

```
DELETE FROM Employee
WHERE DeptNo=500;
```

Example 2

To delete all rows from Employee:

```
DELETE FROM Employee ALL;
```

Example 3

To delete the row for employee 10011 from Employee:

```
DELETE FROM Employee
WHERE EmpNo=10011;
```

Example 4

To delete from Employee rows for employees who work in NYC (this operation joins Employee and Department tables):

```
DELETE FROM Employee
WHERE Employee.DeptNo = Department.DeptNo
AND Department.Location = 'NYC';
```

Example 5

To delete from Employee rows for managers who have less work experience than their employees (this operation performs a self-join on the Employee table):

```
DELETE FROM Employee AS Managers
WHERE Managers.DeptNo = Employee.DeptNo
AND Managers.JobTitle IN ('Manager', 'Vice Pres')
AND Employee.YrsExp > Managers.YrsExp;
```

Example 6

In the following macro, the SELECT and DELETE statements are structured as a single multistatement request:

```
CREATE MACRO ResUse
      (FromDate (DATE,       DEFAULT DATE),
       ToDate   (DATE,       DEFAULT DATE),
       FromTime (INTEGER,    DEFAULT 0),
       ToTime   (INTEGER,    DEFAULT 999999),
       Proc     (VARCHAR(4), DEFAULT 'P'),
       Secs     (SMALLINT,   DEFAULT 600) )
   AS (SELECT
        TheDate (TITLE 'Resource//TheDate'),
        TheTime (TITLE 'Utilized//TheTime'),
        Proc,
        AVERAGE(Hits) (TITLE 'Avg//Hits'),
        MAXIMUM(Cpu)  (TITLE 'Max//CPU'),
        AVERAGE(Cpu)  (TITLE 'Avg//CPU'),
        MAXIMUM(Disk) (TITLE 'Max//Disk'),
        AVERAGE(Disk) (TITLE 'Avg//Disk'),
        MAXIMUM(Host) (TITLE 'Max//Host'),
        AVERAGE(Host) (TITLE 'Avg//Host'),
        MAXIMUM(Chan) (TITLE 'Max//Chan'),
        AVERAGE(Chan) (TITLE 'Avg//Chan')
       FROM DBC.ResUseView
       GROUP BY TheDate, TheTime, Proc
       WHERE TheDate BETWEEN :FromDate AND :ToDate
       AND TheTime BETWEEN :FromTime AND :ToTime
       AND Proc CONTAINS :Proc AND Secs EQ :Secs
       ORDER BY Proc, TheDate, TheTime;

      DELETE FROM ResUseView ALL ; ) ;
```

In Teradata mode, if the preceding macro was executed, and not within a user-generated transaction, the DELETE statement would be processed using the "fast path" method.

In ANSI mode, the “fast path” method is used (no WHERE clause) when a COMMIT is included in the macro or COMMIT is used at the end of the request line, and immediately following the execution of the macro, within the request.

The last line of the example above would read:

```
DELETE FROM ResUseView ALL ;
COMMIT;) ;
```

The slow path method is used when no COMMIT statement is stated in the macro, or the request that executes the macro.

Example 7

The DELETE...ALL statement in the following BTEQ request also invokes “fast path” processing. Note that the statements are structured as a single multistatement request, and, therefore, are processed as an implicit transaction. Note that DELETE is the last statement in the series.

```
SELECT *
FROM DBC.LogOnOff
WHERE LogDate = DATE
AND UserName = 'Administrator'
;SELECT LogTime,
UserName, Event, AccountName
FROM DBC.LogOnOff
WHERE LogDate = DATE
AND UserName NOT IN ('Administrator', 'SecAdmin', 'Oper')
;SELECT LogTime, Event, LogicalHostId, IFPNo
FROM DBC.LogOnOff
WHERE LogDate = DATE
AND UserName = 'Oper'
;DELETE FROM DBC.LogOnOff ALL ;
```

In ANSI mode, the COMMIT statement must follow the DELETE statement. The last line of the above example would read:

```
DELETE FROM DBC.LogOnOff ALL ;
COMMIT;
```

DELETE (Basic/Searched Form)

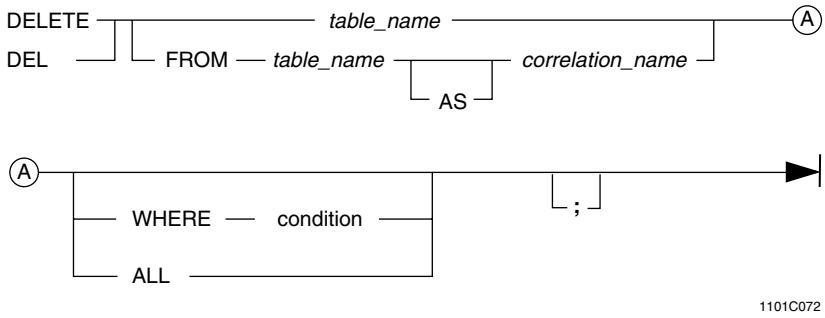
Purpose

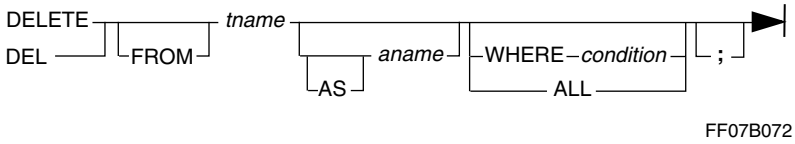
Removes one or more rows from a table.

Invocation

Executable.

Syntax





where:

Syntax Element . . .	Specifies . . .
table_name	the table the DELETE is to remove rows from. Used when the DELETE includes a condition that directly references more than one table.
FROM *table_name*	the name of the table to delete rows from (Basic DELETE). This phrase is optional for the Basic DELETE statement.
AS	an optional keyword introducing the alias name of the table to be deleted from.
correlation_name	an optional alias name, used for self-joins. ANSI calls table aliases correlation names. They are also referred to as range variables.
WHERE	the conditional clause. See "WHERE Clause" on page 1-32.

Syntax Element . . .	Specifies . . .
condition	the conditional expression to be used for determining rows to be deleted. The expression operands can be constants, or references to fields in the specified table or other tables. The conditional also can specify an embedded SELECT operation.
ALL	that all rows to be deleted, resulting in an empty table. This is the default and is used when a WHERE condition is not specified. The ALL option is a non-ANSI Teradata extension.

ANSI Compliance

The basic/searched form of DELETE is ANSI SQL-99-compliant with extensions.

Authorization

You must have the DELETE privilege on the table.

If the DELETE statement is a 'searched' DELETE (includes the WHERE condition), you must also have SELECT privilege on all tables accessed for checking *condition*.

Use caution when granting the privilege to delete data through a view. Data in fields that might not be visible to the user is also deleted when a row is deleted through a view.

Recommendation

This form of DELETE is ANSI-compliant and should be used for all DELETEs referencing a single table, or referencing tables other than the one deleted from only by subqueries in the *condition*.

Embedded SQL and Stored Procedure Error Condition Handling

If DELETE is specified with a WHERE clause and the specified search condition fails (selects no rows), the value +100 is assigned to SQLCODE and no rows are deleted.

Related Information

See "DELETE" on page 3-36.

DELETE (Join Condition Form)

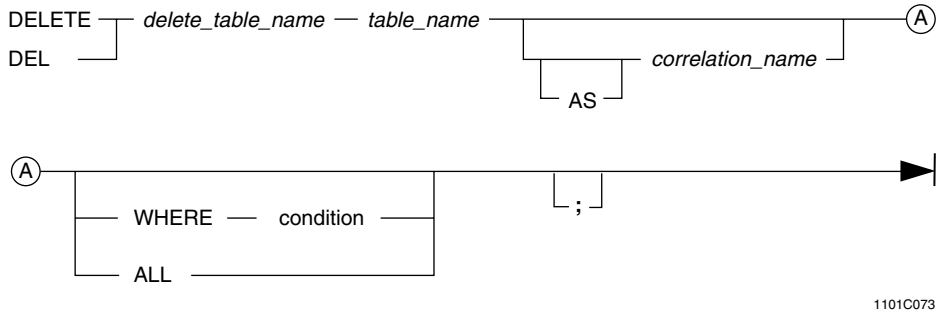
Purpose

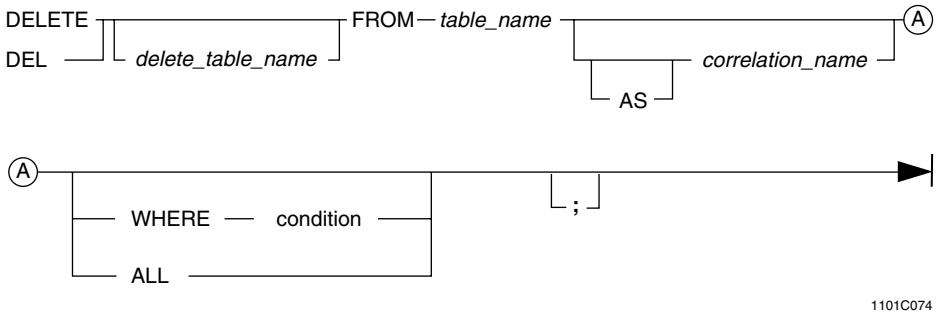
Removes rows from a table when the WHERE condition directly references fields in tables other than the one that rows are to be deleted from, that is, if the WHERE *condition* includes a subquery or references a derived table.

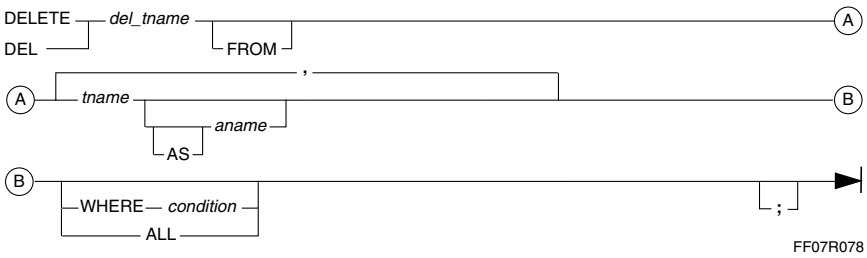
Invocation

Executable.

Syntax







where:

Syntax Element . . .	Specifies . . .
delete_table_name	the table the DELETE is to remove rows from. Used when the DELETE includes a condition that directly references more than one table.
table_name	the name of a derived table or a joined table in the subquery referenced within the *condition* variable of the WHERE clause.
FROM *table_name*	the name of the table from which rows are to be deleted. This phrase is optional.
correlation_name	an optional alias name, used for self-joins. ANSI calls table aliases correlation names. They are also referred to as range variables.
condition	the conditional expression to be used for determining rows to be deleted. The expression operands can be constants, or references to fields in the specified table or other tables. The conditional also can specify an embedded SELECT operation. See "WHERE Clause" on page 1-32.
ALL	that all rows to be deleted, resulting in an empty table. This is the default and is used when a WHERE condition is not specified.

ANSI Compliance

The join condition form of DELETE is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must have the DELETE privilege on the table.

Use caution when granting the privilege to delete data through a view. Data in fields that might not be known to the user, is also deleted when a row is deleted through a view.

Related Information

See "DELETE" on page 3-36.

DELETE (Positioned Form)

Purpose

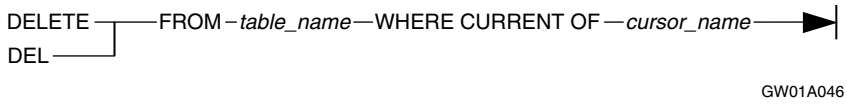
Deletes the most current fetched row from an updatable cursor invocation.

Invocation

Executable.

Embedded SQL and stored procedures only.

Syntax



where:

Syntax element ...	Specifies ...
table_name	the name of the table in which the row to be deleted is found.
cursor_name	the name of the cursor to be used to position to the row to be deleted.

ANSI Compliance

The positioned form of DELETE is ANSI SQL-99-compliant

This form is not valid in Teradata mode.

Authorization

You must have the DELETE privilege on the table.

Use caution when granting the privilege to delete data through a view. Data in fields that might not be known to the user, is also deleted when a row is deleted through a view.

Rules

The following rules apply to the positioned form of DELETE:

- The preprocessor TRANSACT or -tr option must be set to ANSI.
- The WHERE CURRENT OF clause enables a DELETE statement to act on a row currently pointed to by the cursor named in WHERE CURRENT OF *cursor_name*.

Restrictions

The following restrictions apply to usage of the WHERE CURRENT OF feature in DELETE statements:

- *cursor_name* must be a valid updatable cursor.
- Multiple updates of the current row of cursor or updates followed by a delete of the current row of cursor are allowed.
- *table_name* must be the same table as that SELECTed in the updatable cursor request.
- The referenced cursor must be positioned at a valid row via the FETCH statement.

Example

In this example, the name of the cursor being used to delete from the table is `X01`.

```
EXEC SQL
DELETE FROM CUSTOMER
WHERE CURRENT OF X01;
```

Related Information

See "DELETE" on page 3-36.

ECHO

Purpose

Returns a fixed character string to the requestor.

Invocation

Executable.

Syntax



where:

Syntax Element . . .	Specifies . . .
'string'	the text to be returned to the requestor.
'command'	a BTEQ format command to be returned to the requestor. The format command must be terminated by a semicolon.

ANSI Compliance

ECHO is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

ECHO Not Supported for Embedded SQL

The Teradata preprocessor does not support ECHO.

How ECHO Works

When Teradata encounters an ECHO statement, it sends an ECHO parcel containing *string* to the client system, where it is then interpreted.

ECHO and BTEQ

ECHO is used most frequently in macros to perform BTEQ commands during macro execution.

ECHO cannot be used with the following BTEQ commands.

- =
- EXPORT
- IMPORT
- LOGOFF
- LOGON
- QUIT
- REPEAT
- RUN

If *command* is not a BTEQ format command, BTEQ logs the illegal command as an error.

Example

The following ECHO statement could be used in a macro that creates a report through BTEQ. When Teradata encounters the statement during macro execution, the width of a printed page is set to 72 characters.

```
… ECHO 'SET WIDTH 72;'…
```

END TRANSACTION

Purpose

Defines the end of a single logical transaction.

Invocation

Executable.

Syntax



ANSI Compliance

END TRANSACTION is a Teradata extension to the ANSI SQL-99 standard and is valid only in Teradata mode.

For ANSI mode transaction control statements, see "COMMIT" on page 3-33 and "ROLLBACK" on page 3-141.

Authorization

None.

Usage Notes

See "BEGIN TRANSACTION" on page 3-7 for additional usage information about END TRANSACTION.

Use BEGIN TRANSACTION (see "BEGIN TRANSACTION" on page 3-7) to initiate a transaction.

While the Teradata END TRANSACTION statement terminates a transaction with commit, the ANSI statements ABORT and ROLLBACK terminate a transaction with rollback (see "ABORT" on page 3-3 and "ROLLBACK" on page 3-141).

In general, the rules for committing transactions using the COMMIT statement (see "COMMIT" on page 3-33) also apply to END TRANSACTION.

Rules for Embedded SQL

The following rules apply to the use of END TRANSACTION in embedded SQL.

- END TRANSACTION is valid only when you specify the TRANSACT(BTET) or -tr(BTET) options to the preprocessor. Otherwise, an error is returned and the precompilation fails.
- END TRANSACTION cannot be performed as a dynamic SQL statement.
- When END TRANSACTION is processed, the transaction, if not already aborted, is committed.

Related Information

See "BEGIN TRANSACTION" on page 3-7.

EXECUTE (Macro Form)

Purpose

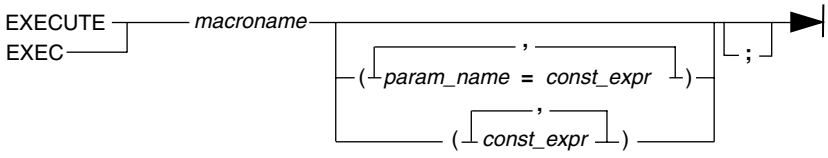
Performs a macro.

Invocation

Executable.

Interactive SQL only.

Syntax



FF07A023

where:

Syntax Element . . .	Specifies . . .
macro_name	the name of the macro that is to be executed.
constant_expression	a constant or an expression involving constants that specifies a parameter value. Values are listed in left-to-right order according to the order in which parameter names were established in the CREATE MACRO statement. You can specify nulls by typing successive COMMA characters, for example, (a, , b) This is referred to as a positional parameter list. If a value is not supplied for a parameter, a comma must be entered to account for the missing parameter. Any default value defined for that parameter is inserted automatically.
parameter_name = *constant_expression*	a parameter name (*param_name*) as defined in the CREATE MACRO statement, and supplies the value for that parameter. This is referred to as a named parameter list. The value can be a constant, or an expression involving constants. If a parameter is omitted, any default value defined for that parameter is inserted automatically.

ANSI Compliance

EXECUTE is a Teradata extension to the ANSI SQL-99 standard.

Authorization

You must have EXECUTE privilege on the macro. The creator or any owner of the macro can grant the EXECUTE privilege to another. In addition, the immediate owner of the macro (the database in which the macro resides) must have the necessary privileges on objects named in the statement set that are contained in the macro.

Recommendation

A data definition statement in a macro is not fully resolved until the macro is submitted for execution. Unqualified references to database objects are resolved at that time using the default database of the executing user.

Because of this, object references in data definition statements should always be fully qualified (as databasename.tablename) in the body of the macro.

Rules for Performing Macros

The following rules apply to performing macros.

- The number of commas must match the macro definition.
- Any value in the EXECUTE parameter list can be a constant or an expression involving constants. In this context, DATE, TIME, and USER are considered constants.
- If an error message is returned when a macro is executed, it can be caused by an SQL statement in the macro.
- The number of parameters used in the calling sequence must be equal to the number of parameters defined.
- When, for example, two parameters are defined and used, if they are both null, the following statements are all valid, unless defaults have been used in the macro definition:

```
EXECUTE MACRO '(, 1);
EXECUTE MACRO (,);
EXECUTE MACRO (NULL, NULL);
```

Access Logging and Errors

Any syntactic or semantic error identified and reported by the SQL parser results in an error being returned to you without logging the statement.

Examples

The following examples illustrate the use of EXECUTE:

Example 1

The following statement uses a named parameter list to execute macro NewEmp1 (see "CREATE MACRO" in the chapter "SQL Data Definition Language Statement Syntax" of *Teradata RDBMS SQL Reference, Volume 4*). Note that named parameters can be listed in any order.

```
EXEC NewEmp1(number=10015, dept=500, name='Omura H', sex='M',
position='Programmer');
```

The row for new employee Omura is inserted in the Employee table.

Example 2

This example uses a positional parameter list to execute macro NewEmp2 (see "CREATE MACRO" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*). Note that a value is not specified for the dept parameter, which has a macro-defined default value. A comma is entered in the dept position to maintain the integrity of the positional sequence.

```
EXEC NewEmp2 (10021, 'Smith T', , 'Manager', 'F', 'May 8,
1959', 16);
```

Example 3

When the following statement is processed, the default value for the dept parameter (900) is inserted automatically. The row for new employee Smith is added to the Employee table, and then the Department table is updated by incrementing the value for Department 900 in the EmpCount column. The statement uses a named parameter list to execute a macro named NewHire. Note that the value of the DOH (Date of Hire) column is an expression involving the DATE constant.

```
EXEC NewHire (FlName='Toby Smith', Title='Programmer',
DOH=DATE -1);
```

EXPLAIN Modifier

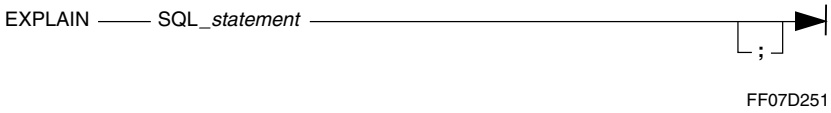
Purpose

Reports a summary of the plan generated by the SQL query optimizer: the steps Teradata would perform to resolve a request. The request itself is not processed.

Invocation

Executable.

Syntax



FF07D251

where:

Syntax element …	Specifies …
SQL_statement	the text of an SQL statement about which Optimizer processing information is desired.

ANSI Compliance

EXPLAIN is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

EXPLAIN Report Overview

EXPLAIN is an extremely useful utility for query designers because it provides a summary of the access and join plans generated by the Optimizer for the query *SQL_statement*: the report details which indexes would be used to process the request, identifies any intermediate spool files that would be generated, indicates the types of join to be performed, shows whether the statements in a transaction would be dispatched in parallel, and so on.

When you perform an EXPLAIN against any SQL statement, that statement is parsed and optimized, and the parse tree (access and join plans) generated by the Optimizer is returned to the requestor in the form of a text file that explains the parallel steps taken in the optimization of the statement as well as the relative time it would take to complete the statement given the statistics the Optimizer had to work with.

EXPLAIN helps you to evaluate complex queries and to develop alternative, more efficient, processing strategies.

References to bit mapping might appear when complex conditional expressions involving nonunique secondary indexes are applied to a very large table.

Such expressions can be resolved by mapping each subtable row ID to a number in the range 0-32767. This number is used as an index into a bit map in which the bit for each qualifying data row is set equal to 1.

The Optimizer is better able to determine whether the table is a candidate for bit mapping when up-to-date statistics exist for the important columns of that table. See "COLLECT STATISTICS" in the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4*.

Only the first 255 characters of a conditional expression are displayed in an EXPLAIN. The entire conditional expression is enclosed in quotation marks.

Although the times shown in the EXPLAIN output are indicated as seconds, they are actually arbitrary units of time. They are valuable because they permit you to compare alternate coding formulations of the same query with respect to their relative performance.

Keep EXPLAIN results with your system documentation because they can be of value when you reevaluate your database design.

Visual EXPLAIN

The Visual EXPLAIN tool provides a graphic display of Optimizer plans and also permits you to compare the plans for queries that return the same result. This feature greatly simplifies the interpretation of EXPLAIN reports.

For more information about the Visual EXPLAIN utility, see *Teradata Visual Explain User Guide*. Related information can be found in the chapter "Query Capture Facility" in *Teradata RDBMS SQL Reference, Volume 2*.

EXPLAIN and Embedded SQL

Use EXPLAIN only with data returning statements.

EXPLAIN returns multiple rows of a single column whose data type is VARCHAR(80). Because it causes multiple rows to be returned, EXPLAIN can only be performed with a cursor.

For static execution of EXPLAIN, see "DECLARE CURSOR (Selection Form)" on page 7-26, substituting an EXPLAIN-modified SQL statement for the *query_expression* clause shown there.

Do not perform EXPLAIN dynamically because the result is unpredictable.

EXPLAIN and Stored Procedures and Macros

Do not use EXPLAIN with macros or compiled stored procedures.

Do use EXPLAIN while building your macros and stored procedures.

EXPLAIN works by parsing and optimizing SQL source code text just as it would be parsed and optimized if it were submitted by a client application for processing. You EXPLAIN a string of SQL text, not the name of an object that contains strings of SQL text. Once a macro has been created or a stored procedure has been compiled, it is a database object and it can no longer be processed using an EXPLAIN modifier as if it were simple SQL text.

For example, if you compile a stored procedure under the name UpdateOrders and then perform the following EXPLAIN, it will fail because UpdateOrders is not a valid SQL statement.

```
EXPLAIN UpdateOrders;
```

While you are developing the SQL statements that constitute the UpdateOrders procedure, however, you should always use EXPLAINs to help you to write the most efficient SQL code possible.

Standard Form of Display for EXPLAIN

EXPLAIN displays character constant strings as follows.

FOR this type of constant . . .	The value is displayed in this format . . .
printable single byte character	Teradata Latin
anything else	internal Teradata hexadecimal.

Secondary Indexes

References to secondary indexes are based on the following:

- Each secondary index is stored as a subtable.
- Each row in a secondary index subtable is made up of an index value and one or more row IDs.
- The subtable row IDs identify the data rows that contain the index value.
- The subtable for a unique secondary index is a hashed table.

2PC Mode

In 2PC mode, if EXPLAIN is used with a request, and if that request is followed by a 2PC function, the SQL portion of the request is explained, but the 2PC function is not.

For example, if the following request were submitted:

```
EXPLAIN INSERT tab1(1,2);SELECT * FROM tab1; VOTE
```

the INSERT and SELECT would be explained; the VOTE would not be explained. The VOTE is, however, sent to the AMPs.

VOTE is not specified by users—it is specified by the system administrator.

EXPLAIN Modifier Terminology

Terms used in EXPLAIN phrases are described in the following list:

Phrase	Explanation
***n* partitions of …**	
	Only *n* of the partitions are accessed. *n* is greater than one.
a single partition of …	
	Only a single partition is accessed.
all-AMPs JOIN step by way of an all-rows scan …	
	On each AMP on which they reside, spooled rows and/or primary table rows are searched row by row; rows that satisfy the join condition are joined.
all-AMPs JOIN step by way of a RowHash match scan …	
	1 The first row is retrieved from the first table; the hash code for that row is used to locate a row from the second table. 2 Each row hash match is located and processed as follows: 1 The row hashes are compared. If not equal, the larger row hash is used to read rows from the other table until a row hash match is found, or until the table is exhausted. 2 If match is found, each pair of rows with the same hash code is accessed (one at a time) from the two tables. For each such pair, if the join condition is satisfied, a join result row is produced. 3 After all rows with the same row hash are processed from both tables, one more row is read from each table. The row hashes from these two rows are compared, restarting the compare process.
all-AMPs RETRIEVE step by way of an all-rows scan …	
	All rows of a table are selected row by row on all AMPs on which the table is stored. BMSMS (bit map set manipulation step); intersects the following row id bit maps: 1) The bit map built for ... by way of index # *n*... 2) The bit map built for ... by way of index # *n*... … The resulting bit map is placed in Spool *n*... BMSMS... Indicates that two or more bit maps are intersected by ANDing them to form one large bit map. index # *n*... Identifies, in the order in which they are ANDed, each nonunique secondary index used in the intersection. resulting bit map is placed in Spool *n*... Identifies the temporary file in which the large bit map produced by the BMSMS is stored to make it available for use in producing the final result.
all partitions of …	
	All partitions are accessed for a primary index access.

Phrase	Explanation
a rowkey-based …	
	The join is hash based by partition (that is, by the rowkey). In this case, there are equality constraints on the partitioning columns and primary index columns. This allows for a faster join since each non-eliminated partition needs to be joined with at most only one other partition. When the phrase is not given, the join is hash based. That is, there are equality constraints on the primary index columns from which the hash is derived. For a partitioned table, there is some additional overhead in processing the table in hash order. Note that with either method, the join conditions must still be validated.
by way of index # n and the bit map in Spool n …	
	The data row associated with the row ID is accessed only if the associated bit is turned on in the bit map (see Usage Notes).
by way of the sort key in spool field1 …	
	Field1 is created to allow a tag sort.
computed globally …	
	The computation involves all the intermediate spool file data.
condition …	
	An intermediate condition that joins table rows (as compared with the overall join condition).
duplicated on all AMPs …	
	A spool file containing intermediate data that is used to produce a result is copied to all AMPs containing data with which the intermediate data is compared.
eliminating duplicate rows …	
	Duplicate rows can exist in spool files, either as a result of selection of nonunique columns from any table or of selection from a MULTISET table. This is a DISTINCT operation.
END TRANSACTION step …	
	This indicates that processing is complete and that any locks on the data may be released. Changes made by the transaction are committed.
estimated size …	
	This value, based on any statistics collected for a table, is used to estimate the size of the spool file needed to accommodate spooled data. If statistics have not been collected for a table, this estimate may be grossly incorrect (see "COLLECT STATISTICS" in the chapter "Query and Workload Analysis Statements" in *Teradata RDBMS SQL Reference, Volume 4*).
estimated time …	
	This approximate time is based on average times for the suboperations that comprise the overall operation, and the likely number of rows involved in the operation. The accuracy of the time estimate is also affected by the accuracy of the estimated size.

Phrase	Explanation
execute the following steps in parallel ...	
	This identifies a set of steps that are processed concurrently. The explanatory text immediately following the list describes the execution of each step.
join condition ...	
	The overall constraint that governs the join. In the following statement, "Employee.EmpNo = Department.MgrNo" is the overall constraint governing the join. SELECT DeptName, Name FROM Employee, Department WHERE Employee.EmpNo = Department.Mgrno;
last use ...	
	This term identifies the last reference to a spool file that contains intermediate data that produces a statement's final result. The file is released following this step
locally on the AMPs ...	
	That portion of spooled intermediate or result data for which an AMP is responsible is stored on the AMP; it is not duplicated on or redistributed to other AMPs that are processing the same request.
lock ...	
	Teradata places an ACCESS, READ, WRITE, or EXCLUSIVE lock on the database object that is to be accessed by a request.
merge join ...	
	One of the types of join processing performed by Teradata.
nested join ...	
	One of the types of join processing performed by Teradata
no residual conditions ...	
	Rows are selected in their entirety; there are no specific search conditions. All applicable conditions have been applied to the rows.
of *n* partitions ...	
	The optimizer was able to determine that all rows in each of n partitions may be completely deleted. *n* is greater than one. In some cases, this allows for faster deletion of entire partitions.
of a single partition ...	
	The optimizer was able to determine that all rows in a single partition may be completely deleted. In some cases, this allows for faster deletion of the entire partition.
product join ...	
	One of the types of join processing performed by Teradata.

Phrase	Explanation
redistributed by hash code to all AMPs …	
	Given the values of an indexed or nonindexed column, rows are sent to the AMPs that are responsible for storing the rows that use these values as a primary index.
single-AMP JOIN step by way of the unique primary index …	
	A row is selected on a single AMP using the unique primary index for the table. Using a value in the row that hashes to a unique index value in a second table, the first row is joined with a second row located on another AMP.
single-AMP RETRIEVE by way of unique index # n …	
	A single row of a table is selected using a unique secondary index value that hashes to the AMP on which the table row is stored.
single-AMP RETRIEVE step by way of the unique primary index …	
	A single row of a table is selected using a unique primary index value that hashes to the single AMP on which the data row is stored. Although not explicitly stated in the LOCK portion of the Explain text, a rowhash lock is required because the table is accessed via the unique primary index.
SORT to order Spool n by row hash …	
	Rows in the spool file are sorted by hash code to put them the same order as rows in the primary table, or in another spool file on the same AMP, with which they are to be matched
SORT to order Spool n by row hash and the sort key in spool field1 eliminating duplicate rows …	
	Rows in the spool file are sorted by hash code using a uniqueness sort to eliminate duplicate rows. Uniqueness is based on the data in field1. The contents of field1 depend on the query and may comprise any of the following: • A concatenation of all the fields in the spool row (used for queries with SELECT DISTINCT or that involve a UNION, INTERSECT, EXCEPT, or MINUS operation). • A concatenation of the row IDs that identify the data rows from which the spool row was formed (used for complex queries involving subqueries). Some other value that uniquely defines the spool row (used for complex queries involving aggregates and subqueries).
SORT to order Spool 1 by the sort key in spool field1 …	
	Rows in the spool file are sorted by the value in field1. The contents of field1 are determined by the column or columns defined in the ORDER BY or WITH...BY clause of the request being processed.
SORT to partition Spool n by rowkey …	
	The optimizer determined that a spool file is to be partitioned based on the same partitioning expression as a table to which the spool file is be joined. That is, the spool is to be sorted by rowkey (partition and hash). Partitioning the spool file in this way allows for a faster join with the partitioned table. *n* is the spool number.

Phrase	Explanation
spool n ...	
	Identifies a spool file, which is a temporary file used to contain data during an operation. • Spool 1 is normally used to hold a result before it is returned to the user. • Spools 2, 3, etc., contain intermediate data that produces a result.
spool_n.Field ...	
	Identifies the field of the spooled rows that is being used in a join constraint or comparison operation. For example: PERSONNEL.Employee.EmpNo = Spool_2.MgrNo Statement 1... This term refers to the initiating request.
the estimated time is *nn* seconds.	
	The time shown is not clock seconds; you should consider it to be an arbitrary unit of measure that compares different operations.
two-AMP RETRIEVE step by way of unique index #n ...	
	A row of a table is selected based on a unique secondary index: • A single row in the unique secondary index subtable is selected using the index value that hashes to the AMP on which the subtable row is stored. • The hash value in the index row ID determines the AMP on which the data row is stored.
we do a BMSMS... (bit map set manipulation step)	
	BMSMS is a method for handling weakly selective secondary indexes that have been ANDed; NUSI bit mapping.
we do an ABORT test ...	
	An ABORT or ROLLBACK statement was detected.
we do a SMS (set manipulation step) ...	
	Combine rows under control of a UNION, EXCEPT, MINUS, or INTERSECT operator.
which is duplicated on all AMPs ...	
	Relocating data in preparation for a join.
which is redistributed by hash code to all AMPs ...	
	Relocating data in preparation for a join.

EXPLAIN Modifier in Greater Depth

This section examines the usefulness of the EXPLAIN modifier in different situations.

You should always use EXPLAINs to analyze any new queries under development. Subtle differences in the way a query is structured can produce enormous differences in its resource impact and performance while at the same time producing the identical end result.

EXPLAIN modifier terms are defined in "EXPLAIN Modifier" on page 3-55.

The following topics are described here.

- "EXPLAIN: Examples of Complex Queries" on page 3-65
- "EXPLAIN and Join Processing" on page 3-71
- "EXPLAIN and Standard Indexed Access" on page 3-77
- "EXPLAIN and Parallel Steps" on page 3-80
- "EXPLAIN and Partitioned Primary Index Access" on page 3-82
- "EXPLAIN and MERGE Conditional Steps" on page 3-90
- "EXPLAIN and UPDATE (Upsert Form) Conditional Steps" on page 3-97

EXPLAIN: Examples of Complex Queries

Introduction

The Personnel.Employee table has a unique primary index defined on the EmpNo column and a nonunique secondary index defined on the Name column.

Example 1: SELECT

The EXPLAIN modifier generates the following response for this request.

```
EXPLAIN
SELECT Name, DeptNo
FROM Employee
WHERE EmpNo = 10009;
```

```
Explanation
-----------------------------------------------------
1) First, we do a single-AMP RETRIEVE step from Personnel.Employee by way of the
unique primary index "PERSONNEL.Employee.EmpNo = 10009" with no residual
conditions. The estimated time for this step is 0.04 seconds.
-> The row is sent directly back to the user as the result of statement 1. The
total estimated time is 0.04 seconds.
```

Example 2: SELECT with WHERE on Nonunique Index

The WHERE condition in this example is based on a column that is defined as a nonunique index. Note that Teradata places a READ lock on the table.

The EXPLAIN modifier generates the following response for this request.

```
EXPLAIN
SELECT EmpNo, DeptNo
FROM Employee
WHERE Name = 'Smith T';
```

```
Explanation
---------------------------------------------------------------------
1) First, we lock a distinct PERSONNEL."pseudo table" for read on a RowHash to
prevent global deadlock for PERSONNEL.employee.
2) Next, we lock PERSONNEL.employee for read.
3) We do an all-AMPs RETRIEVE step from PERSONNEL.employee by way of an all-rows
scan with a condition of ("PERSONNEL.employee.Name = 'Smith t'") into Spool 1,
which is built locally on the AMPs. The size of Spool 1 is estimated to be 2 rows.
The estimated time for this step is 0.03 seconds.
->The contents of Spool 1 are sent back to the user as the result of statement 1.
The total estimated time is 0 hours and 0.03 seconds.
```

Example 3: SELECT with WHERE on Unique Secondary Index

Assume that the Employee table has another column (SocSecNo), where SocSecNo is defined as a unique secondary index.

If the WHERE condition is based on this column, then the EXPLAIN modifier generates the following response for this request.

```
EXPLAIN
SELECT Name, EmpNo
FROM Employee
WHERE SocSecNo = '123456789';
```

```
Explanation
-----------------------------------------------------
1) First, we do a two-AMP RETRIEVE step from PERSONNEL.Employee by way of unique
index # 20 "PERSONNEL.Employee.socSecNo = 123456789" with no residual conditions.
The estimated time for this step is 0.09 seconds.
-> The row is sent directly back to the user as the result of statement 1. The
total estimated time is 0.09 seconds.
```

Example 4: SELECT With WHERE Based On a Join

In this example, the WHERE clause defines an equality constraint that governs a join.

The rows of the Department table are copied to a spool file for use in the join operation.

The EXPLAIN modifier generates the following response for this request.

```
EXPLAIN
SELECT DeptName, Name
FROM Employee, Department
WHERE Employee.EmpNo = Department.MgrNo ;
```

```
Explanation
---------------------------------------------------------------------
1) First, we lock a distinct PERSONNEL."pseudo table" for read on a RowHash to
prevent global deadlock for PERSONNEL.department.
2) Next, we lock a distinct PERSONNEL."pseudo table" for read on a RowHash to
prevent global deadlock for PERSONNEL.employee.
3) We lock PERSONNEL.department for read, and we lock PERSONNEL.employee for read.
4) We do an all-AMPs RETRIEVE step from PERSONNEL.department by way of an all-rows
scan with no residual conditions into Spool 2, which is redistributed by hash code
to all AMPs. Then we do a SORT to order Spool 2 by row hash.  The size of Spool 2
is estimated to be 8 rows.  The estimated time for this step is 0.11 seconds.
5) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of a RowHash match
scan, which is joined to PERSONNEL.employee. Spool 2 and PERSONNEL.employee are
joined using a merge join, with a join condition of ("PERSONNEL.employee.EmpNo =
Spool_2.MgrNo"). The result goes into Spool 1, which is built locally on the AMPs.
The size of Spool 1 is estimated to be 8 rows.  The estimated time for this step
is 0.07 seconds.
-> The contents of Spool 1 are sent back to the user as the result of statement
1.  The total estimated time is 0 hours and 0.19 seconds.
```

Example 5: SELECT With WHERE Based on Subquery

In this example, the constraint that governs the join is defined by a subquery predicate and the ORDER BY clause specifies a sorted result.

The EXPLAIN modifier generates the following response for this request.

```
EXPLAIN
SELECT Name, EmpNo
FROM Employee
WHERE EmpNo IN
 (SELECT EmpNo
  FROM Charges)
ORDER BY Name ;
```

```
Explanation
---------------------------------------------------------------------
1) First, we lock a distinct PERSONNEL."pseudo table" for read on a RowHash to
prevent global deadlock for PERSONNEL.charges.
2) Next, we lock a distinct PERSONNEL."pseudo table" for read on a RowHash to
prevent global deadlock for PERSONNEL.employee.
3) We lock PERSONNEL.charges for read, and we lock PERSONNEL.employee for read.
4) We do an all-AMPs RETRIEVE step from PERSONNEL.charges by way of an all-rows
scan with no residual conditions into Spool 2, which is redistributed by hash code
to all AMPs.  Then we do a SORT to order Spool 2 by row hash and the sort key in
spool field1 eliminating duplicate rows.  The size of Spool 2 is estimated to be
12 rows.  The estimated time for this step is 0.15 seconds.
5) We do an all-AMPs JOIN step from PERSONNEL.employee by way of an all-rows scan
with no residual conditions, which is joined to Spool 2 (Last Use).
PERSONNEL.employee and Spool 2 are joined using an inclusion merge join, with a
join condition of ("PERSONNEL.employee.EmpNo = Spool_2.EmpNo"). The result goes
into Spool 1, which is built locally on the AMPs.  Then we do a SORT to order Spool
1 by the sort key in spool field1.  The size of Spool 1 is estimated to be 12 rows.
The estimated time for this step is 0.07 seconds.
  -> The contents of Spool 1 are sent back to the user as the result of statement
1.  The total estimated time is 0 hours and 0.23 seconds.
```

Example 6: Large Table SELECT with More Complex Condition

Assume that a table named Main is very large and that its columns named NumA, NumB, Kind, and Event are each defined as nonunique secondary indexes.

The request in this example uses these indexes to apply a complex conditional expression.

The EXPLAIN modifier generates the following response for this request.

```
EXPLAIN
SELECT COUNT(*)
FROM Main
WHERE NumA = '101'
AND NumB = '02'
AND Kind = 'B'
AND Event = '001';
```

The response indicates that bit mapping would be used.

```
Explanation -----------------------------------------------------------
1) First, we lock TESTING.Main for read.
2) Next, we do a BMSMS (bit map set manipulation) step that intersects the
following row id bit maps:
 1) The bit map built for TESTING.Main by way of index # 12 "TESTING.Main.Kind =
'B'".
 2) The bit map build for TESTING.Main by way of index # 8 "TESTING.Main.NumB =
'02'".
 3) The bit map built for TESTING.Main by way of index # 16 "TESTING.Main.Event =
'001'".
The resulting bit map is placed in Spool 3. The estimated time for this step is
17.77 seconds.
3) We do a SUM step to aggregate from TESTING.Main by way of index # 4
"TESTING.Main.NumA = '101'" and the bit map in Spool 3 (Last Use) with a residual
condition of ("(TESTING.Main.NumB = '02') and ((TESTING.Main.Kind = 'B') and
(TESTING.Main.Event = '001'))"). Aggregate Intermediate Results are computed
globally, then placed in Spool 2.
4) We do an all-AMPs RETRIEVE step from Spool 2 (Last Use) by way of an all-rows
scan into Spool 1, which is built locally on the AMPs. The size of Spool 1 is
estimated to be 20 rows. The estimated time for this step is 0.11 seconds.
5) Finally, we send out an END TRANSACTION step to all AMPs involved in processing
the request.
-> The contents of Spool 1 are sent back to the user as a result of statement 1.
```

Example 7: Implicit Multistatement Transaction, INSERT

In the following BTEQ multistatement request, which is treated as an implicit transaction, the statements are processed concurrently.

In Teradata mode, the EXPLAIN modifier generates the following response for this request.

```
EXPLAIN
INSERT Charges (30001, 'AP2-0004', 890825, 45.0)
; INSERT Charges (30002, 'AP2-0004', 890721, 12.0)
; INSERT Charges (30003, 'AP2-0004', 890811, 2.5)
; INSERT Charges (30004, 'AP2-0004', 890831, 37.5
; INSERT Charges (30005, 'AP2-0004', 890825, 11.0)
; INSERT Charges (30006, 'AP2-0004', 890721, 24.5)
; INSERT Charges (30007, 'AP2-0004', 890811, 40.5)
; INSERT Charges (30008, 'AP2-0004', 890831, 32.0
; INSERT Charges (30009, 'AP2-0004', 890825, 41.5)
; INSERT Charges (30010, 'AP2-0004', 890721, 22.0) ;

Explanation -----------------------------------------------------------
1) First, we execute the following steps in parallel.
    1)    We do an INSERT into PERSONNEL.charges.
    2)    We do an INSERT into PERSONNEL.charges.
    3)    We do an INSERT into PERSONNEL.charges.
    4)    We do an INSERT into PERSONNEL.charges.
    5)    We do an INSERT into PERSONNEL.charges.
    6)    We do an INSERT into PERSONNEL.charges.
    7)    We do an INSERT into PERSONNEL.charges.
    8)    We do an INSERT into PERSONNEL.charges.
    9)    We do an INSERT into PERSONNEL.charges.
    10)   We do an INSERT into PERSONNEL.charges.
2) Finally, we send out an END TRANSACTION step to all
   AMPs involved in processing the request.
-> No rows are returned to the user as the result of
     statement 1.
     No rows are returned to the user as the result of
     statement 2.
     No rows are returned to the user as the result of
     statement 3.
     No rows are returned to the user as the result of
     statement 4.
     No rows are returned to the user as the result of
     statement 5.
     No rows are returned to the user as the result of
     statement 6.
     No rows are returned to the user as the result of
     statement 7.
     No rows are returned to the user as the result of
     statement 8.
     No rows are returned to the user as the result of
     statement 9.
     No rows are returned to the user as the result of
     statement 10.
```

Example 8: ANSI Versus Teradata Mode

This example shows the EXPLAIN differences between running the session in ANSI versus Teradata mode:

```
EXPLAIN
UPDATE Employee
SET deptno = 650
WHERE deptno = 640;
```

In ANSI mode, EXPLAIN generates the following response for this request:

```
Explanation
------------------------------------------------------------------------
1) First, we lock a distinct PERSONNEL."pseudo table" for write
on a RowHash to prevent global deadlock for PERSONNEL.employee.
2) Next, we lock PERSONNEL.employee for write.
3) We do an all-AMPs UPDATE from PERSONNEL.employee by way of an
    all-rows scan with a condition of ("PERSONNEL.employee.DeptNo = 640").
  -> No rows are returned to the user as the result of statement 1.
```

In Teradata mode, EXPLAIN generates this response for the same request.

```
Explanation
------------------------------------------------------------------------
1) First, we lock a distinct PERSONNEL."pseudo table" for write on a RowHash to
prevent global deadlock for PERSONNEL.employee.
2) Next, we lock PERSONNEL.employee for write.
3) We do an all-AMPs UPDATE from PERSONNEL.employee by way of an all-rows scan
with a condition of ("PERSONNEL.employee.DeptNo = 640").
4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing
the request.
  -> No rows are returned to the user as the result of statement 1.
```

In ANSI mode the transaction is not committed, therefore it is not ended, whereas in Teradata mode, no COMMIT is required to end the transaction.

EXPLAIN and Join Processing

Introduction

The following descriptions are generated by EXPLAIN when applied to sample join requests.

Each explanation is preceded by the request syntax and is followed by a listing of any new terminology found in the display.

Example 1: Product Join

This request has a WHERE condition based on the value of a unique primary index, which produces efficient processing even though a product join is used (compare with the merge join used to process the request in "Example 2: Merge Join" on page 3-72).

```
EXPLAIN
SELECT Hours, EmpNo, Description
FROM Charges, Project
WHERE Charges.Proj_Id = 'ENG-0003'
AND Project.Proj_Id = 'ENG-0003'
AND Charges.WkEnd > Project.DueDate ;
```

This request returns the following EXPLAIN report:

```
Explanation -----------------------------------------------------------
1) First, we lock PERSONNEL.Charges for read.
2) Next, we do a single-AMP RETRIEVE step from PERSONNEL.Project by way of the
unique primary index "PERSONNEL.Project.Proj_Id = 'ENG-003'" with no residual
conditions into Spool 2, which is duplicated on all AMPs. The size of Spool 2 is
estimated to be 4 rows. The estimated time for this step is 0.07 seconds.
3) We do an all AMPs JOIN step from Spool 2 (Last Use) by way of an all-rows scan,
which is joined to PERSONNEL.Charges with a condition of
("PERSONNEL.Charges.Proj_Id = 'ENG-0003'"). Spool 2 and PERSONNEL.Charges are
joined using a product join, with a join condition of ("PERSONNEL.Charges.WkEnd >
Spool_2.DueDate"). The result goes into Spool 1, which is built locally on the
AMPs. The size of Spool 1 is estimated to be 6 rows. The estimated time for this
step is 0.13 seconds.
4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing
the request.
-> The contents of Spool 1 are back to the user as the result of statement 1. The
total estimated time is 0.20 seconds.
```

Terminology

New terminology in this explanation is defined as follows:

Phrase	Definition
single-AMP RETRIEVE step from ... by way of the unique primary index	A single row of a table is selected using a unique primary index that hashes to the single AMP on which the row is stored.
duplicated on all AMPs	The contents of a spool file, selected from the first table involved in the join, is replicated on all the AMPs that contain another table involved in the join.
all-AMPs JOIN step ... by way of an all-rows scan	Table rows are searched row by row on each AMP on which they reside. Rows that satisfy the join condition are joined to the spooled row or rows.
condition of ...	An intermediate condition used to qualify the joining of selected rows with spooled rows (as compared with an overall join condition).
product join	One of the types of join processing performed by Teradata. See "Product Join" in the chapter "Query Optimization" in *Teradata RDBMS SQL Reference, Volume 2*.

Example 2: Merge Join

This request returns the following EXPLAIN report:

```
EXPLAIN
SELECT Name, DeptName, Loc
FROM Employee, Department
WHERE Employee.DeptNo = Department.DeptNo ;
```

```
Explanation -----------------------------------------------------------
1) First, we lock PERSONNEL.Department for read, and we lock
PERSONNEL.Employee for read.
2) Next, we do an all-AMPs RETRIEVE step from PERSONNEL.Employee by way of an all-
rows scan with no residual conditions into Spool 2, which is redistributed by hash
code to all AMPs. Then we do a SORT to order Spool 2 by row hash. The size of Spool
2 is estimated to be 8 rows. The estimated time for this step is 0.10 seconds.
3) We do an all-AMPs JOIN step from PERSONNEL.Department by way of a RowHash match
scan with no residual conditions, which is joined to Spool 2 (Last Use).
PERSONNEL.Department and Spool 2 are joined using a merge join, with a join
condition of ("Spool_2.DeptNo = PERSONNEL.Department.DeptNo"). The  result goes
into Spool 1, which is built locally on the AMPs. The size of Spool 1 is estimated
to be 8 rows. The estimated time for this step is 0.11 seconds.
4) Finally, we send out an END TRANSACTION step to all AMPs
involved in processing the request.
-> The contents of Spool 1 are sent back to the user as the result of statement
1. The total estimated time is 0.22 seconds.
```

Terminology

New terminology in this explanation is defined as follows.

Phrase	Definition
merge join	One of the types of join processing performed by Teradata. See "Merge Join" in the chapter "Query Optimization" in *Teradata RDBMS SQL Reference, Volume 2*.

Example 3: Hash Join

This request returns the following EXPLAIN report:

```
EXPLAIN
SELECT Employee.EmpNum,Department.DeptName, Employee.Salary
FROM Employee, Department
WHERE Employee.Location = Department.Location ;

***Help information returned. 30 rows.
***Total elapsed time was 1 second.
```

```
Explanation -------------------------------------------

1) First, we lock a distinct PERSONNEL."pseudo table" for read on a RowHash to
prevent global deadlock for PERSONNEL.Department.
2) Next, we lock a distinct PERSONNEL."pseudo table" for read on a RowHash to
prevent global deadlock for PERSONNEL.Employee.
3) We lock PERSONNEL.Department for read, and we lock PERSONNEL.Employee for read.
4) We do an all-AMPs RETRIEVE step from PERSONNEL.Employee by way of an all-rows
scan with no residual conditions into Spool 2 fanned out into 22 hash join
partitions, which is redistributed by hash code to all AMPs. The size of Spool 2
is estimated to be 3,995,664 rows. The estimated time for this step is 3 minutes
and 54 seconds.
5) We do an all-AMPs RETRIEVE step from PERSONNEL.Department by way of an all-rows
scan with no residual conditions into Spool 3 fanned out into 22 hash join
partitions, which is redistributed by hash code to all AMPs. The size of Spool 3
is estimated to be 4,000,256 rows. The estimated time for this step is 3 minutes
and 54 seconds.
6) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an all-rows scan,
which is joined to Spool 3 (Last Use). Spool 2 and Spool 3 are joined using a hash
join of 22 partitions, with a join condition of ("Spool_2.Location =
Spool_3.Location"). The result goes into Spool 1, which is built locally on the
AMPs. The result spool field will not be cached in memory. The size of Spool 1 is
estimated to be 1,997,895,930 rows. The estimated time for this step is 4 hours
and 42 minutes.
7) Finally, we send out an END TRANSACTION step to all AMPs involved in processing
the request.
-> The contents of Spool 1 are sent back to the user as the result of statement
1. The total estimated time is 4 hours and 49 minutes.


DBS Control Record - Performance Fields:

HTMemAlloc       = 5%
SkewAllowance     = 75%
```

Terminology

New terminology in this explanation is defined as follows:

Phrase	Definition
hash join	One of the types of join processing performed by Teradata. See "Hash Join" in the chapter "Query Optimization" in *Teradata RDBMS SQL Reference, Volume 2*.

Example 4: Nested Join

This request returns the following EXPLAIN report:

```
   EXPLAIN
   SELECT DeptName, Name, YrsExp
   FROM Employee, Department
   WHERE Employee.EmpNo  = Department.MgrNo
   AND Department.DeptNo = 100;
Explanation -----------------------------------------------------

1) First, we do a single AMP JOIN step from PERSONNEL.Department by way of the
unique primary index "PERSONNEL.Department.DeptNo = 100" with no residual
condition which is joined to PERSONNEL.Employee by way of the unique primary index
"PERSONNEL.Employee.EmpNo = PERSONNEL.Department.MgrNo". PERSONNEL.Department and
PERSONNEL.Employee are joined using a nested join. The result goes into Spool 1,
which is built locally on that AMP. The size of Spool 1 is estimated to be 1 rows.
The estimated time for this step is 0.10 seconds.

> The contents of Spool 1 are sent back to the user as the result of statement 1.
The total estimated time is 0.10 seconds.
```

Terminology

New terminology in this explanation is defined as follows:

Phrase	Definition
single-AMP JOIN step from ... by way of the unique primary index	A single row on one AMP is selected using the value of a unique index defined for its table. Using the value in that row which hashes to a unique index value in a row of a second table on another AMP, the first row is joined with the second row. (Note that when a table is accessed by its unique primary index, the need for a rowhash lock on the table is implied, even though it is not explicitly stated in the Explain text.)
nested join	One of the types of join processing performed by Teradata. See "Nested Joins" in the chapter "Query Optimization" in *Teradata RDBMS SQL Reference, Volume 2*.

Example 5: Exclusion Merge Join

The request in this example selects columns only from the primary table.

If an additional column was selected from the table being joined via the embedded select (for example, if the request was “SELECT Name, DeptNo, Loc FROM Employee, Department”), the result would be a Cartesian product.

```
EXPLAIN
SELECT Name, DeptNo
FROM Employee
WHERE DeptNo NOT IN
 (SELECT DeptNo
  FROM Department
  WHERE Loc = 'CHI'
 )
ORDER BY Name ;
```

This request returns the following EXPLAIN report:

```
Explanation -----------------------------------------------------------
1) First, we lock PERSONNEL.Department for read, and we lock PERSONNEL.Employee
for read.
2) Next, we execute the following steps in parallel.
   1) We do an all-AMPs RETRIEVE step from PERSONNEL.Department by way of an all-
rows scan with a condition of ("PERSONNEL.Department.Loc = 'CHI'") into Spool 2,
which is redistributed by hash code to all AMPs. Then we do a SORT to order Spool
2 by row hash and the sort key in spool field1 eliminating duplicate rows. The
size of Spool 2 is estimated to be 4 rows. The estimated time for this step is
0.07 seconds.
   2) We do an all-AMPs RETRIEVE step from PERSONNEL.Employee by way of an all-
rows scan with no residual conditions into Spool 3, which is redistributed by hash
code to all AMPs. Then we do a SORT to order Spool 3 by row hash. The size of Spool
3 is estimated to be 8 rows. The estimated time for this step is 0.10 seconds.
3) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of an all-rows scan,
which is joined to Spool 2 (Last Use). Spool 3 and Spool 2 are joined using an
exclusion merge join, with a join condition of ("Spool_3.DeptNo =
Spool_2.DeptNo"). The result goes into Spool 1, which is built locally on the AMPs.
Then we do a SORT to order Spool 1 by the sort key in spool field1. The size of
Spool 1 is estimated to be 8 rows. The estimated time for this step is 0.13
seconds.
4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing
the request.
-> The contents of Spool 1 are sent back to the user as the result of statement
1. The total estimated time is 0.23 seconds.
```

Terminology

New terminology in this explanation is defined as follows.

Phrase	Definition
SORT to order Spool 2 by row hash and the sort key in spool field1 eliminating duplicate rows...	Rows in the spool file are sorted by hash code using a uniqueness sort to eliminate duplicate rows. Uniqueness is based on the data in field1. The contents of field1 depend on the query and may comprise any of the following: • A concatenation of all the fields in the spool row (used for queries with SELECT DISTINCT or that involve a UNION, INTERSECT, or MINUS operation). • A concatenation of the row IDs that identify the data rows from which the spool row was formed (used for complex queries involving subqueries). Some other value which uniquely defines the spool row (used for complex queries involving aggregates and subqueries).
SORT to order Spool 1 by the sort key in spool field1	This last sort is in response to the "ORDER BY" clause attached to the primary SELECT request.
exclusion merge join	One of the types of join processing performed by Teradata. See "Exclusion Merge Join" in the chapter "Query Optimization" in *Teradata RDBMS SQL Reference, Volume 2*.

EXPLAIN and Standard Indexed Access

Introduction

The EXPLAIN modifier is useful in determining whether the indexes defined for a table are properly defined, useful, and efficient.

As illustrated in the preceding join examples, EXPLAIN identifies any unique indexes that may be used to process a request.

When conditional expressions use nonunique secondary indexes, EXPLAIN also indicates whether the data rows are retrieved using spooling or bit mapping.

This feature is illustrated in the following examples. Compare the two requests and their corresponding EXPLAIN descriptions.

Note that in the first request the table is small (and that DeptNo, Salary, YrsExp, and Edlev have been defined as separate, nonunique indexes), so a full-table scan is used to access the data rows.

In the second request, however, the table is extremely large. Because of this, the Optimizer determines that bit mapping of the sub-table row IDs is the faster retrieval method.

Example 1: Full-Table Scan

This request returns the following EXPLAIN report:

```
   EXPLAIN
   SELECT COUNT(*)
   FROM Employee
   WHERE DeptNo = 500
   AND Salary > 25000
   AND YrsExp >= 3
   AND EdLev >= 12 ;
Explanation -----------------------------------------------------------
1) First, we lock PERSONNEL.Employee for read.
2) Next, we do a SUM step to aggregate from PERSONNEL.Employee by way of an all-
rows scan with a condition of
   ("(PERSONNEL.Employee.DeptNo = 500) AND
   ((PERSONNEL.Employee.Salary > 25000.00) AND
   ((PERSONNEL.Employee.YrsExp >= 3) AND
   (PERSONNEL.Employee.EdLev >= 12 )))").
Aggregate Intermediate Results are computed globally, then placed in Spool 2.
3) We do an all-AMPs RETRIEVE step from Spool 2 (Last Use) by way of an all-rows
scan into Spool 1, which is built locallyon the AMPs. The size of Spool 1 is
estimated to be 1 rows. The estimated time for this step is 0.06 seconds.
4) Finally, we send out an END TRANSACTION step to all AMPs involved in processing
the request.
-> The contents of Spool 1 are sent back to the user as the result of statement 1.
```

Terminology

New terminology in this explanation is defined as follows.

Phrase	Definition
SUM step to aggregate	The table is searched row by row, the qualifying rows are counted for each AMP on which they were found, and each count is held in a local spool file.
computed globally	The final computation involves all the intermediate spool-file data.

Example 2: Indexed Access With Bit Mapping

This request returns the following EXPLAIN report:

```
EXPLAIN
SELECT COUNT(*)
FROM Main
WHERE NumA = '101'
AND NumB  = '02'
AND Kind  = 'B'
AND Event = '001';
```

```
Explanation -----------------------------------------------------
1) First, we lock TESTING.Main for read.
2) Next, we do a BMSMS (bit map set manipulation) step that intersects the
following row id bit maps:
   1) The bit map built for TESTING.Main by way of index # 12
      "TESTING.Main.Kind = 'B'".
   2) The bit map build for TESTING.Main by way of index # 8
      "TESTING.Main.NumB = '02'".
   3) The bit map built for TESTING.Main by way of index # 16
      "TESTING.Main.Event ='001'".
   The resulting bit map is placed in Spool 3. The estimated time for this step
is 17.77 seconds.
3) We do a SUM step to aggregate from TESTING.Main by way of index # 4
"TESTING.Main.NumA = '101'" and the bit map in Spool 3 (Last Use) with a residual
condition of ("(TESTING.Main.NumB = '02') and  ((TESTING.Main.Kind = 'B') and
TESTING.Main.Event = '001'))"). Aggregate Intermediate Results are  computed
globally, then placed in Spool 2.
4) We do an all-AMPs RETRIEVE step from Spool 2 (Last Use) by way of an all-rows
scan into Spool 1, which is built locally on the AMPs. The size of Spool 1 is
estimated to be 20 rows. The estimated time for this step is 0.11 seconds.
5) Finally, we send out an END TRANSACTION step to all AMPs involved in processing
the request.
-> The contents of Spool 1 are sent back to the user as a result of statement 1.
```

Terminology

New terminology in this explanation is defined as follows.

Phrase	Definition
A BMSMS (bit map set manipulation step) that intersects the following Row-Id bit maps: **1** The bit map built for ... by way of index # n ... **2** The bit map built for ... by way of index #n ...	On each nonunique secondary index sub-table, each data row ID is assigned a number from 0-32767. This number is used as an index into a bit map in which the bit for each qualifying row is turned on. BMSMS indicates that the intersection of sets of qualifying rows is computed by applying the logical AND operation to the bitmap representation of the sets.
residual condition	Selected rows are further qualified by one or more conditional expressions.

EXPLAIN and Parallel Steps

The EXPLAIN modifier also reports whether or not statements will be processed in parallel.

The steps that can be executed in parallel are numbered, indented in the explanatory text, and preceded by the following message:

```
... we execute the following steps in parallel.
```

Parallel steps can be used to process a request submitted in a transaction (which can be a user-generated transaction, a multi-statement request, a macro, or a solitary statement that affects multiple rows).

Up to 20 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 10 channels can be used for parallel processing when a request is not constrained to a primary index value.

For example, a non-primary-constrained request that does not involve redistribution of rows to other AMPs, such as a SELECT or UPDATE, requires only two channels. A request that does involve row redistribution, such as a join or an INSERT … SELECT, requires four channels.

The following BTEQ request is structured as a single transaction, and thus generates parallel-step processing.

In Teradata mode, the transaction is structured as follows.

```
BEGIN TRANSACTION
   ;INSERT Department (100,'Administration','NYC',10011)
   ;INSERT Department (600,'Manufacturing','CHI',10007)
   ;INSERT Department (500,'Engineering', 'ATL',10012)
   ;INSERT Department (600, 'Exec Office','NYC', 10018)
   ; END TRANSACTION ;
```

In ANSI mode, the transaction is structured as follows.

```
   INSERT Department (100,'Administration','NYC',10011)
   ;INSERT Department (600,'Manufacturing','CHI',10007)
   ;INSERT Department (500,'Engineering', 'ATL',10012)
   ;INSERT Department (600, 'Exec Office','NYC', 10018)
   ;COMMIT ;
```

If you issue an EXPLAIN modifier against these transactions, the request returns the identical explanation in either mode, except that the last line is not returned for an ANSI mode transaction.

```
Explanation
-----------------------------------------------------------------------

1)  First, we execute the following steps in parallel.

    1)  We do an INSERT into PERSONNEL.Department
    2)  We do an INSERT into PERSONNEL.Department
    3)  We do an INSERT into PERSONNEL.Department
    4)  We do an INSERT into PERSONNEL.Department
2)  Finally, we send out an END TRANSACTION step to all AMPs involved in
    processing the request.
->  No rows are returned to the user as the result of statement 1.
    No rows are returned to the user as the result of statement 2.
    No rows are returned to the user as the result of statement 3.
    No rows are returned to the user as the result of statement 4.
    No rows are returned to the user as the result of statement 5.
    No rows are returned to the user as the result of statement 6.
```

EXPLAIN and Partitioned Primary Index Access

Introduction

EXPLAIN reports indicate partition accesses, deletions, joins, and eliminations performed during query optimization.

Example 1

The following example demonstrates an EXPLAIN report for accessing a subset of partitions for a SELECT statement. The relevant phrase is highlighted in boldface type.

```
CREATE TABLE t1
  (a INTEGER,
   b INTEGER)
  PRIMARY INDEX(a) PARTITION BY RANGE_N(
    b BETWEEN 1 AND 10 EACH 1);

EXPLAIN SELECT *
FROM t1
WHERE b > 2;
```

```
1) First, we lock a distinct mws."pseudo table" for read on a RowHash
     to prevent global deadlock for mws.t1.
  2) Next, we lock mws.t1 for read.
  3) We do an all-AMPs RETRIEVE step from 8 partitions of mws.t1
     and with a condition of ("mws.t1.b > 2") into Spool 1
     (group_amps), which is built locally on the AMPs.  The size of
     Spool 1 is estimated with no confidence to be 1 row.  The
     estimated time for this step is 0.14 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> The contents of Spool 1 are sent back to the user as the result of
     statement 1.  The total estimated time is 0.14 seconds.
```

Terminology

New terminology in this explanation is defined as follows:

Phrase	Definition
n partitions of	Only *n* of the partitions are accessed, where *n* > 1. In this case, n = 8.

Example 2

The following example demonstrates an EXPLAIN report for a SELECT with an equality constraint on the partitioning column. The relevant phrase is highlighted in boldface type. The report indicates that all rows in a single partition are scanned across all AMPs.

```
CREATE TABLE t1
  (a INTEGER,
   b INTEGER)
  PRIMARY INDEX(a) PARTITION BY RANGE_N(
    b BETWEEN 1 AND 10 EACH 1);

EXPLAIN SELECT *
FROM t1
WHERE t1.b = 1;
```

```
1) First, we lock a distinct mws."pseudo table" for read on a RowHash
     to prevent global deadlock for mws.t1.
2) Next, we lock mws.t1 for read.
3) We do an all-AMPs RETRIEVE step from a single partition of mws.t1
   with a condition of ("mws.t1.b = 1") into Spool 1
   (group_amps), which is built locally on the AMPs.  The size of
   Spool 1 is estimated with no confidence to be 2 rows.  The
   estimated time for this step is 0.15 seconds.
4) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1.  The total estimated time is 0.15 seconds.
```

Terminology

New terminology in this explanation is defined as follows:

Phrase	Definition
a single partition of	Only one partition is accessed in processing this query.

Example 3

The following example demonstrates an EXPLAIN report for partitioned primary index access without any constraints on the partitioning column. The relevant phrase is in boldface type. The report indicates that all partitions are accessed by way of the primary index on a single AMP.

```
CREATE TABLE t1
  (a INTEGER,
   b INTEGER)
  PRIMARY INDEX(a) PARTITION BY RANGE_N(
    b BETWEEN 1 AND 10 EACH 1);

EXPLAIN SELECT *
FROM t1
WHERE t1.a = 1;
```

```
1) First, we do a single-AMP RETRIEVE step from all partitions
     of mws2.t1 by way of the primary index "mws2.t1.a = 1" with no
     residual conditions into Spool 1 (one_amp), which is built
     locally on that AMP.  The size of Spool 1 is estimated with low
     confidence to be 2 rows.  The estimated time for this step is 0.15
     seconds.
  -> The contents of Spool 1 are sent back to the user as the result of
     statement 1.  The total estimated time is 0.15 seconds.
```

Terminology

New terminology in this explanation is defined as follows:

Phrase	Definition
all partitions of	All partitions are accessed for primary index access in processing this query.

Example 4

The following example demonstrats the processing of a SELECT statement without any partition elimination. The phrase "n partitions of" does not occur in the report.

```
CREATE TABLE t1
  (a INTEGER,
   b INTEGER)
  PRIMARY INDEX(a) PARTITION BY RANGE_N(
    b BETWEEN 1 AND 10 EACH 1);

EXPLAIN SELECT *
FROM t1
WHERE b > -1;
```

```
1) First, we lock a distinct mws."pseudo table" for read on a RowHash
     to prevent global deadlock for mws.t1.
  2) Next, we lock mws.t1 for read.
  3) We do an all-AMPs RETRIEVE step from mws.t1 by way of an all-rows
     scan with a condition of ("mws.t1.b > -1") into Spool 1
     (group_amps), which is built locally on the AMPs.  The size of
     Spool 1 is estimated with no confidence to be 1 row.  The
     estimated time for this step is 0.15 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> The contents of Spool 1 are sent back to the user as the result of
     statement 1.  The total estimated time is 0.15 seconds.
```

Example 5

Two steps are generated to perform the partial and full partition deletions, respectively, as demonstrated in the following EXPLAIN reports. The relevant phrases are highlighted in boldface type.

```
CREATE TABLE t2
  (a INTEGER,
   b INTEGER)
  PRIMARY INDEX(a) PARTITION BY RANGE_N(
    b BETWEEN 1 AND 10 EACH 2);

EXPLAIN DELETE FROM t2
WHERE b BETWEEN 4 AND 7;
```

```
1) First, we lock a distinct mws."pseudo table" for write on a
     RowHash to prevent global deadlock for mws.t2.
  2) Next, we lock mws.t2 for write.
  3) We do an all-AMPs DELETE from 2 partitions of mws.t2
     with a condition of ("(mws.t2.b <= 7) AND (mws.t2.b >= 4)").
  4) We do an all-AMPs DELETE of a single partition from mws.t2 with a
     condition of ("(mws.t2.b <= 7) AND (mws.t2.b >= 4)").
  5) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

```
EXPLAIN DELETE FROM t2
WHERE b BETWEEN 4 AND 8;
```

```
1) First, we lock a distinct mws."pseudo table" for write on a
     RowHash to prevent global deadlock for mws.t2.
  2) Next, we lock mws.t2 for write.
  3) We do an all-AMPs DELETE from a single partition of mws.t2
     with a condition of ("(mws.t2.b <= 8) AND (mws.t2.b >= 4)").
  4) We do an all-AMPs DELETE of 2 partitions from mws.t2 with a
     condition of ("(mws.t2.b <= 8) AND (mws.t2.b >= 4)").
  5) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

Terminology

New terminology in this explanation is defined as follows:

Phrase	Definition
of a single partition	The optimizer determined that all rows in a single partition can be deleted. In some cases, this allows for faster deletion of the entire partition.
of *n* partitions	The optimizer determined that all rows in each of *n* partitions can be deleted, where $n > 1$. In some cases, this allows for faster deletion of entire partitions.

Example 6

The following example demonstrates a spool with a partitioned primary index and a rowkey-based join. The relevant phrases are highlighted in boldface type.

```
CREATE TABLE t3
  (a INTEGER,
   b INTEGER)
PRIMARY INDEX(a);

CREATE TABLE t4
  (a INTEGER,
   b INTEGER)
PRIMARY INDEX(a)
 PARTITION BY b;

EXPLAIN SELECT *
FROM t3, t4
WHERE t3.a = t4.a
AND t3.b = t4.b;
```

```
1) First, we lock a distinct mws."pseudo table" for read on a RowHash
     to prevent global deadlock for mws.t3.
  2) Next, we lock a distinct mws."pseudo table" for read on a RowHash
     to prevent global deadlock for mws.t4.
  3) We lock mws.t3 for read, and we lock mws.t4 for read.
  4) We do an all-AMPs RETRIEVE step from mws.t3 by way of an all-rows
     scan with a condition of ("(NOT (mws.t3.b IS NULL )) AND (NOT
     (mws.t3.a IS NULL ))") into Spool 2 (all_amps), which is built
     locally on the AMPs.  Then we do an all-AMPs SORT to partition
     Spool 2 by rowkey.  The size of Spool 2 is estimated with no
     confidence to be 2 rows.  The estimated time for this step is
     0.03 seconds.
  5) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of a
     RowHash match scan, which is joined to mws.t4 with a condition of (
     "NOT (mws.t4.a IS NULL)").  Spool 2 and mws.t4 are joined using a
     rowkey-based merge join, with a join condition of ("(a = mws.t4.a)
     AND (b = mws.t4.b)").  The input table mws.t4 will not be cached in
     memory.  The result goes into Spool 1 (all_amps), which is built locally
     on the AMPs.  The size of Spool 1 is estimated with no confidence to
     be 1 row.  The estimated time for this step is 0.20 seconds.
  6) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
     statement 1.  The total estimated time is 0.22 seconds.
```

Terminology

New terminology in this explanation is defined as follows:

Phrase	Definition
SORT to partition Spool *n* by rowkey.	The optimizer determined that a spool file is to be partitioned based on the same partitioning expression as a table to which the spool file is be joined. That is, the spool is to be sorted by rowkey (partition and hash). Partitioning the spool file in this way enables a faster join with the partitioned table. *n* is the spool number.
a rowkey-based	The join is hash-based by partition (rowkey). In this case, there are equality constraints on both the partitioning and primary index columns. This enables a faster join since each non-eliminated partition needs to be joined with at most only one other partition. When this phrase is not reported,then the join is hash-based. That is, there are equality constraints on the primary index columns from which the hash is derived. For a partitioned table, there is additional overhead incurred by processing the table in hash order. Note that with either method, the join conditions must still be validated.

Example 7

The following example demonstrates one step for joining two tables having the same partitioning and primary keys. The relevant phrases are highlighted in boldface type.

```
CREATE TABLE Orders
     (o_orderkey INTEGER NOT NULL,
      o_custkey INTEGER,
      o_orderstatus CHAR(1) CASESPECIFIC,
      o_totalprice DECIMAL(13,2) NOT NULL,
      o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
      o_orderpriority CHAR(21),
      o_clerk CHAR(16),
      o_shippriority INTEGER,
      o_comment VARCHAR(79))
   PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(
  o_orderdate BETWEEN DATE '1992-01-01' AND DATE '1998-12-31'
              EACH INTERVAL '1' MONTH)
   UNIQUE INDEX (o_orderkey);
```

```
CREATE TABLE Lineitem
     (l_orderkey INTEGER NOT NULL,
      l_partkey INTEGER NOT NULL,
      l_suppkey INTEGER,
      l_linenumber INTEGER,
      l_quantity INTEGER NOT NULL,
      l_extendedprice DECIMAL(13,2) NOT NULL,
      l_discount DECIMAL(13,2),
      l_tax DECIMAL(13,2),
      l_returnflag CHAR(1),
      l_linestatus CHAR(1),
      l_shipdate DATE FORMAT 'yyyy-mm-dd',
      l_commitdate DATE FORMAT 'yyyy-mm-dd',
      l_receiptdate DATE FORMAT 'yyyy-mm-dd',
      l_shipinstruct VARCHAR(25),
      l_shipmode VARCHAR(10),
      l_comment VARCHAR(44))
   PRIMARY INDEX (l_orderkey)
PARTITION BY RANGE_N(
  l_shipdate BETWEEN DATE '1992-01-01' AND DATE '1998-12-31'
  EACH INTERVAL '1' MONTH);


EXPLAIN SELECT *
FROM lineitem, ordertbl
WHERE  l_orderkey = o_orderkey
AND l_shipdate = o_orderdate
AND (o_orderdate < DATE '1993-10-01')
AND (o_orderdate >= DATE '1993-07-01')
ORDER BY o_orderdate, l_orderkey;

…
…
3) We do an all-AMPs JOIN step from 3 partitions of TH.ORDERTBL
   with a condition of (
   "(TH.ORDERTBL.O_ORDERDATE < DATE '1993-10-01') AND
   (TH.ORDERTBL.O_ORDERDATE >= DATE '1993-07-01')"),
   which is joined to TH.LINEITEM with a condition of (
   "TH.LINEITEM.L_COMMITDATE < TH.LINEITEM.L_RECEIPTDATE").
   TH.ORDERTBL and TH.LINEITEM are joined using
   a rowkey-based inclusion merge join, with a join condition of (
   "TH.LINEITEM.L_ORDERKEY = TH.ORDERTBL.O_ORDERKEY").
   The input tables TH.ORDERTBL and TH.LINEITEM will
   not be cached in memory.  The result goes into Spool 3 (all_amps),
   which is built locally on the AMPs.  The size of Spool 3 is
   estimated with no confidence to be 7,739,047 rows.  The
   estimated time for this step is 1 hour and 34 minutes.
…
…
```

Example 8

The following example demonstrates partition elimination in an aggregation. The relevant phrase is highlighted in boldface type.

```
CREATE TABLE t1
  (a INTEGER,
   b INTEGER)
  PRIMARY INDEX(a) PARTITION BY RANGE_N(
    b BETWEEN 1 AND 10 EACH 1);

EXPLAIN SELECT MAX(a)
FROM t1
WHERE b > 3;
```

```
Explanation
-------------------------------------------------------------------------------
  1) First, we lock a distinct mws."pseudo table" for read on a RowHash
     to prevent global deadlock for mws.t1.
  2) Next, we lock mws.t1 for read.
  3) We do a SUM step to aggregate from 7 partitions of mws.t1
     with a condition of ("mws.t1.b > 3").  Aggregate
     Intermediate Results are computed globally, then placed in Spool 3.
     The input table will not be cached in memory, but it is eligible
     for synchronized scanning.  The size of Spool 3 is estimated with
     high confidence to be 1 row.  The estimated time for this step is
     2.35 seconds.
  4) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of
     an all-rows scan into Spool 1 (group_amps), which is built locally
     on the AMPs.  The size of Spool 1 is estimated with high
     confidence to be 1 row.  The estimated time for this step is 0.17
     seconds.
  5) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> The contents of Spool 1 are sent back to the user as the result of
     statement 1.
```

EXPLAIN and MERGE Conditional Steps

Introduction

The EXPLAIN modifier is useful in determining the conditional steps in MERGE processing.[1]

Terminology

New terminology in this set of EXPLAIN reports is defined as follows:

Phrase	Definition
if no update in <step number>	The match condition for the update phase was not met, so the conditional insert phase will be performed. <step number> indicates the MERGE update step.
if <step number> not executed,	Indicates the action taken if the first step in an UPDATE block in not performed. Reported only if <step number> is greater than 1.

Database Object DDL for Examples

The following DDL statements create the tables accessed by the EXPLAINed DML statement examples:

```
CREATE TABLE contact
     (
      contact_number INTEGER,
      contact_name   CHARACTER(30),
      area_code      SMALLINT NOT NULL,
      phone          INTEGER NOT NULL,
      extension      INTEGER )
      UNIQUE PRIMARY INDEX ( contact_number );
```

1 MERGE conditional steps are insert operations that are performed after an unconditional update operation does not meet its matching condition only when both WHEN MATCHED and WHEN NOT MATCHED clauses are specified.

```
CREATE TABLE contact_t
   (
    number    INTEGER,
    name      CHARACTER(30),
    area_code SMALLINT NOT NULL,
    phone     INTEGER NOT NULL,
    extension INTEGER)
    UNIQUE PRIMARY INDEX (number);
```

Example 1

The following example demonstrates simple conditional insert processing without trigger or join index steps. The relevant phrases in the EXPLAIN report are highlighted in boldface type:

```
EXPLAIN MERGE INTO contact_t AS t
USING (SELECT contact_number, contact_name, area_code, phone,
              extension
       FROM contact
       WHERE contact_number = 8005) s
ON (t.number =  8005)
WHEN MATCHED THEN
    UPDATE SET name ='Name beingUpdated' ,
               extension = s.extension
WHEN NOT MATCHED THEN
    INSERT (number, name, area_code, phone, extension)
    VALUES (s.contact_number, s.contact_name,
            s.area_code, s.phone, s.extension) ;

 *** Help information returned. 20 rows.
 *** Total elapsed time was 1 second.
```

```
Explanation
-----------------------------------------------------------------------------
  1) First, we do a single-AMP MERGE DELETE to TEST.contact_t from
     TEST.contact by way of a RowHash match scan.  New updated rows are
     built and the result goes into Spool 1 (one-amp), which is built
     locally on the AMPs.  Then we do a SORT to order Spool 1 by row
     hash.
  2) Next, we execute the following steps in parallel.
       1) We do a single-AMP MERGE into TEST.contact_t from Spool 1
          (Last Use).
       2) If no update in 2.1, we do a single-AMP RETRIEVE step from
          TEST.contact by way of the unique primary index
          "TEST.contact.contact_number = 8005" with no residual
          conditions into Spool 2 (one-amp), which is built locally on
          that AMP.  Then we do a SORT to order Spool 2 by row hash.
          The size of Spool 2 is estimated with high confidence to be 1
          row.  The estimated time for this step is 0.02 seconds.
  3) If no update in 2.1, we do a single-AMP MERGE into TEST.contact_t
     from Spool 2 (Last Use).
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

Only if no update in <step number> conditional steps are reported by this EXPLAIN, indicating the steps to be performed only if the update to contact_t fails. The report indicates that the MERGE first attempts to perform an update operation (step 2.1). If no row is found to update, then the statement inserts a new row (step 3).

Example 2

The following example demonstrates slightly more complicated conditional insert processing when a join index is defined on tables t1 and t2.

The DDL for the join index is as follows:

```
CREATE JOIN INDEX j AS
SELECT * FROM t1 LEFT OUTER JOIN t2
ON (t1.y1 = t2.y2);
```

The relevant phrases in the EXPLAIN report are highlighted in boldface type:

```
EXPLAIN MERGE INTO t1
USING VALUES(1,2) AS s(x1, y1)
ON t1.x1 = 4
WHEN MATCHED THEN
UPDATE SET y1 = 5
WHEN NOT MATCHED THEN
INSERT(4,5);

 *** Help information returned. 44 rows.
 *** Total elapsed time was 1 second.
```

```
Explanation
-----------------------------------------------------------------------------
  1) First, we lock a distinct TEST."pseudo table" for read on a
     RowHash to prevent global deadlock for TEST.t2.
  2) Next, we lock TEST.t2 for read.
  3) We execute the following steps in parallel.
       1) We do a single-AMP DELETE from TEST.j by way of the primary
          index "TEST.j.x1 = 4" with no residual conditions.
       2) We do an all-AMPs RETRIEVE step from TEST.t2 by way of an
          all-rows scan with a condition of ("TEST.t2.y2 = 5") into
          Spool 2 (one-amp), which is redistributed by hash code to all
          AMPs.  The size of Spool 2 is estimated with no confidence to
          be 1 row.  The estimated time for this step is 0.02 seconds.
  4) We do a single-AMP JOIN step from TEST.t1 by way of the primary
     index "TEST.t1.x1 = 4" with no residual conditions, which is
     joined to Spool 2 (Last Use).  TEST.t1 and Spool 2 are left outer
     joined using a product join, with a join condition of ("(1=1)").
     The result goes into Spool 1 (one-amp), which is redistributed by
     hash code to all AMPs.  Then we do a SORT to order Spool 1 by row
     hash.  The size of Spool 1 is estimated with no confidence to be 3
     rows.  The estimated time for this step is 0.03 seconds.
  5) We execute the following steps in parallel.
       1) We do a single-AMP MERGE into TEST.j from Spool 1 (Last Use).
       2) We do a single-AMP UPDATE from TEST.t1 by way of the primary
          index "TEST.t1.x1 = 4" with no residual conditions.
       3) If no update in 5.2, we do an INSERT into TEST.t1.
       4) If no update in 5.2, we do an INSERT into Spool 3.
       5) If no update in 5.2, we do an all-AMPs RETRIEVE step from
          TEST.t2 by way of an all-rows scan with no residual
          conditions into Spool 5 (all_amps), which is duplicated on
          all AMPs.  The size of Spool 5 is estimated with low
          confidence to be 4 rows.  The estimated time for this step is
          0.02 seconds.
  6) If no update in 5.2, we do an all-AMPs JOIN step from Spool 3
     (Last Use) by way of an all-rows scan, which is joined to Spool 5
     (Last Use).  Spool 3 and Spool 5 are left outer joined using a
     product join, with a join condition of ("y1 = y2").  The result
     goes into Spool 4 (one-amp), which is redistributed by hash code
     to all AMPs.  Then we do a SORT to order Spool 4 by row hash.  The
     size of Spool 4 is estimated with no confidence to be 2 rows.  The
     estimated time for this step is 0.03 seconds.
  7) If no update in 5.2, we do a single-AMP MERGE into TEST.j from
     Spool 4 (Last Use).
  8) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

Again, only if no update in <step number> conditional steps are reported by this EXPLAIN. These steps are performed only if the initial unconditional update attempt to table t1 is unsuccessful. The report indicates that the MERGE first attempts to perform an update operation (step 5.2). If no row is found to update, then the statement inserts a new row (steps 5.3 and 5.4). The join index j is updated by steps 3.1, 5.1, and 7.

Example 3

The following example demonstrates conditional insert processing when an upsert trigger is defined on table t1.

The DDL for the trigger is as follows:

```
CREATE TRIGGER r1 AFTER INSERT ON t1
(UPDATE t2 SET y2 = 9 WHERE x2 = 8
 ELSE INSERT t2(8,9);
);
```

The relevant phrases in the EXPLAIN report are highlighted in boldface type:

```
EXPLAIN MERGE INTO t1
USING VALUES(1,2) AS s(x1, y1)
ON t1.x1 = 4
WHEN MATCHED THEN
UPDATE SET y1 = 5
WHEN NOT MATCHED THEN
INSERT(4,5);

 *** Help information returned. 11 rows.
 *** Total elapsed time was 1 second.

Explanation
-----------------------------------------------------------------------------
  1) First, we execute the following steps in parallel.
       1) We do a single-AMP UPDATE from TEST.t1 by way of the primary
          index "TEST.t1.x1 = 4" with no residual conditions.
       2) If no update in 1.1, we do an INSERT into TEST.t1.
       3) If no update in 1.1, we do a single-AMP UPDATE from TEST.t2
          by way of the primary index "TEST.t2.x2 = 8" with no residual
          conditions.  If the row cannot be found, then we do an INSERT
          into TEST.t2.
  2) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

The unconditional update to table t1 is attempted in step 1.1. If the update is unsuccessful, then the MERGE inserts a new row into t1 in step 1.2 and fires trigger r1, which attempts to update table t2 in step 1.3. If an update cannot be performed, then the trigger inserts a new row into table t2.

Example 4

The following example demonstrates conditional insert processing when an abort trigger is defined to fire after an update on table t4.

This example uses the conditional abort trigger defined by this DDL:

```
CREATE TRIGGER aborttrig AFTER UPDATE ON t4
(UPDATE t5 SET y5 =5 WHERE x5 = 3
ELSE INSERT t5(3,5);
ABORT FROM t5 WHERE x5 =1;
DELETE t3 WHERE x3 = 10;
ABORT 'unconditional abort';
);
```

The relevant phrases in the EXPLAIN report are highlighted in boldface type:

```
EXPLAIN MERGE INTO t4
USING VALUES(1,2) AS s(x1, y1)
ON t4.x4 = 4
WHEN MATCHED THEN
UPDATE SET y4 = 5
WHEN NOT MATCHED THEN
INSERT(4,5);

 *** Help information returned. 18 rows.
 *** Total elapsed time was 1 second.

Explanation
--------------------------------------------------------------------------------
  1) First, we execute the following steps in parallel.
       1) We do a single-AMP UPDATE from TEST.t4 by way of the primary
          index "TEST.t4.x4 = 4" with no residual conditions.
       2) We do a single-AMP UPDATE from TEST.t5 by way of the primary
          index "TEST.t5.x5 = 3" with no residual conditions.  If the
          row cannot be found, then we do an INSERT into TEST.t5.
  2) Next, we execute the following steps in parallel.
       1) If no update in 1.1, we do a single-AMP ABORT test from
          TEST.t5 by way of the primary index "TEST.t5.x5 = 1" with no
          residual conditions.
       2) If no update in 1.1, we do a single-AMP DELETE from TEST.t3
          by way of the primary index "TEST.t3.x3 = 10" with no
          residual conditions.
  3) If no update in 1.1, we unconditionally ABORT the transaction.
  4) If no update in 1.1, we do an INSERT into TEST.t4.
  5) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

The unconditional update to table t4 is attempted in step 1.1. If the update is successful, then trigger aborttrig is fired, which attempts to perform an atomic upsert on table t5 in step 1.2.

If no update is made to table t4, then the MERGE inserts a new row into it in step 1.2 and fires trigger aborttrig, which attempts to perform an atomic upsert operation update on table t2 in step 1.3. If an update cannot be performed, then the trigger inserts a new row into table t2.

EXPLAIN and UPDATE (Upsert Form) Conditional Steps

Introduction

The EXPLAIN modifier is useful in determining the conditional steps in UPDATE (Upsert Form) processing.[2]

Terminology

New terminology in this set of EXPLAIN reports is defined as follows:

Phrase	Definition
if no update in <step number>	The match condition for the update phase was not met, so the conditional insert phase will be performed. <step number> indicates the MERGE update step.
if <step number> not executed,	Indicates the action taken if the first step in an UPDATE block in not performed. Reported only if <step number> is greater than 1.

Database Object DDL for Examples

The following DDL statements create the tables accessed by the EXPLAINed DML statement examples.

```
CREATE TABLE t1
     (
      x1 INTEGER,
      y1 INTEGER );

CREATE TABLE t2
   (
    x2    INTEGER NOT NULL,
    y2    INTEGER NOT NULL );
```

2 Atomic upsert conditional steps are insert operations that are performed after an unconditional update operation does not meet its matching condition.

Example 1

The following example demonstrates simple conditional insert processing into table t1 without trigger or join index steps:

```
EXPLAIN UPDATE t1 SET y1 = 3 WHERE x1 = 2
ELSE INSERT t1(2, 3);

 *** Help information returned. 4 rows.
 *** Total elapsed time was 1 second.
```

```
Explanation
-----------------------------------------------------------------------------
  1) First, we do a single-AMP UPDATE from TEST.t1 by way of the
     primary index "TEST.t1.x1 = 2" with no residual conditions.  If
     the row cannot be found, then we do an INSERT into TEST.t1.
  -> No rows are returned to the user as the result of statement 1.
```

Both the unconditional update attempt and the conditional insert are combined in a single step.

Example 2

The following example demonstrates slightly more complicated upsert processing when a join index is defined on tables t1 and t2.

The DDL for the join index is as follows:

```
CREATE JOIN INDEX j AS
SELECT * FROM t1 LEFT OUTER JOIN t2
ON (t1.y1 = t2.y2);

EXPLAIN UPDATE t1 SET y1 = 3 WHERE x1 = 2
ELSE INSERT t1(2, 3);

 *** Help information returned. 44 rows.
 *** Total elapsed time was 1 second.
```

```
Explanation
------------------------------------------------------------------------
  1) First, we lock a distinct TEST."pseudo table" for read on a
     RowHash to prevent global deadlock for TEST.t2.
  2) Next, we lock TEST.t2 for read.
  3) We execute the following steps in parallel.
       1) We do a single-AMP DELETE from TEST.j by way of the primary
          index "TEST.j.x1 = 2" with no residual conditions.
       2) We do an all-AMPs RETRIEVE step from TEST.t2 by way of an
          all-rows scan with a condition of ("TEST.t2.y2 = 3") into
          Spool 2 (one-amp), which is redistributed by hash code to all
          AMPs.  The size of Spool 2 is estimated with no confidence to
          be 1 row.  The estimated time for this step is 0.02 seconds.
  4) We do a single-AMP JOIN step from TEST.t1 by way of the primary
     index "TEST.t1.x1 = 2" with no residual conditions, which is
     joined to Spool 2 (Last Use).  TEST.t1 and Spool 2 are left outer
     joined using a product join, with a join condition of ("(1=1)").
     The result goes into Spool 1 (one-amp), which is redistributed by
     hash code to all AMPs.  Then we do a SORT to order Spool 1 by row
     hash.  The size of Spool 1 is estimated with no confidence to be 3
     rows.  The estimated time for this step is 0.03 seconds.
  5) We execute the following steps in parallel.
       1) We do a single-AMP MERGE into TEST.j from Spool 1 (Last Use).
       2) We do a single-AMP UPDATE from TEST.t1 by way of the primary
          index "TEST.t1.x1 = 2" with no residual conditions.  If the
          row cannot be found, then we do an INSERT into TEST.t1.
       3) If no update in 5.2, we do an INSERT into Spool 3.
       4) If no update in 5.2, we do an all-AMPs RETRIEVE step from
          TEST.t2 by way of an all-rows scan with no residual
          conditions into Spool 5 (all_amps), which is duplicated on
          all AMPs.  The size of Spool 5 is estimated with low
          confidence to be 4 rows.  The estimated time for this step is
          0.02 seconds.
  6) If no update in 5.2, we do an all-AMPs JOIN step from Spool 3
     (Last Use) by way of an all-rows scan, which is joined to Spool 5
     (Last Use).  Spool 3 and Spool 5 are left outer joined using a
     product join, with a join condition of ("y1 = y2").  The result
     goes into Spool 4 (one-amp), which is redistributed by hash code
     to all AMPs.  Then we do a SORT to order Spool 4 by row hash.  The
     size of Spool 4 is estimated with no confidence to be 2 rows.  The
     estimated time for this step is 0.03 seconds.
  7) If no update in 5.2, we do a single-AMP MERGE into TEST.j from
     Spool 4 (Last Use).
  8) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

The EXPLAIN report is more complicated because of the join index j. Notice the instances of the phrase "If no update in <step number>" in steps 5.3, 5.4, 6, and 7, indicating the operations undertaken if the match condition for update was not met. The report indicates that the MERGE first attempts to perform an update operation (step 5.2). If no row is found to update, then the statement inserts a new row (step 5.3). The join index j is updated by steps 1 through 4, 6, and 7.

Example 3

The following upsert statement involves a simple trigger. The relevant phrases in the EXPLAIN report are highlighted in boldface type.

The DDL for the trigger is as follows:

```
CREATE TRIGGER r1 AFTER INSERT ON t1
(UPDATE t2 SET y2 = 9 WHERE x2 = 8
 ELSE INSERT t2(8,9);
);

EXPLAIN UPDATE t1 SET y1 = 3 WHERE x1 = 2
ELSE INSERT t1(2, 3);

 *** Help information returned. 11 rows.
 *** Total elapsed time was 1 second.
Explanation

--------------------------------------------------------------------------
  1) First, we execute the following steps in parallel.
       1) We do a single-AMP UPDATE from TEST.t1 by way of the primary
          index "TEST.t1.x1 = 2" with no residual conditions.  If the
          row cannot be found, then we do an INSERT into TEST.t1.
       2) If no update in 1.1, we do a single-AMP UPDATE from TEST.t2
          by way of the primary index "TEST.t2.x2 = 8" with no residual
          conditions.  If the row cannot be found, then we do an INSERT
          into TEST.t2.
  2) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

The EXPLAIN report is moderately complex because of the trigger. Step 1.1 handles the unconditional update attempt and the conditional insert, while step 1.2 handles the triggered update to table t2.

Notice the the phrase "If no update in <step number>" in step 1.2, indicating that the step performs only if the match condition for update was not met.

Example 4

This example uses the conditional abort trigger defined by this DDL:

```
CREATE TRIGGER aborttrig AFTER UPDATE ON t4
(UPDATE t5 SET y5 =5 WHERE x5 = 3
ELSE INSERT t5(3,5);
ABORT FROM t5 WHERE x5 =1;
DELETE t3 WHERE x3 = 10;
ABORT 'unconditional abort';
);
```

The relevant phrases in the EXPLAIN report are highlighted in boldface type.

```
EXPLAIN UPDATE t4 SET y4 = 3 WHERE x4 = 2
ELSE INSERT t4(2, 3);

 *** Help information returned. 18 rows.
 *** Total elapsed time was 1 second.

Explanation
-----------------------------------------------------------------------------
  1) First, we execute the following steps in parallel.
       1) We do a single-AMP UPDATE from TEST.t4 by way of the primary
          index "TEST.t4.x4 = 2" with no residual conditions.
       2) We do a single-AMP UPDATE from TEST.t5 by way of the primary
          index "TEST.t5.x5 = 3" with no residual conditions.  If the
          row cannot be found, then we do an INSERT into TEST.t5.
  2) Next, we execute the following steps in parallel.
       1) If no update in 1.1, we do a single-AMP ABORT test from
          TEST.t5 by way of the primary index "TEST.t5.x5 = 1" with no
          residual conditions.
       2) If no update in 1.1, we do a single-AMP DELETE from TEST.t3
          by way of the primary index "TEST.t3.x3 = 10" with no
          residual conditions.
  3) If no update in 1.1, we unconditionally ABORT the transaction.
  4) If no update in 1.1, we do an INSERT into TEST.t4.
  5) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

Notice the instances of the phrase "If no update in <step number>" in steps 2.1, 2.2, 3, and 4, indicating that the step performs only if an update was not successful.

Step 1.1 handles the unconditional update attempt to t4, while step 1.2 performs the update processing defined by the trigger. Steps 2 and 3 continue to perform trigger-related operations, while step 4 performs the upsert-specified insert operation if the update to t4 fails.

Example 5

This example uses the 5 tables defined by the following DDL statements:

```
CREATE TABLE t7 (x7 INTEGER, y7 INTEGER);

CREATE TABLE t8 (x8 INTEGER, y8 INTEGER);

CREATE TABLE t9 (x9 INTEGER, y9 INTEGER);

CREATE TABLE t10 (x10 INTEGER, y10 INTEGER);

CREATE TABLE t11 (x11 INTEGER, y11 INTEGER);
```

The example also uses the following definitions for triggers r6 through r10:

```
CREATE TRIGGER r6 ENABLED AFTER UPDATE ON t1
(UPDATE t7 SET y7 = 7 WHERE x7 = 6
 ELSE INSERT t7(6, 7); );

CREATE TRIGGER r7 ENABLED BEFORE UPDATE ON t7
(UPDATE t8 SET y8 = 8 WHERE x8 = 7
 ELSE INSERT t8(7, 8); );

CREATE TRIGGER r8 ENABLED AFTER UPDATE ON t7
(UPDATE t9 SET y9 = 8 WHERE x9 = 7
 ELSE INSERT t9(7, 8); );

CREATE TRIGGER r9 ENABLED BEFORE INSERT ON t7
(UPDATE t10 SET y10 = 9 WHERE x10 = 8
 ELSE INSERT t10(8, 9); );

CREATE TRIGGER r10 ENABLED AFTER INSERT ON t7
(UPDATE t11 SET y11 = 10 WHERE x11 = 9
 ELSE INSERT t11(9, 10); );
EXPLAIN UPDATE t1 SET y1 = 20 WHERE x1 = 30
ELSE INSERT t1(30, 20);

 *** Help information returned. 41 rows.
 *** Total elapsed time was 1 second.
```

The relevant phrases in the EXPLAIN report are highlighted in boldface type.

```
Explanation
------------------------------------------------------------------------------
  1) First, we lock a distinct UTEST3AB."pseudo table" for read on a
     RowHash to prevent global deadlock for UTEST3AB.t7.
  2) Next, we lock UTEST3AB.t7 for read.
  3) We do a single-AMP UPDATE from UTEST3AB.t1 by way of the primary
     index "UTEST3AB.t1.x1 = 30" with no residual conditions.
  4) We do a single-AMP UPDATE from UTEST3AB.t8 by way of the primary
     index "UTEST3AB.t8.x8 = 7" with no residual conditions.  If the
     row cannot be found, then we do an INSERT into UTEST3AB.t8.
  5) We do a single-AMP RETRIEVE step from UTEST3AB.t7 by way of the
     primary index "UTEST3AB.t7.x7 = 6" with no residual conditions
     into Spool 4 (all_amps), which is duplicated on all AMPs.  The
     size of Spool 4 is estimated with low confidence to be 4 rows.
     The estimated time for this step is 0.02 seconds.
  6) We do an all-AMPs JOIN step from UTEST3AB.t7 by way of an all-rows
     scan with no residual conditions, which is joined to Spool 4 (Last
     Use).  UTEST3AB.t7 and Spool 4 are joined using a product join,
     with a join condition of ("(1=1)").  The result goes into Spool 3
     (all_amps), which is redistributed by hash code to all AMPs.  Then
     we do a SORT to order Spool 3 by the sort key in spool field1.
     The size of Spool 3 is estimated with low confidence to be 4 rows.
     The estimated time for this step is 0.03 seconds.
```

```
 7) We execute the following steps in parallel.
      1) We do a Single AMP MERGE Update to UTEST3AB.t7 from Spool 3
         (Last Use) via ROWID.
      2) If 7.1 not executed, we do a single-AMP UPDATE from
         UTEST3AB.t9 by way of the primary index "UTEST3AB.t9.x9 = 7"
         with no residual conditions.  If the row cannot be found,
         then we do an INSERT into UTEST3AB.t9.
 8) If no update in 7.1, we do a single-AMP UPDATE from UTEST3AB.t10
    by way of the primary index "UTEST3AB.t10.x10 = 8" with no
    residual conditions.  If the row cannot be found, then we do an
    INSERT into UTEST3AB.t10.
 9) If no update in 7.1, we do an INSERT into UTEST3AB.t7.
10) If no update in 7.1, we do a single-AMP UPDATE from UTEST3AB.t11
    by way of the primary index "UTEST3AB.t11.x11 = 9" with no
    residual conditions.  If the row cannot be found, then we do an
    INSERT into UTEST3AB.t11.
11) If no update in 3, we do an INSERT into UTEST3AB.t1.
12) Finally, we send out an END TRANSACTION step to all AMPs involved
    in processing the request.
 -> No rows are returned to the user as the result of statement 1.
```

Notice the instances of the phrase "If no update in <step number>" in steps 8, 9, 10, and 11, indicating operations that are performed only if an update does not succeed.

Only steps 3 and 11 relate directly to the atomic upsert statement. The other steps all perform triggered actions specified by triggers r6 through r10.

Step 7.2 introduces the phrase "If <step number> not executed," indicating that the action taken if the first step in an THEN-ELSE block in not performed. Notice that the parallel step number, 7.1, is greater than 1. This phrase is reported only when <step number> is greater than 1.

GROUP BY Clause

Groups result rows of a SELECT query by the values in one or more columns.

See "GROUP BY Clause" on page 1-57 for a detailed description of GROUP BY.

HAVING Clause

Specifies a conditional expression that must be satisfied by the rows in a SELECT query to be included in the resulting data. The condition can define one or more aggregates (for example, MAX, MIN, AVG, SUM, COUNT), and can be applied to the rows of:

- A single group defined in the select expression-list, which has only aggregate results
- One or more groups defined in a GROUP BY clause

See "HAVING Clause" on page 1-61 for a detailed description of the HAVING clause.

INSERT

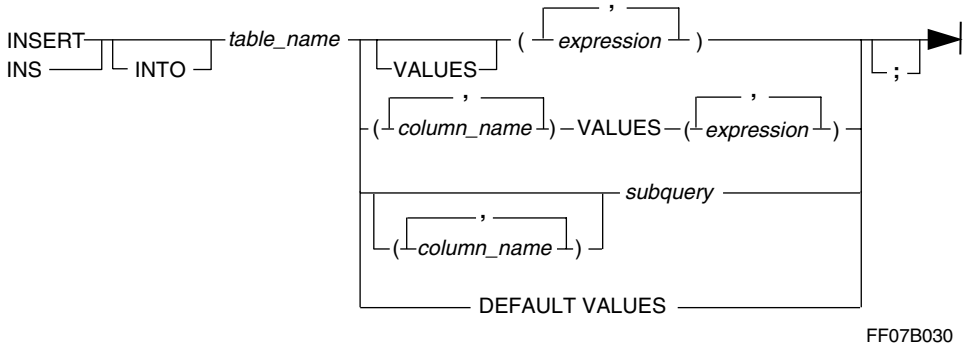
Purpose

Adds new rows to a named table by directly specifying the row data to be inserted (valued form) or by retrieving the new row data from another table (selected form).

Invocation

Executable.

Syntax



FF07B030

where:

Syntax Element …	Specifies …
INTO	a keyword that is required for ANSI compliance. If you do not specify INTO, then the statement is non-ANSI standard.
table_name	the name of the table directly or by means of a view.
VALUES	a keyword required for ANSI compliance. The colon is optional with host variables used in this clause.

WHEN *column_name* is …	THEN the VALUES keyword is …
specified	required.
not specified	optional.

Syntax Element . . .	Specifies . . .
column_name	the name of a column for which the insert operation supplies a new row value. *column_name* can be specified in any order. If a *column_name* is omitted, then any default value defined in the CREATE TABLE or CREATE VIEW statement is used.
expression	a constant or constant expression to be inserted into the new row for the columns specified in *column_name*. A constant expression is an expression containing only constants (for example, 3+5, or 19000/12). The system values CURRENT_DATE, DATE, CURRENT_TIME, TIME, and USER can be included in a constant expression. You can insert a NULL by specifying the reserved word NULL as an *expression* value. Values obtained from imported data, with a USING modifier, or as macro parameters, are accepted as constants. When no column name list is specified, then values are inserted in column definition order, from left to right. Use commas to indicate skipped columns (see "Example 5" on page 3-116). When a column name list and an expr list are used, values are assigned by matching *column_name* position with *expression* position. The two lists must have the same number of items. When an element of the expression list is omitted, it is treated as a NULL. This is not valid in ANSI SQL.
subquery	that the row or rows to be inserted consist of column values accessed by a query specification. If a column name list is not specified, then the SELECT statement must query the same number of columns as there are in the table that is receiving new rows. NULLs can be included in the select expression list for columns that are not to be assigned values. If the INSERT operation includes a column name list, values are assigned by matching *column_name* position with position of items in the select *expression* list.
DEFAULT VALUES	that a row consisting of default values is to be added to *table_name*. If any of the columns do not have a DEFAULT phrase defined for them, an error is returned and the insertion fails.

INSERT is sometimes described as having two forms: Valued and Selected (also known as INSERT … SELECT).

The syntax presented here describes both forms of the statement.

ANSI Compliance

INSERT is ANSI SQL-99-compliant with extensions.

The INS abbreviation is a Teradata extension to the ANSI SQL-99 standard.

Authorization

The following privilege rules apply to the INSERT statement:

- You must have the INSERT privilege on the referenced table.
- To insert rows into a table through a view, you must have the INSERT privilege on the view. Also, the immediate owner of the view (that is, the database in which the view resides) must have the INSERT privilege on the underlying object (view or base table).
- To insert rows into a table using a query specification, you must have the SELECT privilege for the referenced tables or views.
- When you perform an INSERT with a WHERE clause predicate that requires READ access to an object, you must have the SELECT privilege on the accessed object.

DEFAULT VALUES Option

When a table has all its columns defined as in the following table, an INSERT defined with the DEFAULT VALUES keywords adds a row consisting of defined default values (where indicated) and nulls for other columns.

Nullable?	Default defined?
Yes	No
Yes	Yes
No	Yes

This INSERT occurs when the row satisfies the conditions imposed by the table definition. Otherwise an error is returned.

When any column in the table is defined NOT NULL and has no defined DEFAULT, then an error is returned.

When a row is otherwise valid but duplicates an existing row and the table has been defined not to accept duplicate rows, then an error is returned.

Inserting Into NOT NULL Columns

The following rules apply to an INSERT operation that does not assign values for every column (field) in a new row:

- If the column is not declared as NOT NULL and no default value is declared, nulls are inserted.
- If the column is declared as NOT NULL and no default value is declared, an error message is returned.

Inserting Into Global Temporary Tables

Inserting into a global temporary table creates an instance of that table in the current session.

Inserting Into PPI Tables

The following rules apply to inserting rows into tables with a partitioned primary index.

- The outcome of expression evaluation errors of PPI tables, such as divide-by-zero errors, depends on the session mode.

In this session mode ...	Expression evaluation errors roll back this work unit ...
ANSI	request.
Teradata	transaction.

- If you attempt to insert a row into a PPI table for which the partitioning expression produces a partition number that does not result in a non-null value between 1 and 65 535 (after casting to INTEGER if it does not have an INTEGER data type), an error is returned.

Valid INSERT Operations

An INSERT operation does not return an error message if many of the following are true.

- The operation attempts to insert a character string that is longer or shorter than that declared for the column. The string is silently adjusted and inserted. However, in ANSI mode, if the string is too long a report is generated.
- The operation uses a query and no rows are returned.

Non-Valid INSERT Operations

An INSERT operation causes an error message to be returned if any of the following are true.

- The operation attempts to assign a value that will result in a violation of a unique index specification.
- The operation attempts to insert a row with no value or default for a column that is defined as NOT NULL.
- The operation attempts to assign a non-null value that violates a CHECK constraint declared for a column.
- The operation attempts to assign a value that is of a different numeric type than that declared for the column and the assigned value cannot be converted correctly.
- The operation attempts to assign a character value that is not in the repertoire of the destination character data type.
- The operation attempts to insert a character string having trailing pad characters into a VARCHAR field, and that operation causes the row to become identical to another row (except for the number of trailing pad characters).
- In ANSI mode, inserting character data, if in order to comply with maximum length of the target column, non-pad characters are truncated from the source data.

  In Teradata mode, the above insertion is permitted (characters are truncated silently). This could result in improper strings for the Kanji1 character data type.

Teradata ignores trailing pad characters in character strings when comparing values for field or row duplication.

Duplicate Rows and INSERT

When an insert operation would create a duplicate row, the outcome of the operation depends on how the table is defined.

IF the table is defined as . . .	THEN the duplicate row is . . .
MULTISET with no UNIQUE constraints	inserted.
• SET • MULTISET with UNIQUE constraints	not inserted. An error is returned to the requestor.

Duplicate Rows and INSERT … SELECT

If an INSERT using a SELECT subquery will create duplicate rows, then the result depends on the table definition.

Duplicate rows are permitted in a table defined as MULTISET if no UNIQUE constraints or UNIQUE indexes are defined.

Duplicate rows are *never* permitted in a table defined as SET.

FOR this table type . . .	Duplicate rows are . . .
MULTISET with no unique constraints	permitted. Inserted duplicate rows are stored in the table.
MULTISET with unique constraints[a]	not permitted. An error message is returned to the requestor.
SET	not permitted. **In this mode ...** / **Teradata ...** ANSI: rejects the entire transaction and returns a duplicate row error message to the requestor. Teradata: • Rejects the duplicate rows in the transaction. • Inserts the non-duplicate rows into the table. • Does not return an error message to the requestor.

a. All of the following are unique constraints in Teradata:
- Unique primary index (UPI)
- Unique secondary index (USI)
- Primary key (PK)
- UNIQUE column constraint
- GENERATED ALWAYS identity column

Identity Columns and INSERT

You can only perform singleton inserts into a table with an identity column. Bulk insert operations on tables having identity columns are not permitted for the INSERT statement. See "Example 6: INSERT and GENERATED ALWAYS Identity Columns" on page 3-117 and "Example 7: INSERT and GENERATED BY DEFAULT Identity Columns" on page 3-117.

Identity Columns and INSERT … SELECT

INSERT … SELECT operations into a target table with an identity column behave the same as singleton inserts. See "Example 8: Identity Columns and INSERT … SELECT" on page 3-118.

The INSERT Process

An INSERT process performs the following functions:

Stage	Process
1	Sets a write lock on the row, or on the table in the case of an INSERT with a SELECT subquery.
2	Handles the entire INSERT operation as a transaction in which every value is inserted successfully or no values are inserted. This is to prevent a partial insert from occurring.

The INSERT operation takes more processing time on a table defined with FALLBACK or a secondary index, because the FALLBACK copy of the table or the secondary index subtable also must be changed.

Inserting Rows Using Views

Use caution when granting the privilege to insert data through a view because data in fields not visible to the user might be inserted when a row is inserted through a view.

The following rules apply to inserting rows through a view:

- Both you and the immediate owner of the view must have the appropriate privileges.
- The view must reference columns in only one table or view.
- None of the columns in the view can be derived by using an expression to change values in the underlying table.
- Each column in the view must correspond one to one with a column in the underlying table or view.
- The view must include any column in the underlying table or view that is declared as NOT NULL.
- No two view columns can reference the same column in the underlying table.
- If the statement used to define a view contains a WHERE clause, and WITH CHECK OPTION, all values inserted through that view must satisfy constraints specified in the WHERE clause.

  If a view includes a WHERE clause but does not include the WITH CHECK OPTION, then data can be inserted that is not visible through that view.

Subqueries In INSERT Statements

An INSERT operation that uses a subquery, referred to as an INSERT … SELECT, differs from a simple INSERT in that many rows can be inserted in a single operation, and the row values can come from more than one table.

The query specification must always include a FROM *table_name* clause.

Also, a column name list must be defined in an INSERT statement that includes a subquery when the following conditions exist:

- The number of columns listed in the query specification differs from the number of columns in the table receiving new rows.
- The order of columns listed in the query specification differs from the order the columns were defined in the table receiving new rows.

Data Takes the Attributes of the New Table

If the column attributes defined for a new table differ from those of the columns whose data is being inserted using INSERT, then the data takes on the attributes of the new table.

The source column names can differ from the destination column names in the new table, but the data is inserted correctly as long as the SELECT statement lists the source column names in the same order as the corresponding destination columns in the CREATE TABLE statement. This is true even if the new table has a column that is to contain data derived (either arithmetically or by aggregate operation) from the column data in the existing table.

Fast Path INSERT … SELECT Statements

Multistatement INSERT … SELECTs are optimized so that each statement except for the last in the series returns a ‘zero row inserted’ message as a response. The retrieved rows for each SELECT are sent to an empty target table. The last INSERT … SELECT statement returns the total number of rows inserted for the entire request and sorts and merges the spool table into the target table.

Columns defined with the COMPRESS option cannot participate in fast path optimizations, so if you perform an INSERT … SELECT operation on compressed columns, fast path optimization is not specified by the Optimizer when it creates an access plan.

INSERT … SELECT Performance and Target Table Identity Column Primary Indexes

For those cases where INSERT … SELECT is optimized so that a direct merge from source to target table is possible, like when source and target tables have the identical structure, there can be as much as a threefold degradation of performance caused by the need for an extra step to spool and redistribute rows when the target table has an identity column primary index.

Rules for Fast Path INSERT … SELECT Statements

There are several restrictions that must be observed in order for the high performance fast path optimization to succeed in a multistatement request.

These rules are as follows:

- The target table must be empty.
- All INSERT statements in the multistatement request must have the same target table.
- Only INSERT statements can be included in the request.

If you insert other statement types into the multistatement request, the fast path optimization does not occur (only the first INSERT … SELECT in the series is optimized) and performance degrades accordingly.

General Rules for INSERT in Embedded SQL and Stored Procedures

The following rules apply to both forms of INSERT in embedded SQL and stored procedures:

- The row to be inserted is constructed by building a candidate row as follows:

Step	Action
1	Define each column value of the row to be NULL.
2	Assign each inserted value to the corresponding column.

IF the result row has this value for any NOT NULL column …	THEN the insertion …
any non-null value	succeeds.
a null	fails and SQLCODE is set to -1002.

- Insert values are set in the corresponding row column values according to the rules for defining host variables.
- If the table identified by *table_name* is a view that was defined WITH CHECK OPTION, then the row to be inserted must be in the set of rows selected by the view.

Rules for Valued INSERT in Embedded SQL

The following rule applies to using the valued form of INSERT with embedded SQL:

- The colon is optional with host variables used in the VALUES clause.

Rules for Selected INSERT in Embedded SQL and Stored Procedures

The following rules apply to using the selected form of INSERT with embedded SQL and stored procedures:

- The number of rows in the temporary table returned by *subquery* determines the number of rows to be inserted.
- If an error occurs after one or more of the selected rows has been inserted, then the value -1002 is returned to SQLCODE and the current transaction is terminated with rollback.
- If *subquery* selects no rows, then the value +100 is returned to SQLCODE and no rows are inserted.

Examples

The following examples illustrate the use of INSERT.

Example 1

The following statement can be used to insert a row for the new employee Smith in the Employee table:

```
INSERT INTO Employee (Name,EmpNo,DeptNo,DOB,Sex,EdLev)
VALUES ('SMITH T',10021,700,460729,'F',16);
```

Example 2

This example uses a SELECT subquery to insert a row for every employee who has more than ten years of experience into a new table, Promotion, which contains three columns defined as Name, DeptNo, and YrsExp.

```
INSERT INTO Promotion (DeptNum, EmpName, YearsExp)
SELECT DeptNo, Name, YrsExp
FROM Employee
WHERE YrsExp > 10 ;
```

Example 3

The INSERT operation performed in the previous example can also be accomplished by the following statement. Note that a column name list is not given for the Promotion table; therefore, the Employee columns referenced in the SELECT subquery must match, in both quantity and sequence, the columns contained by Promotion.

```
INSERT INTO Promotion
SELECT Name, DeptNo, YrsExp
FROM Employee
WHERE YrsExp > 10 ;
```

Example 4

To add a row to the Employee table for new employee Orebo, you could enter the following statement:

```
INSERT INTO employee (name, empno, deptno, dob, sex, edlev)
VALUES ('Orebo B', 10005, 300, 'Nov 17 1957', 'M', 18) ;
```

In this example, you list the columns that are to receive the data, followed by the data to be inserted, in the same order as the columns are listed.

If you do not specify a field value for a named column then, then a null is inserted.

Example 5

You could achieve the same result using the statement:

```
INSERT INTO employee (10005, 'Orebo B',300,,,, 'Nov 17 1957',
'M',,,18,);
```

Using this form, you do *not* specify column names for fields because you enter the field data in the same order as the columns are defined in the Employee table. In all cases, a comma is required as a place marker for any field for which data is not specified.

For ANSI compliance, you should use the keyword NULL for nulls, for example, (10005, 'Ore60B', 300, NULL, NULL, NULL, 'Nov 17 1957', 'M', NULL, NULL, 18).

Example 6: INSERT and GENERATED ALWAYS Identity Columns

Assume that column_1 of newly created table_1 is an identity column defined as GENERATED ALWAYS. The following INSERT statements automatically generate numbers for column_1 of the inserted rows, even if they specify a value for that column.

```
INSERT INTO table_1 (column_1, column_2, column_3) VALUES
(111,111,111);

INSERT INTO table_1 (column_1, column_2, column_2) VALUES
(,222,222);
```

Check the inserts by selecting all the rows from table_1.

```
SELECT *
FROM table_1;

*** Query completed. 2 rows found. 3 columns returned.
*** Total elapsed time was 1 second.

 column_1  column_2  column_3
--------- --------- ---------
        2       111       111
        1       222       222
```

Note the following things about this result:

- Even though the value 111 was specified for column_1 in the first insert, the value is rejected and replaced by the generated value 2 because column_1 is defined as GENERATED ALWAYS.
- A warning is returned to the requestor when a user-specified value is overridden by a system-generated number.
- The first number in the identity column sequence is not necessarily allocated to the first row inserted because of parallelism.

Example 7: INSERT and GENERATED BY DEFAULT Identity Columns

The result of inserting values into the same table defined in “Example 6: INSERT and GENERATED ALWAYS Identity Columns” on page 3-117 demonstrates the difference in the behavior of the two identity column types. Because GENERATED BY DEFAULT only generates values when an INSERT statement does not supply them, the same insert operations produce different results.

Assume that column_1 of newly created table_1 is an identity column defined as GENERATED BY DEFAULT. The following INSERT statements generate numbers for column_1 of the inserted rows only if they do not specify a value for that column.

```
INSERT INTO table_1 (column_1, column_2, column_3) VALUES
(111,111,111);

INSERT INTO table_1 (column_1, column_2, column_2) VALUES
(,222,222);
```

Check the inserts by selecting all the rows from table_1.

```
SELECT *
FROM table_1;

*** Query completed. 2 rows found. 3 columns returned.
*** Total elapsed time was 1 second.

 column_1  column_2  column_3
--------- --------- ---------
      111       111       111
        1       222       222
```

Example 8: Identity Columns and INSERT … SELECT

Assume that column_1 of table_1 is a GENERATED BY DEFAULT identity column.

```
INSERT INTO table_2 (column_1, column_2, column_3) VALUES
(111,111,111);
INSERT INTO table_2 (column_1, column_2, column_3) VALUES
(222,222,222);
INSERT INTO table_2 (column_1, column_2, column_3) VALUES
(1,333,333);


INSERT INTO table_1 (column_1, column_2, column_3)
SELECT * FROM table_2;
```

Check the inserts by selecting all the rows from table_1.

```
SELECT *
FROM table_1;

*** Query completed. 3rows found. 3 columns returned.
*** Total elapsed time was 1 second.

 column_1  column_2  column_3
--------- --------- ---------
      111       111       111
      222       222       222
        1       333       333
```

LOCKING Modifier

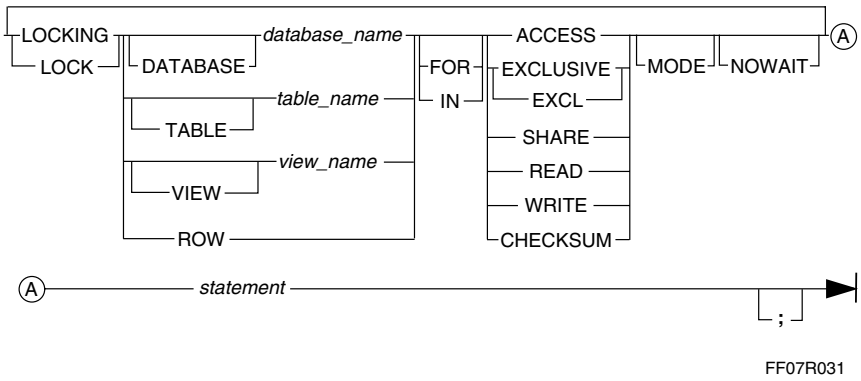
Purpose

Locks a database, table, view, or row hash, overriding the default usage lock that Teradata places on a database, table, view, or row hash in response to a request.

Invocation

Executable.

Syntax



where:

Syntax Element . . .	Specifies . . .
DATABASE TABLE VIEW	optional keywords that introduce the name of a database, table, or view that is to be locked
ROW	an optional keyword specifying a row hash to be locked in accordance with the defining *statement* (see "Using LOCK ROW" on page 3-123).
database_name	the name of a database (or user) that is to be locked.
table_name	the name of the table to be locked.
view_name	the name of the view to be locked.
FOR IN	an introduction to the type of lock to be placed.

Syntax Element . . .	Specifies . . .
ACCESS READ SHARE WRITE EXCLUSIVE CHECKSUM	the type of lock to be placed. The SHARE keyword can be used as a synonym for READ. CHECKSUM is only available for embedded SQL and stored procedure applications with updatable cursors.
MODE	an optional noise keyword.
NOWAIT	Indicates that if the indicated lock cannot be obtained, the statement should be aborted. Used for situations where it is not desirable to have a statement wait for resources, possibly also tying up resources in the process of waiting.
statement	an SQL DML statement. If *statement* is null, then the only effect of LOCK is to lock the specified object. This variable is optional for DATABASE, TABLE, and VIEW, but mandatory for ROW.

ANSI Compliance

LOCKING is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Lock Severity Types

You can specify the following types of lock severity:

Lock Type	Description
ACCESS	Permits selection of data from a table that can be locked for write access by other users. Because the data selected using ACCESS can be inconsistent because the data is concurrently being modified, this lock should be used only for casual inspection of data. Placing an ACCESS lock requires the SELECT privilege on the specified database or table.
READ	Ensures data consistency during a read operation (execution of a SELECT statement, for example). A number of users can hold a READ lock on a table at the same time. As long as a read lock is in place, no modification of the table is allowed. Placing a READ lock requires the SELECT privilege on the specified database or table.
WRITE	Enables a single user to modify data. As long as the WRITE lock is in place, all other users are locked out except readers who are viewing data using an ACCESS lock. Until a WRITE lock is released, no new READ locks are permitted. Placing a WRITE lock requires an UPDATE, INSERT, or DELETE privilege on the specified database or table.
EXCLUSIVE	Excludes all other users. This is the most restrictive lock. EXCLUSIVE locks are rarely used except to make structural changes to a database. Placing an EXCLUSIVE lock requires the DROP privilege on the specified database or table.
CHECKSUM	Used only with updatable cursors in embedded SQL and stored procedures.

LOCKING Modifier Use With Different DML Statements

The following table indicates associations between individual SQL DML statements and lock upgrades and downgrades at the row hash, view, and table database object levels:

This LOCKING modifier severity specification ...	Is available for this SQL DML statement ...
EXCLUSIVE	DELETE INSERT MERGE SELECT UPDATE
WRITE	SELECT
READ	SELECT
ACCESS	SELECT

The reason you can only specify the LOCKING FOR EXCLUSIVE modifier for the DELETE, INSERT, MERGE, and UPDATE statements is that the default lock severity for these statements is WRITE. You cannot downgrade a WRITE lock for these statements because doing so would compromise the integrity of the database. Because the SELECT statement does not change data, and therefore its actions cannot compromise database integrity, you are permitted to change its default locking severity to any other severity.

For details about locking levels, locking severities, and the relationship between them, see the chapter "Transaction Processing" in *Teradata RDBMS SQL Reference, Volume 2*.

Cancelling a Lock

The wait for a requested lock can be indefinite unless you specify the NOWAIT option. When you work interactively and do not want to wait on a lock, you can issue a BTEQ .ABORT command to cancel the transaction.

When Explicit Locking Is Necessary

Teradata automatically places usage locks each time an SQL statement is entered. Therefore, it is never necessary to use the LOCK modifier—it is only useful when you want to override the automatic locking of the Log Manager.

Positioning Non-Default Locks

When a lock other than the default lock is needed, the LOCK modifier should *precede* the SQL statement that is to be affected by the lock.

When the LOCK modifier is performed, Teradata places the specified lock on the object referenced by the modifier for the duration of the transaction containing the SQL statement.

If the transaction is a single-statement transaction, then the lock is only in effect while that statement is processed.

LOCK can be used to specify a database, table, or row hash lock.

Using LOCK ROW

Note that LOCK ROW does not generally lock a single row, but rather *all* rows that hash to a specific value. It is a row *hash* lock, not a *row* lock.

The LOCK ROW modifier cannot be used to lock multiple row hashes. If you specify LOCK ROW FOR ACCESS with multiple row hashes, the lock is converted to LOCK TABLE FOR ACCESS, but a WRITE (or higher) row hash lock severity is not transferred to a table level lock. If a LOCK ROW WRITE severity is not applicable, the default table level lock severity is maintained.

A single value in an IN clause is accepted as a valid row hash lock specification.

The use of LOCK ROW prevents possible deadlocks occurring when two simultaneous primary key SELECTs are followed by an UPDATE, DELETE, or INSERT on the same row.

For example:

User A:

```
BEGIN TRANSACTION;

SELECT y
FROM t
WHERE x=1;

UPDATE t
SET y=0
WHERE x=1;
```

User B:

```
BEGIN TRANSACTION;

SELECT z
FROM t
WHERE x=1;

UPDATE t
SET z=0
WHERE x=1;
```

The Lock Manager assigns a ROW READ LOCK when a simple SELECT is done on a Unique Primary Index, Primary Index, or Unique Secondary Index.

The User A UPDATE statement ROW WRITE LOCK request waits for the User B ROW READ LOCK to be released. The ROW READ LOCK is not released because the User B UPDATE statement ROW WRITE LOCK request is also waiting for the User A ROW READ LOCK to be released. Previously, this deadlock was avoided by using the LOCK TABLE modifier:

```
BEGIN TRANSACTION;

LOCK TABLE t FOR WRITE
SELECT z
FROM t
WHERE x = 1;

UPDATE …
```

Locking an entire table across all AMPS is undesirable, and the use of LOCK ROW here prevents the need to lock an entire table across all AMPs.

The following example illustrates the use of LOCK ROW:

User A:

```
BEGIN TRANSACTION;

LOCK ROW WRITE
SELECT y
FROM t
WHERE x=1;

UPDATE t
SET y=0
WHERE x=1

END TRANSACTION;
```

User B:

```
BEGIN TRANSACTION;

LOCK ROW WRITE
SELECT z
FROM t
WHERE x=1;

UPDATE t
SET z=0
WHERE x=1;
```

No deadlock occurs because the User B LOCK ROW WRITE request is blocked by the User A LOCK ROW WRITE. The User B LOCK ROW WRITE statement completes when the User A END TRANSACTION statement is complete.

The LOCK ROW modifier works only in the following situations.

- For single table retrievals, using primary (NUPIs or UPIs) or unique secondary index (USI) searches.
- When specified with a SELECT statement, because the LOCK ROW modifier picks up the row hash to be locked from the SELECT statement.
- To upgrade any of the following locks

Upgrade from this lock . . .	To this lock . . .
READ	WRITE
READ	EXCLUSIVE
WRITE	EXCLUSIVE

The LOCK ROW modifier does *not* work in the following situations.

- Where a lock on the target table is already in place.
- Where an aggregate/DISTINCT/GROUP BY operation is part of the SELECT statement.
- To downgrade a lock from *READ* to *ACCESS,* once that lock is in place.

  If no row hash lock is in place already, an ACCESS lock can be set instead of a default READ row hash lock.

When two primary key SELECTS are followed by primary key UPDATES and the SELECTs are on different row hashes, deadlock occurs. This is because a primary index update requires a table write lock on the target table.

For example:

User A:

```
BEGIN TRANSACTION;

LOCK ROW WRITE
SELECT x
FROM t
WHERE x = 1; (x is UPI, NUPI, or USI)

UPDATE t
SET x=2
WHERE x=1;
```

User B:

```
BEGIN TRANSACTION;

LOCK ROW WRITE
SELECT x
FROM t
WHERE x=2;

UPDATE t
SET x=1
WHERE x=2;
```

The User B SELECT ROW WRITE LOCK is not queued behind the User A transaction because it has a different row hash access. Deadlock occurs because the User A table lock request waits on the User B row hash lock, while the User B table lock request waits on the User A row hash lock.

Using Locks with NULL SQL Statements

If the LOCK modifier is followed by a null SQL statement, the only effect of the modifier is to lock the specified object. While the LOCK modifier can be used with a null statement, it is best to use LOCK with a user-generated transaction or as part of a multistatement request.

Multiple Locks

Multiple LOCK modifiers can precede a statement if it is necessary to lock more than one object at the same time.

```
LOCK TABLE Employee FOR ACCESS
LOCK TABLE Department FOR ACCESS
SELECT Name, Loc
FROM Employee, Department
WHERE (EmpNo=MgrNo);
```

Referencing a Locked Object

The object that is locked by the LOCK modifier does not have to be referenced in a subsequent SQL statement.

A LOCK can be executed separately from the SQL statement that it precedes. Therefore, a LOCK modifier must reference the object on which a lock is being placed. The objects referenced in the SQL statement have no effect on the execution of a LOCK modifier.

When a LOCK modifier references a view, the specified lock is applied to all underlying base tables. For example, if a view refers to tables T_1 and T_2, then a lock on that view would apply to both tables.

LOCK DATABASE is the only way to lock a database or a user.

Include the Keyword for the Locked Object

Be sure to include the keyword for the entity (DATABASE, TABLE, or VIEW) that is to be locked.

For example, if there is a database and a table, and if both are named AccntRec, then the following form could be used:

```
LOCK TABLE AccntRec FOR ACCESS SELECT * FROM AccntRec;
```

If the TABLE keyword was not included, the lock would have been placed on the AccntRec database.

Locks and Views

A LOCK modifier can be included in a view definition. When a view is defined with a LOCK modifier, the specified lock is placed on the underlying table set each time the view is referenced in an SQL statement.

See “CREATE VIEW” in the chapter “SQL Data Definition Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 4* for more information on defining views.

When Both the Request and View Referenced Include LOCK Clauses

Although views are often created to enforce a LOCK FOR ACCESS rule, any user can override the LOCK FOR ACCESS by using a LOCK FOR READ statement on the view. For example:

```
REPLACE VIEW VPROD.AD_ME_INF AS
LOCK TABLE PROD.AD_ME_INF FOR ACCESS
SELECT AD_ME_ID, AD_ME_DSC_TX
FROM PROD.AD_ME_INF;
```

If you do an EXPLAIN on the following query, the access lock can be seen in statement 1.

```
SELECT COUNT(*)
FROM VPROD.AD_ME_INF;
```

If you do an EXPLAIN on the following query, you can see a read lock in statement 1 of the report.

```
LOCK TABLE VPROD.AD_ME_INF FOR READ
SELECT COUNT (*)
FROM VPROD.AD_ME_INF;
```

This behavior could be considered undesirable because the LOCK FOR ACCESS statement can be overridden by anyone at any time. However, some users find this to be useful and depend on being able to override lock clauses in views by placing a lock in the request.

Determining What Locks Have Been Set

Use the EXPLAIN modifier at the beginning of an SQL statement to determine what locks are set when the statement is executed.

Users with the performance monitor MONITOR SESSION privilege can view all locks in the system through use of the Performance Monitor/Application Programming Interface (PM/API) and can determine which other user locks are blocking their own.

Examples

The following examples illustrate the use of LOCK.

Example 1

The following LOCK clause can be used to select data from the Employee table while the table is being modified:

```
LOCK TABLE Personnel.Employee IN ACCESS MODE
SELECT Name, Salary
FROM Employee
WHERE Salary < 25000  ;
```

This query might do any of the following things.

- Return rows whose data can be updated or deleted an instant later by a concurrent operation initiated by another user who has obtained a WRITE lock.
- Omit rows that are undergoing a concurrent insert operation.
- Include rows that were never permanently in the table, because a transaction that was being used to insert new rows was aborted and its new rows backed out.

Example 2

Teradata deals properly with the synchronization of base data rows and index subtable rows. However, an ACCESS lock can allow inconsistent results even when secondary indexes are used in conditional expressions because index constraints are not always rechecked against the data row.

For example, if a column named "QualifyAccnt" is defined as a secondary index for a table named "AccntRec", the following request:

```
LOCK TABLE AccntRec FOR ACCESS
SELECT AccntNo, QualifyAccnt
FROM AccntRec
WHERE QualifyAccnt = 1587;
```

could return:

```
AccntNo     QualifyAccnt
-------    -------------
   1761             4214
```

In this case, the value 1587 was found in the secondary index subtable and the corresponding data row was selected and returned. However, the data for account 1761 had been changed by another user while the selection was in process.

Returns such as this are possible even if the data is changed only momentarily by a transaction that is ultimately aborted. The ACCESS lock is most useful to those who simply want an overview of data and are not concerned with consistency.

Example 3: LOCK ROW

This example demonstrates the proper use of a row hash lock.

```
CREATE TABLE customer
 (cust_id INTEGER,
  phone   INTEGER,
  fax     INTEGER,
  telex   INTEGER)
PRIMARY INDEX (cust_id),
UNIQUE INDEX(fax),
INDEX(telex):

CREATE TABLE sales
 (custcode INTEGER,
  zip      INTEGER,
  salesvol INTEGER);
```

User A:

```
BEGIN TRANSACTION;

LOCK ROW EXCL
SELECT phone
FROM customer
WHERE cust_id=12345;

UPDATE customer
SET phone=3108292488
WHERE cust_id=12345;
```

The User A row hash lock prevents another user from accessing the same record.

In the following, the user A row hash lock, in conjunction with LOCK TABLE, prevents user B from accessing the same record.

```
LOCK TABLE sales FOR READ,
LOCK ROW FOR WRITE
SELECT telex
FROM customer
WHERE fax=0;

UPDATE customer
SET telex=0
WHERE fax=0;

SELECT zip
FROM sales
WHERE custcode=111;

SELECT salesvol
FROM sales
WHERE custcode=222;

…

END TRANSACTION;
```

User B:

```
BEGIN TRANSACTION;

LOCK ROW WRITE
SELECT *
FROM customer
WHERE cust_id=12345
INSERT INTO customer (12345, 3108284422, 3108684231, 5555);

END TRANSACTION;
```

The User B LOCK ROW statement waits until the User A transaction ends before it can be completed.

Further Information

See the chapter "Transaction Processing" in *Teradata RDBMS SQL Reference, Volume 2* for additional information about locks.

MERGE

Purpose

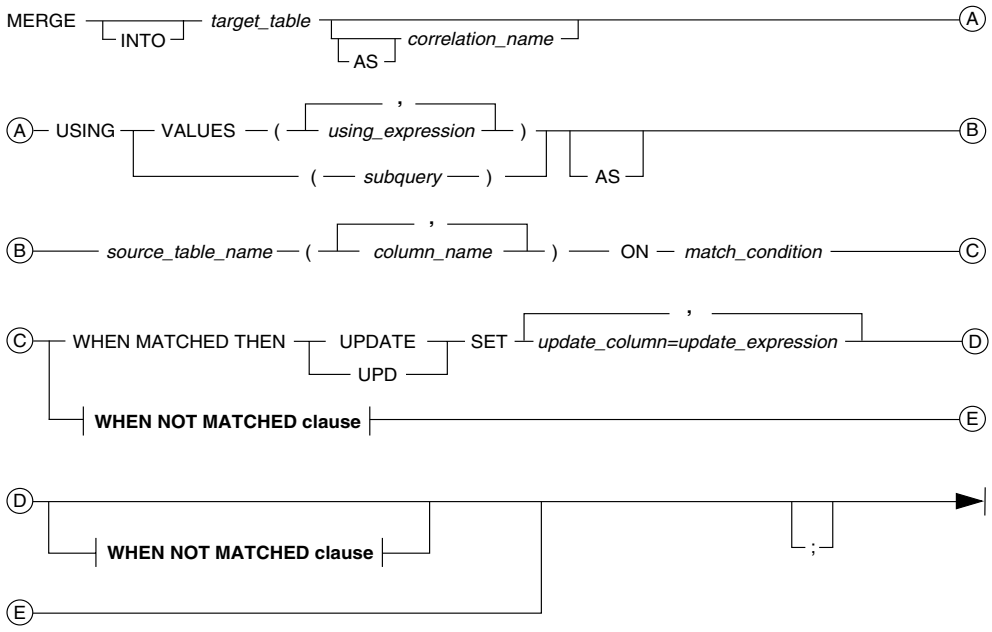
Merges a source row into a target table based on whether any target rows satisfy a specified matching condition with the source row.

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.

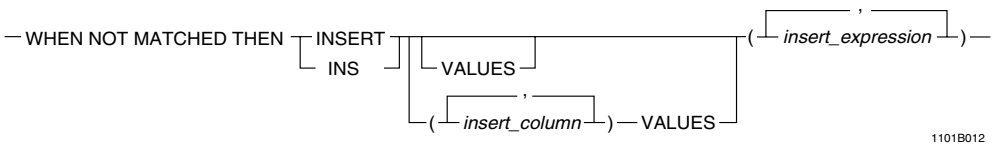
Invocation

Executable.

Syntax



WHEN NOT MATCHED clause



where:

Syntax element …	Specifies …
target_table	the table to be updated.
correlation_name	an optional alias for *target_table*.
USING	an introduction to the table expression that defines the single row source table for the merge.
using_expression	an expression defining one field of the table expression that specifies the source table row for the merge as a list of *using_expression* values. The table expression representing the source table row is composed of the comma-separated list of *using_expression* values. It is not a single value expression. *using_expression* can reference any of the following: • constants • built-in functions • USING row descriptor variables • macro parameters • stored procedure parameters • host variables *using_expression* cannot reference columns in any base table, including *target_table*.
subquery	a singleton subquery table expression that specifies the source table for the merge with a cardinality of at most one row. You can specify host variables in the WHERE clause, but *not* in the select list, of *subquery*. All host variables must be preceded by a COLON character. If *subquery* returns no rows, the source table is empty, no triggers are fired (see "MERGE As a Triggering Action" on page 3-137), and the merge operation performs neither update nor insert operations on *target_table*. *subquery* can reference any of the following: • rows in the queried table • constants • USING row descriptor variables The WHERE clause for *subquery* must include an equality condition on the UPI or USI of the queried table to ensure a singleton select.

Syntax element …	Specifies …
source_table_name	the correlation name of the single row source table to be merged into *target_table*.
column_name	the name to be used for the source row column defined by the corresponding *using_expression* or *subquery* select list expression. Source row columns, qualified by *source_table_name*, can be referenced in *match_condition* or in an *update_expression* or *insert_expression*.
match_condition	a conditional expression that determines whether the source row matches a given row in the target table. If the condition is met for any target rows and a WHEN MATCHED clause is specified, then the matching target rows are updated. *match_condition* must fully specify the primary index of *target_table* to ensure that the candidate target row set can be hash-accessed on a single AMP and it cannot refer to any column in a USING (*subquery*) clause whose value derives from a subquery. Host variables are permitted in *match_condition*. All host variables must be preceded by a COLON character. *match_condition* cannot reference a table that is neither the target table nor the source table.
WHEN MATCHED THEN UPDATE SET	an introduction to an update set clause operation to be performed for matching rows. If both a WHEN MATCHED THEN clause and a WHEN NOT MATCHED THEN clause are specified in the same statement, then you must specify the WHEN MATCHED THEN clause first.
update_column = *update_expression*	an equality condition on a target table column (specified by *update_column*) that defines how the specified field is to be updated. *update_expression* produces a new value to be placed into the field to be updated. *update_expression* can include a constant, a null (expressed by the reserved word NULL), host variables, or an arithmetic expression for calculating the new value. All host variables must be preceded by a COLON character.

Syntax element …	Specifies …
WHEN NOT MATCHED THEN INSERT	an introduction to a value list to be inserted for nonmatching rows. If both a WHEN MATCHED THEN clause and a WHEN NOT MATCHED THEN clause are specified in the same statement, then you must specify the WHEN MATCHED THEN clause first. Because an inserted row might be a duplicate of an existing row in *target_table*, its acceptance depends on whether *target_table* is defined to be SET or MULTISET.
insert_column	a column name into which a corresponding value in the value list is to be inserted for nonmatching rows.
insert_expression	a value to be inserted into a corresponding *insert_column* for nonmatching rows. Host variables are permitted in *insert_expression*. All host variables must be preceded by a COLON character.

ANSI Compliance

MERGE is a Teradata extension to the ANSI SQL-99 standard.

The Teradata MERGE statement is a partial implementation of the anticipated full ANSI SQL-99 specification for MERGE.

Authorization

The privileges required to perform MERGE depend on the merge matching clause of the statement you submit.

IF you specify this type of merge matching clause …	THEN you must have this privilege …
WHEN MATCHED	UPDATE on every column of *target_table* that is specified in the UPDATE SET set clause list.
WHEN NOT MATCHED	INSERT on *target_table.* The INSERT privilege is also required on all of the columns specified in the INSERT column list.
both	all the update and insert privileges required for the WHEN MATCHED and WHEN NOT MATCHED clauses

Rules

The following set of rules applies to MERGE:

- Only single row statements are supported. The meaning of "single row" is complex and refers to the following set of limitations:
  - The specified *match_condition* must specify a single primary index value for the target row.
  - If you specify a subquery for *source_table_reference*, then that subquery must be a singleton SELECT (that is, it cannot retrieve more than a single row from the referenced table) and it must specify either a UPI or USI to make the retrieval.

    The selected columns must be referenced explicitly: the `SELECT *` syntax is not permitted.
  - If you specify a simple list of column values for *source_table_reference*, then the retrieval, by definition, retrieves only a single row.
  - This set of rules means that all cases of the USING *source_table_reference* clause must define a source table with only one row to be applied either to the update specified by the WHEN MATCHED THEN clause or to the insert specified by the WHEN NOT MATCHED THEN clause.
- Only those rows that exist in the target table before MERGE starts to perform are candidates for being updated.
- The *match_condition* must fully specify the primary index of *target_table*. If a subquery is specified, then it cannot reference columns derived from the subquery table.
- If you specify a WHEN MATCHED THEN and a WHEN NOT MATCHED THEN in the same MERGE statement, then you must specify the WHEN MATCHED THEN clause first.
- MERGE can be prepared as dynamic SQL and performed dynamically.
- MERGE cannot be specified as a triggered action.
- The target table in a MERGE statement can be an identity column table. Teradata generates numbers for MERGE inserts into the identity column in the same way it performs singleton inserts into tables that do not have an identity column.
- You can specify host variables for the WHERE clause of the USING *subquery* clause (but not in its select list), *match_condition*, *update_expression*, and *insert_expression*.

  All host variables must be preceded by a COLON character.

Rules for Using MERGE With PPI Tables

The following rules apply to using the MERGE statement to insert rows into a partitioned primary index table or updating the columns of a PPI table partitioning expression.

- If a MERGE attempts to update the partitioning columns of a row of a table with a partitioned primary index such that the partitioning expression does not generate, after casting to INTEGER if it is not already INTEGER, a non-null value between 1 and 65 535, an error is reported.
- If a MERGE attempts to insert a row into a table with a partitioned primary index such that the partitioning expression for that row does not generate, after casting to INTEGER if it is not already INTEGER, a non-null INTEGER value between 1 and 65 535, an error is reported.

MERGE As a Triggering Action

If *subquery* returns no rows, then no triggers are fired.

Triggers invoke the following behavior when fired as the result of a MERGE statement:

IF a trigger is defined on this type of action ...	THEN it is fired when this clause is specified ...	AND *match_condition* is ...
UPDATE	WHEN MATCHED THEN	either met or not met.
INSERT	WHEN NOT MATCHED THEN	not met.

MERGE As a Triggered Action

MERGE is not supported as a triggered action.

Example 1

This example uses dynamically supplied values for an Employee table row to update the table if the data matches an existing employee or insert the new row into the table if the data does not match an existing employee. Column empno is the unique primary index for the Employee table.

```
USING (empno  INTEGER,
       name   VARCHAR(50),
       salary INTEGER)
MERGE INTO employee AS t
USING VALUES (:empno, :name, :salary) AS s(empno, name, salary)
ON t.empno = s.empno
WHEN MATCHED THEN UPDATE
 SET salary = s.salary
WHEN NOT MATCHED THEN INSERT (empno, name, salary)
 VALUES (s.empno, s.name, s.salary);
```

This example could also be coded as the following upsert form of the UPDATE statement (see "UPDATE (Upsert Form)" on page 3-160):

```
USING (empno  INTEGER,
       name   VARCHAR(50),
       salary INTEGER)
UPDATE  employee
SET salary = :salary
WHERE empno = :empno
ELSE INSERT INTO employee
(empno, name, salary) VALUES ( :empno, :name, :salary);
```

Example 2

This example generalizes "Example 1" on page 3-138 by using a subquery for *source_table_reference* rather than an explicit value list.

```
USING (empno  INTEGER,
       salary INTEGER)
MERGE INTO employee AS t
USING (SELECT :empno, :salary, name
       FROM names
       WHERE empno = :empno) AS s(empno, salary, name)
ON t.empno = s.empno
WHEN MATCHED THEN UPDATE
 SET salary = s.salary, name = s.name
WHEN NOT MATCHED THEN INSERT (empno, name, salary)
 VALUES ( s.empno, s.name, s.salary);
```

ORDER BY Clause

Specifies how result data in a SELECT statement is to be ordered.

See "ORDER BY Clause" on page 1-77 for a detailed description of the ORDER BY clause.

QUALIFY Clause

Eliminates rows from a SELECT query based on the value of a computation.

Unlike the SELECT HAVING clause, the QUALIFY clause operates on statistical data produced by ordered analytical functions rather than on a conditional expression.

See "QUALIFY Clause" on page 1-64 for a detailed description of the QUALIFY clause.

ROLLBACK

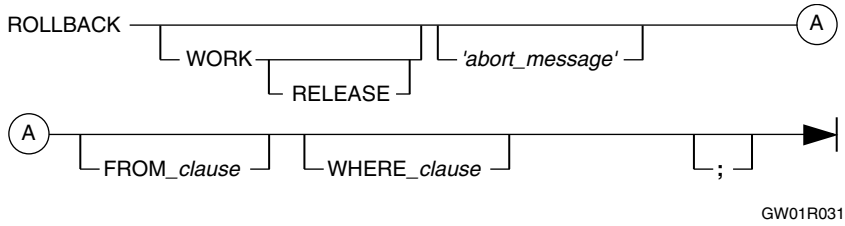
Purpose

Terminates and rolls back the current transaction.

Invocation

Executable.

Syntax



where:

Syntax Element . . .	Specifies . . .
WORK	an optional noise keyword.
RELEASE	that the connection between the application program and Teradata is to be terminated for a successful ROLLBACK.
'abort_message'	the text of the message to be returned when the transaction is terminated.
FROM_clause	a clause that is required only if *WHERE_clause* includes subqueries. *FROM_clause* is described in detail in "FROM Clause" on page 1-22.
WHERE_clause	an optional clause that specifies an expression whose result must evaluate to TRUE if the transaction is to be rolled back. If the result of *WHERE_clause* evaluates to FALSE, then transaction processing continues. If you do not specify *WHERE_clause*, then rollback occurs unconditionally. The abort condition described by *WHERE_clause* can also specify an aggregate operation (see Usage Notes) under ABORT. *WHERE_clause* is described in detail in "WHERE Clause" on page 1-32.

ANSI Compliance

ROLLBACK is ANSI SQL-99-compliant.

The following ROLLBACK options are Teradata extensions to the ANSI standard:

- RELEASE
- *abort_message*
- *FROM_clause*
- *WHERE_clause*

Authorization

None.

ROLLBACK With Correlated Subqueries

See "Correlated Subqueries and the ABORT/ROLLBACK Statements" on page 1-56 for information about using correlated subqueries with ROLLBACK statements.

Rules for Embedded SQL

ROLLBACK observes the following rules:

- ROLLBACK cannot be performed as a dynamic SQL statement.
- ROLLBACK is not valid when you specify the TRANSACT(2PC) option to the preprocessor.

ROLLBACK and ABORT Are Synonyms

ROLLBACK is a synonym for ABORT.

The COMMIT statement also causes the current transaction to be terminated, but with commit rather than rollback.

See "ABORT" on page 3-3 and "COMMIT" on page 3-33.

How ROLLBACK Works

ROLLBACK performs the following actions:

Stage	Process
1	Backs out changes made to the database as a result of the transaction.
2	Deletes spooled output for the request.
3	Releases locks associated with the transaction.
4	If the transaction is in the form of a macro or a multistatement request, bypasses execution of the remaining statements.
5	Returns a failure response to the user.

How ROLLBACK Works With Embedded SQL

ROLLBACK performs the following additional actions when used within an embedded SQL application:

Stage	Process
1	See "How ROLLBACK Works" on page 3-143.
2	Discards dynamic SQL statements prepared within the current transaction.
3	Closes open cursors.
4	Cancels database changes made by the current transaction.
5	Sets SQLCODE to zero when ROLLBACK is successful.
6	Sets the first SQLERRD element of the SQLCA to 3513 or 3514, depending on whether *abort_message* is specified.
7	Returns an abort message in the SQLERRM field of the SQLCA if a *WHERE_clause* is specified.
8	Terminates the application program connection (whether explicit or implicit) to Teradata if the RELEASE keyword is specified.

IN this environment . . .	IF no LOGON/CONNECT request precedes the next SQL statement, THEN the preprocessor . . .
Channel-attached	attempts to establish an implicit connection.
Network-attached	issues a No Session Established error.

ROLLBACK With WHERE Clause

ROLLBACK tests each value separately, so a WHERE clause should not introduce both an aggregate and a nonaggregate value.

The aggregate value becomes, in effect, a GROUP BY value, and the mathematical computation is performed on the group aggregate results.

For example, assuming that the following things are true, then the ROLLBACK statement that follows incorrectly terminates the transaction.

- The table Test contains several rows,
- The sum of Test.colA is 188, and
- Only one row contains the value 125 in Test.colB

```
ROLLBACK
WHERE (SUM(Test.colA) <> 188)
AND (Test.ColB = 125);
```

The preceding statement is processed first by performing an all-rows scan with the condition (ColB = 125), which selects a single row and then computes intermediate aggregate results with the condition (SUM(ColA) <> 188).

The condition evaluates to TRUE because the value of ColA in the selected row is less than 188.

If ROLLBACK ... WHERE is used and it requires READ access to an object for execution, then user executing the ROLLBACK statement must have SELECT right to the data being accessed.

The WHERE condition of a ROLLBACK can include subqueries. If so, then the subqueries require FROM clauses, and the ROLLBACK statement should also have a FROM clause if you want the scope of a reference in a subquery to be the ROLLBACK condition.

See "Correlated Subqueries" on page 1-41 and "ABORT" on page 3-3.

ROLLBACK With BTEQ

If you use ROLLBACK in a BTEQ script with either the .SESSION or the .REPEAT command, you must send the ROLLBACK statement and the repeated SQL statement as one request.

If you send the repeated request without the ROLLBACK, one of the requests is eventually blocked by other sessions and the job hangs because of a deadlock.

Related Information

See "COMMIT" on page 3-33.

SAMPLE Clause

Selects rows from a SELECT query for further processing to satisfy a conditional expression expressed in a WHERE clause.

See "SAMPLE Clause" on page 1-67 for a detailed discussion of the SAMPLE clause.

SELECT

Returns selected rows in the form of a result table.

See Chapter 1: “The SELECT Statement” for more detail.

SELECT … INTO

Returns a selected row from a table and assigns its values to host variables.

See Chapter 1: “The SELECT Statement” for more detail.

UPDATE (Searched Form)

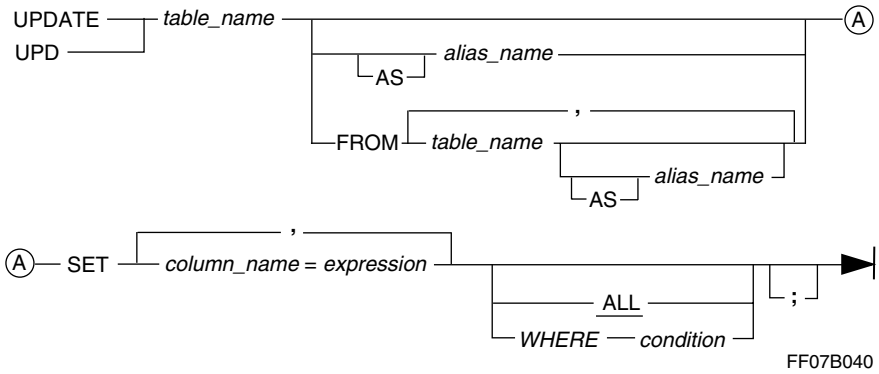
Purpose

Modifies field values in existing rows of a table.

Invocation

Executable.

Syntax



where:

Syntax Element …	Specifies …
table_name	the name of the table to be updated, or the name of a view through which the table is accessed.
AS	an optional keyword introducing *alias_name*.
alias_name	the alias name of a temporary table whose rows are to be updated during a self-join of an actual table (*tname*). The *alias_name* option is a Teradata extension to ANSI SQL. ANSI calls table aliases correlation names. They are also referred to as range variables.
FROM	a keyword introducing the names of one or more tables or view from which *expression* is to be derived. The FROM syntax is a Teradata extension to ANSI SQL.
table_name	the name of a table, view or macro. If database of omitted, it is inferred from context.
AS	an optional keyword introducing *alias_name*.

Syntax Element …	Specifies …
alias_name	a different, temporary name (alias) for the table that is referenced by *table_name*. *alias_name* must be used during a self-join operation on the table. ANSI calls table aliases correlation names. They are also referred to as range variables.
SET	the names of one or more columns whose data is to be updated, and the expressions that are used for update.
column_name	the column name. The *column_nam*e field is for the column name only. Do not use fully-qualified forms such as Databasename.Tablename.Columnname, or Tablename.Columnname.
expression	an expression that produces a new value to be placed into the column to be updated. *expression* can include a constant, a null (expressed by the reserved word NULL), or an arithmetic expression for calculating the new value. Values in a targeted row before the update can be referenced in an expression. When host variables are used in the SET clause, they must always be preceded by a COLON character.
WHERE	a conditional clause. For more information see "WHERE Clause" on page 1-32. If you specify a WHERE clause, you must have SELECT access on the searched objects.
condition	the conditional expression to be used for determining rows whose values are to be updated. The conditional clause can reference more than one table, or specify an embedded select operation.
ALL	Indicates that all rows in the specified table are to be updated. The ALL option is a Teradata extension to ANSI SQL.

ANSI Compliance

UPDATE is ANSI SQL-99-compliant.

An UPDATE that contains DEFAULT is intermediate-level ANSI-compliant.

Authorization

The following privilege rules apply to the UPDATE statement.

- You must have the UPDATE privilege on the table or columns to be updated.
- When executing an UPDATE that also requires READ access to an object, you must have the SELECT privilege on the data being accessed.

  For example, in the following statement, READ access is required by the WHERE condition.

  ```
  UPDATE table_1
  SET field_1 = 1
  WHERE field_1 < 0;
  ```

  Similarly, the following statement requires READ access because you must read f1 values from t1 in order to compute the new values for f1.

  ```
  UPDATE table_1
  SET field_1 = field_1 + 1;
  ```

UPDATE With Correlated Subqueries

See "Correlated Subqueries and the UPDATE Statement" on page 1-52 for information about using correlated subqueries with UPDATE statements.

Locks Set by UPDATE

An UPDATE operation sets a WRITE lock for the table or row being updated.

Activity Count

The activity count in the success response (or ACTIVITY_COUNT system variable for a stored procedure) for an UPDATE statement reflects the total number of rows updated. If no rows qualify for update, then the activity count is zero.

Duplicate Rows and UPDATE

It is not possible to distinguish among duplicate rows in a MULTISET table. Because of this, when a WHERE condition identifies a duplicate row, then all of the duplicate rows are updated.

Duplicate Row Checks

Unless a table is created as MULTISET (without UNIQUE constraints) to allow duplicate rows, Teradata always checks for duplicate rows during the update process. The order in which updates are executed can affect the result of a transaction.

Consider the following example:

```
CREATE SET TABLE t1
 (a INTEGER,
  b INTEGER)
PRIMARY INDEX (a);

INSERT INTO t1 VALUES (1,1);

INSERT INTO t1 VALUES (1,2);

UPDATE t1
SET b = b + 1
WHERE a = 1; /* fails */

UPDATE t1
SET b = b - 1
WHERE a = 1; /* succeeds */
```

The first UPDATE statement fails because it creates a duplicate row.

If the order of the UPDATE statements is reversed, then both UPDATE statements succeed because the UPDATE does not result in duplicate rows.

```
CREATE SET TABLE t1
 (a INTEGER,
  b INTEGER)
PRIMARY INDEX (a);

INSERT INTO t1 VALUES (1,1);

INSERT INTO t1 VALUES (1,2);

UPDATE t1
SET b = b - 1
WHERE a = 1; /* succeeds */

UPDATE t1
SET b = b + 1
WHERE a = 1; /* succeeds */
```

This mode is characteristic of simple updates only. Join updates, or updates that affect primary or secondary index values are implemented as discrete delete and insert operations.

UPDATE Processing Time

Processing time for an UPDATE operation is longer under the following conditions:

- When the FALLBACK option is specified for the table, because the rows in the secondary copy of the table must also be updated.
- When a column on which one or more indexes (secondary or primary) are defined is updated.

You can shorten the processing time for an UPDATE operation by using an indexed column in the WHERE clause of the UPDATE statement.

Rules for Embedded SQL and Stored Procedures

The following rules apply to the searched form of the UPDATE statement:

- If host variable references are used, a COLON character must precede the host variable in the SET and WHERE clauses. The colon is optional otherwise, but strongly recommended.
- If the UPDATE statement is specified with a WHERE clause and the specified search condition selects no rows, then the value +100 is assigned to SQLCODE and no rows are updated.
- The updated row is formed by assigning each update value to the corresponding column value of the original row. The resulting updated row must be NULL for any column that has the NOT NULL attribute or SQLCODE is set to -1002 and the row is not updated.

  Update values are set in the corresponding row column values according to the rules for host variables.
- If the table identified by *table_name* is a view specifying WITH CHECK OPTION, all values updated through that view must satisfy any constraints specified in any WHERE clause defined in the view.

Rule for Updating Partitioning Columns of a PPI Table

If you are updating a partitioning column for a partitioned primary index, then updates to the partitioning columns must be in a range that permits the partitioning expression to produce, after casting values to INTEGER if the value is not already of that type, a non-null value between 1 and 65 535.

Identity Columns and UPDATE

You cannot update a GENERATED ALWAYS identity column.

Update of GENERATED ALWAYS Identity Columns and PARTITION Columns Is Not Permitted

You cannot update the following set of system-generated columns:

- GENERATED ALWAYS identity columns
- PARTITION

You can update a GENERATED BY DEFAULT identity column. The specified value is not constrained by identity column parameters, but is constrained by any CHECK constraints defined on the column.

Rules for Updating Rows Using Views

To update rows using a view through which the table is accessed, the following rules must be observed:

- You must have the UPDATE privilege on the view.

  The immediate owner of the view (that is, the database in which the view resides) must have the UPDATE privilege on the underlying object (view, base table, or columns) whose columns are to updated, and the SELECT privilege on all tables that are specified in the WHERE clause.
- Each column of the view must correspond to a column in the underlying table (that is, none of the columns in the view can be derived using an expression).
- The data type definitions for an index column should match in the view definition and in the base table to which the view refers.

  While it is true that you can generally convert the data type of a view column (for example, from VARCHAR to CHAR), if that converted column is a component of an index, then the Optimizer does not use that index when the base table is updated because the data type of the recast column no longer matches the data type of the index column.

  The resulting all-AMP, all-row scan defeats the performance advantages the index was designed for.
- No two view columns can reference the same table column.
- If the statement used to define a view contains a WHERE clause WITH CHECK OPTION, then all values inserted through that view must satisfy the constraints specified in the WHERE clause.
- If a view includes a WHERE clause and does not specify WITH CHECK OPTION, then data can be inserted through the view that will not be visible through that view.

Non-Valid Uses of UPDATE

An UPDATE operation causes an error message to be returned when any of the following conditions occur:

- The operation attempts to update a field using a value that violates a constraint (for example, UNIQUE or referential) declared for the column.
- The operation attempts to update a field using a value that is of a different numeric type than that declared for the column and the value cannot be converted correctly to the correct type.
- The operation attempts to update a VARCHAR field, and that operation causes the row to become identical to another row (except for the number of trailing pad characters), for a table not permitting duplicate rows.
- The operation attempts to update a CHARACTER field with a value that is not in the repertoire of the destination character data type.
- If in ANSI mode, updating character data, where in order to comply with maximum length of the target column, non-pad characters are truncated from the source data. This update is valid in Teradata mode.
- The operation attempts to update a row by using values that will create a duplicate of an existing row, for a table not allowing duplicate rows.
- The operation references objects in multiple databases without using fully qualified names. Name resolution problems may occur if referenced databases contain tables or views with identical names and these objects are not fully qualified. Name resolution problems may even occur if the identically named objects are not themselves referenced.

See "Examples" on page 3-156.

UPDATES and Joins

When an UPDATE statement specifies a join operation, the join is more efficient if the WHERE condition uses values for indexed columns in the joined tables.

See "WHERE Clause" on page 1-32 for a discussion of using indexes to join expression performance.

FROM Clause

The optional FROM list included in the UPDATE syntax is a Teradata extension to the ANSI extension provided to support correlated subqueries and derived tables in the search conditions for UPDATE.

Use the FROM clause only for these reasons:

- to provide the outer scope of a table with fields referenced in a subquery, making the subquery a correlated subquery
- to permit reference to a derived table

If a table is listed in the FROM clause for the UPDATE and not in the FROM clause for a subquery, then field references in the subquery are scoped at the outer level, making it a correlated subquery.

Subqueries in UPDATE Statements

The condition of an UPDATE statement can include subqueries, referencing the update table and any other tables.

Example:

```
UPDATE Publisher
SET pubnum = NULL
WHERE 0 =
 (SELECT COUNT(*)
  FROM book
  WHERE book.PubNum = Publisher.pubnum);
```

There are two publishers who have books in the library, and two publishers who do not.

The correlated subquery is executed once for each row of the outer reference, Publisher, and since two publishers have not books in the library two rows of publisher are modified.

To modify this UPDATE to be a noncorrelated subquery change the subquery to include all tables it references in the inner FROM clause.

```
UPDATE Publisher
SET PubNum = NULL
WHERE 0 =
 (SELECT COUNT(*)
  FROM book, publisher
  WHERE book.PubNum = Publisher.pubnum);
```

Now, the condition in the subquery has a local defining reference and the statement does not contain a correlated subquery. The count, determined once, is nonzero, and no rows are deleted.

The sections on correlated subqueries in Chapter 1: “The SELECT Statement” describe additional examples of correlated and noncorrelated subqueries.

Examples

The following examples illustrate the use of UPDATE:

Example 1

The following statement can be used to update the YrsExp column in the Employee table for Greene, who has an EmpNo of 10017:

```
UPDATE Employee
SET YrsExp = 16
WHERE EmpNo = 10017;
```

Example 2

The following statement can be used to update the Employee table to apply a 10 percent cost of living increase for all employees:

```
UPDATE Employee
SET Salary = Salary * 1.1 ALL ;
```

Example 3

The following statement can be used to place a null in the salary column for the employee named Peterson:

```
UPDATE EMployee
SET Salary = NULL
WHERE EmpNo = 10001 ;
```

Example 4

```
UPDATE table_1
SET field_1 = 1
WHERE field_1 IN
 (SELECT field_2
  FROM table_2);

UPDATE table_1
SET field_1 = 1
WHERE table_1.field_1=table_2.field_1;
```

UPDATE (Positioned Form)

Purpose

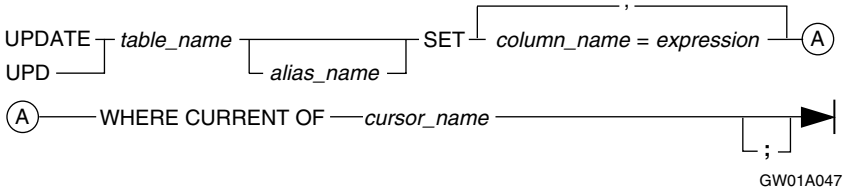
Updates the most current fetched cursor row.

Invocation

Executable.

Embedded SQL and stored procedures only.

Syntax



where:

Syntax element …	Specifies …
table_name	the name of the table to be updated.
alias_name	an alias for the table name. ANSI calls table aliases correlation names. They are also referred to as range variables.
column_name = *expression*	a column name and value with which to update. When host variables are used in the SET clause, they must always be preceded by a COLON character.
cursor_name	the name of the updatable cursor being used to point to the rows to be updated.

ANSI Compliance

The positioned form of UPDATE is ANSI SQL-99-compliant.

Authorization

You must have the UPDATE privilege on the table or columns to be updated.

When executing an UPDATE that also requires READ access to an object, you must have the SELECT right to data being accessed.

For example, in the following statement, READ access is required by the WHERE condition.

```
UPDATE table_1
SET field_1=1
WHERE field_1<0;
```

Similarly, the following statement requires READ access because you must read field_1 values from table_1 in order to compute the new values for field_1.

```
UPDATE table_1
SET field_1 = field_1 + 1;
```

An UPDATE operation sets a WRITE lock for the table or row being updated.

The activity count in the success response for an UPDATE statement reflects the total number of rows updated. If no rows qualify for update, then the activity count is zero.

UPDATE With Correlated Subqueries

See "Correlated Subqueries and the UPDATE Statement" on page 1-52 for information about using correlated subqueries with UPDATE statements.

ANSI Mode Rules

The following rules apply to the positioned form of UPDATE in ANSI mode:

- The WHERE CURRENT OF clause enables a UPDATE statement to act on a data row currently pointed to by the cursor named in WHERE CURRENT OF *cursor_name*. Such a cursor is said to be updatable.
- You need not include a specification of intent to update or delete a row when you declare *cursor_name*.
- Multiple updates of the currently fetched row of *cursor_name* or updates followed by a delete of the currently fetched row of *cursor_name* are valid.

Rule for Updating Partitioning Columns of a PPI Table

If you are updating a partitioning column for a partitioned primary index, then updates to the partitioning columns must be in a range that permits the partitioning expression to produce, after casting values to INTEGER if the value is not already of that type, a non-null value between 1 and 65 535.

Identity Columns and UPDATE

You cannot update a GENERATED ALWAYS identity column.

Update of GENERATED ALWAYS Identity Columns and PARTITION Columns Is Not Permitted

You cannot update the following set of system-generated columns:

- GENERATED ALWAYS identity columns
- PARTITION

You can update a GENERATED BY DEFAULT identity column. The specified value is not constrained by identity column parameters, but is constrained by any CHECK constraints defined on the column.

Example

In this example, the name of the cursor used to update the table is cursor_01.

```
EXEC SQL
 UPDATE table_1
 SET text = :text, K = :I + 1000
 WHERE CURRENT OF cursor_01;
```

UPDATE (Upsert Form)

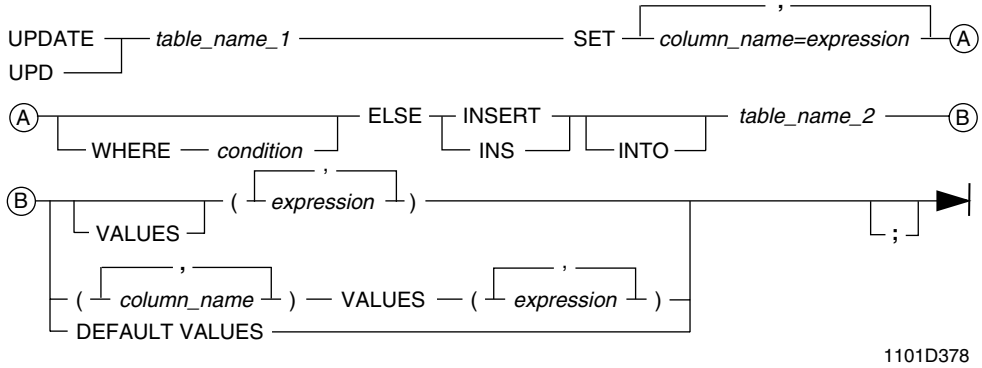
Purpose

Updates column values in a specified row and, if the row does not exist, inserts it into the table with a specified set of initial column values.

Invocation

Executable.

Syntax



where:

Syntax element ...	Specifies ...
table_name_1	the name of the table in which the row set is to be updated, or the name of the view through which the base table is accessed. This must match the specification for *table_name_2*.
column_name = *expression*	the value (*expression*) to be updated in column *column_name*. When host variables are used in the SET clause, they must always be preceded by a COLON character.

<table>
<tr><th>Syntax element …</th><th>Specifies …</th></tr>
<tr><td>condition</td><td>the predicate that specifies the row to be updated.
The specified predicate cannot be a subquery.
<table><tr><th>IF the table to be updated is this type …</th><th>THEN this many rows can be updated …</th></tr><tr><td>SET with a unique primary index</td><td>one.</td></tr><tr><td>• SET with a nonunique primary index
• MULTISET</td><td>multiple.</td></tr></table></td></tr>
<tr><td>table_name_2</td><td>the name of the table in which the row set is to be inserted, or the name of the view through which the base table is accessed.
This name must match the name specified for table_name_1.</td></tr>
<tr><td>expression</td><td>the non-default set of values to be inserted into table_name_2.
Note that you must specify values or nulls for each column defined for the table named table_name_2 or the insert fails.</td></tr>
<tr><td>column_name</td><td>the column set into which the values specified by expression are to be inserted.</td></tr>
<tr><td>DEFAULT VALUES</td><td>that a row consisting of default values is to be added to table_name.
If any column does not have a defined DEFAULT phrase, then the process inserts a null for that column.</td></tr>
</table>

ANSI Compliance

The upsert form of UPDATE is a Teradata extension to the ANSI SQL-99 standard.

Authorization

The following privilege rules apply to both the UPDATE and INSERT portions of this statement.

- You must have *both* update and insert privileges on the base table, view, or column set to be updated regardless of which portion of the statement is performed.

The following privilege rules applies to the UPDATE portion of this statement.

- You must have the UPDATE privilege on the base table, view, or column set to be updated.
- To update rows through a view, you must have the UPDATE privilege on the view.
- Also, the immediate owner of the view (that is, the database in which the view resides) must have the UPDATE privilege on the underlying object (view or base table).

The following privilege rules apply to the INSERT portion of this statement.

- You must have the INSERT privilege on the referenced table.
- To insert rows into a table through a view, you must have the INSERT privilege on the view.

  Also, the immediate owner of the view (that is, the database in which the view resides) must have the INSERT privilege on the underlying object (view or base table).

UPDATE (Upsert Form) and Subqueries

The Upsert form of UPDATE does not support subqueries.

Definition: Upsert

A *simple* upsert is a standalone update operation for which a subsequent standalone insert operation is coded that is performed only if the specific row to be updated is found not to exist. These actions require the performance of *two* individual SQL statements in the case where no row is found to update. This was the only means of performing a conditional insert on an update failure prior to the introduction of the upsert form of the UPDATE statement.

The more performant *atomic* upsert operation, represented by the upsert form of the UPDATE statement, 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.

Purpose of the Atomic Upsert Operation

The single-pass upsert is described as atomic to emphasize that its component UPDATE and INSERT SQL statements are grouped together and performed as a *single*, or atomic, SQL statement.

Existing TPump scripts that use the upsert feature in its TPump-specific form, using paired UPDATE and INSERT statements, do not have to be changed to take advantage of the atomic upsert capability because TPump automatically converts an UPDATE … INSERT statement pair in its older syntax into a corresponding atomic upsert statement whenever appropriate.

The atomic upsert syntax can also be used by any CLIv2-based applications and is particularly useful when coded directly into BEQ scripts.

Constraints on Using Atomic Upsert

There are three fundamental constraints on an atomic upsert operation that apply universally. The following table defines and explains these constraints.

Constraint	Explanation
The UPDATE and INSERT components of the upsert operation must specify the same table.	The purpose of an atomic upsert is to first attempt to update a particular row in a table and then, if that row is not found, to insert it into the table. By specifying different tables for the UPDATE and INSERT components of the statement, the intent of the operation is violated.
The UPDATE and INSERT components of the upsert operation must specify the same row. In other words, the primary index value for the inserted row is identical to the primary index value for the targeted update row.	If an upsert operation cannot update the specific row it targets, then it inserts the row it would have otherwise updated, inserting the specific "update" values into the fields that would have been updated had the row existed in the table. This constraint is met when the primary index value specified by the WHERE clause of the UPDATE component matches the primary index value implied by the column values specified in the INSERT component. Because the value of the number generated for an INSERT into an identity column is not knowable in advance, you cannot perform an upsert operation into a target table that has an identity column as its primary index. Upserts into identity column tables for which the identity column is not the primary index are valid.
The UPDATE component fully specifies the primary index value to ensure that all accesses are one-AMP operations.	When the UPDATE component of the statement fully specifies the primary index, Teradata accesses any targeted row with a single-AMP hashed operation. This rule is applied narrowly in the TPump case, where it is taken to mean that the primary index is specified in the WHERE clause of the UPDATE component as equality constraints using simple constraint values (either constants or USING clause references to imported data fields). Simple values are also assumed for constraints on non-index columns and for the update values in the SET clause, avoiding any costly subqueries or FROM clause references. Similarly, TPump has a performance-driven preference to avoid subqueries in the INSERT.

When you perform an upsert UPDATE to a PPI table, the following rules must be followed or an error is returned:

- The values of the partitioning columns must be specified in the WHERE clause of the UPDATE clause of the statement.
- The INSERT clause of the statement must specify the same partition as the UPDATE clause.
- The UPDATE clause must not modify a partitioning column.

The outcome of expression evaluation errors of PPI tables, such as divide-by-zero errors, depends on the session mode.

In this session mode ...	Expression evaluation errors roll back this work unit ...
ANSI	request.
Teradata	transaction.

UPDATE (Upsert Form) as a Triggering Action

If no rows qualify, then no triggers are fired.

Triggers invoke the following behavior when fired as the result of an UPDATE (Upsert Form) statement:

IF a trigger is defined on this type of action ...	THEN it is fired ...
UPDATE	in all cases. No insert triggers are performed.
INSERT	only if no row is updated and the INSERT is performed.

Triggers are fired as if the UPDATE and INSERT components in the atomic upsert statement are performed as separate statements: unconditionally for UPDATE and conditionally[3] for INSERT.

UPDATE (Upsert Form) as a Triggered Action

UPDATE (Upsert Form) statements are supported as triggered actions. The rules for their use as triggered actions are the same as for any other SQL statement.

3 The INSERT is performed only when no rows qualify for the UPDATE.

Restrictions

The basic upsert constraints described in "Constraints on Using Atomic Upsert" on page 3-163 follow from the conventional definition of upsert functionality. Additional restrictions are stated in the following table in terms of unsupported syntax or features. Any such restricted request returns an error message if submitted to Teradata.

Unsupported Syntax	Explanation
`INSERT … SELECT`	
Positioned `UPDATE`	A WHERE clause cannot use an updatable cursor to do a positioned UPDATE.
`UPDATE … FROM`	
`UPDATE … WHERE subquery`	A WHERE clause cannot use a subquery either to specify the primary index or to constrain a non-index column.
`UPDATE … SET … primary_index_column`	You cannot update a primary index column because any change to the row hash value for a row affects its distribution.
`UPDATE … SET … partitioning_column`	You cannot update a partitioning column because any change to the partitioning for a row affects its location.
`UPDATE … SET identity_column …`	You cannot update GENERATED ALWAYS identity column values.
`UPDATE … SET PARTITION …`	You cannot update PARTITION values.

Examples

This section describes several examples that demonstrate how the atomic upsert form of UPDATE works, including error cases. All examples use the same table called Sales, which is defined as indicated.

```
CREATE TABLE Sales, FALLBACK,
 (ItemNbr   INTEGER NOT NULL,
  SaleDate  DATE FORMAT 'MM/DD/YYYY' NOT NULL,
  ItemCount INTEGER)
PRIMARY INDEX (ItemNbr);
```

Assume that the table has been populated with the following data.

```
INSERT INTO Sales (10, '05/30/2000', 1);
```

Example 1: Upsert Update

This example demonstrates a valid upsert UPDATE statement.

```
UPDATE Sales
SET ItemCount = ItemCount + 1
WHERE (ItemNbr = 10 AND SaleDate = '05/30/2000')
ELSE INSERT
INTO Sales (10, '05/30/2000', 1);
```

After all of the rules have been validated, the row with ItemNbr = 10 and SaleDate = '05/30/2000' gets updated.

A message noting the successful update of one row returns to the user.

Example 2: Upsert Insert

This example demonstrates a valid upsert INSERT statement.

```
UPDATE Sales
SET ItemCount = ItemCount + 1
WHERE (ItemNbr = 20
AND SaleDate = '05/30/2000')
ELSE INSERT INTO Sales (20, '05/30/2000', 1);
```

After all of the rules have been validated and no row was found that satisfies the compound predicate Item = 20 and SaleDate = '05/30/2000' for the update, a new row is inserted with ItemNbr = 20.

A message noting the successful insert of one row returns to the user.

Example 3:

This example demonstrates an upsert UPDATE statement that does not specify the same table name for both the UPDATE part and the INSERT part of the statement.

```
UPDATE Sales
SET ItemCount = ItemCount + 1
WHERE (ItemNbr = 10
AND SaleDate = '05/30/2000')
ELSE INSERT INTO NewSales (10, '05/30/2000', 1);
```

One of the rules of the upsert form of UPDATE is that only one table is processed for the statement. Because the tables Sales and NewSales are not the same for this example Teradata returns an error to the user indicating that the name of the table must be the same for both the UPDATE and the INSERT.

Example 4:

This example demonstrates an upsert UPDATE statement that does not specify the same primary index value for the UPDATE and INSERT parts of the statement.

```
UPDATE Sales
SET ItemCount = Itemcount + 1
WHERE (ItemNbr = 10
AND SaleDate = '05/30/2000')
ELSE INSERT INTO Sales (20, '05/30/2000', 1);
```

The primary index value specified for the UPDATE and the INSERT must be the same. Otherwise, the operation is looking at two different rows: one for UPDATE and the other for the INSERT. This is not the purpose of the upsert form of UPDATE.

Because the specified primary index values of 10 and 20 are not the same, this case returns an error to the user, indicating that the primary index value must be the same for both the UPDATE and the INSERT.

Example 5:

This example demonstrates an upsert UPDATE statement that does not specify the primary index in its WHERE clause.

```
UPDATE Sales
SET ItemCount = ItemCount + 1
WHERE SaleDate = '05/30/2000'
ELSE INSERT INTO Sales (10, '05/30/2000', 1);
```

When the primary index is not specified in the UPDATE portion of an upsert statement, the operation could have to perform an all-row scan to find rows to update. This is not the purpose of upsert form of UPDATE. This case returns an error.

Example 6:

This example demonstrates an upsert UPDATE statement that fails to specify the ELSE keyword.

```
UPDATE Sales
SET ItemCount = ItemCount + 1
WHERE (ItemNbr = 10
AND SaleDate = '05/30/2000')
INSERT INTO Sales (10, '05/30/2000', 1);
```

This case returns an appropriate syntax error to the user.

USING Row Descriptor

Purpose

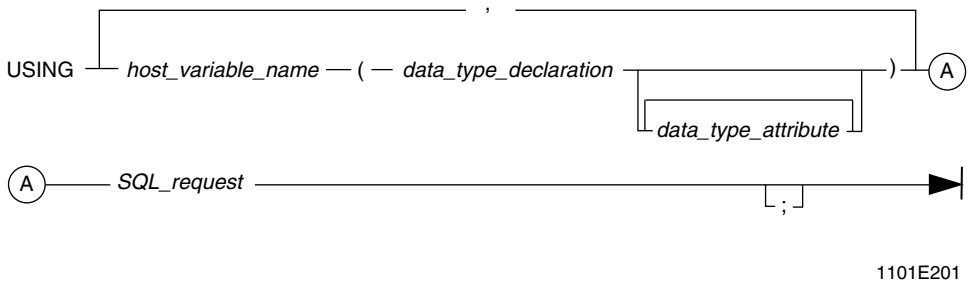
Defines one or more variable parameter names.

Invocation

Executable.

Interactive SQL only.

Syntax



1101E201

where:

Syntax Element . . .	Specifies . . .
host_variable_name	the name of a client system variable that is to be referenced as a parameter in the SQL request associated with the modifier. Each name specified must be unique. During processing, *host_variable_name* is replaced by a constant value stored in the specified host variable.
data_type_declaration	the data type of the constant value substituted for *name*. This can be used to set a default value. See the chapter "SQL Data Definition" in *Teradata RDBMS SQL Reference, Volume 3* for a list of data types.

Syntax Element . . .	Specifies . . .
data_type_attribute	one of the data type attributes listed below. The only data type attributes that affect the handling of values imported by USING are the following: • CASESPECIFIC • NOT CASESPECIFIC • UPPERCASE The character data type attribute (for example CHARACTER SET LATIN) *cannot* be part of *data_type_attribute*. Type attributes are Teradata extensions to the ANSI SQL-99 standard.
SQL_request	the SQL request with which the USING modifier is associated. This can be a multistatement request, any non-DDL single-statement request, or an explicit transaction. All host variable names must be preceded by a COLON character.

ANSI Compliance

USING is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

USING Not Supported for Embedded SQL

The Teradata preprocessor does not support the USING row descriptor.

Instead, declare and use input host variables in your embedded SQL applications where you would use the USING row descriptor interactively.

How USING Works

USING works as described in the following table.

Stage	Process
1	A value is substituted for each parameter name when the modified request is processed.
2	Each value is retrieved as a data record from a client system-resident source such as a disk file.
3	The data records are passed to Teradata as message parcels along with the text of the request.

USING and DateTime System Functions

In non-ANSI Teradata applications, when you access the system functions DATE or TIME, the values are obtained directly from the operating system and placed in a USING row for access by the query being performed. In this situation, the value for DATE is stored with type DATE and the value for TIME is stored with type REAL. These values are stored in known fixed fields in the USING row.

The ANSI system functions CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP behave differently.

CURRENT_DATE is stored in the field previously used exclusively for DATE.

CURRENT_TIME is *not* stored in the field previously used for TIME because both its storage format and its content are different and are not interchangeable with those of TIME. Instead, another system-value field is added to the USING row. Notice that CURRENT_TIME includes TIME ZONE data in addition to the more simple TIME data, so the differences between the two extend beyond storage format.

CURRENT_TIMESTAMP is also stored as an additional separate field in the USING row.

With the addition of CURRENT_TIME and CURRENT_TIMESTAMP, both the existing Teradata DATE and TIME functionality and the ANSI DateTime functionality are supported for the USING row descriptor.

Host Variables

The maximum number of host variables you can specify in a USING row descriptor is 2 550.

Each host variable name must be unique in a USING clause.

The host variable construct (:*name*) is allowed anywhere a character, numeric, or byte constant is allowed. The constant values in the client system data record must match, item for item, the USING description of those values. Also, the constants in the data record must appear in the order in which they are defined in the USING clause.

One USING clause can only modify one request. If several requests are to use client system variables, each request must be associated with its own USING clause (see "Examples" on page 3-175).

Valid Query Types

A USING modifier can be associated with one of the following types of queries:

- Any single-statement request except a DDL statement
- A multistatement request
- An explicit transaction. If the modifier is associated with an explicit transaction, the USING keyword must appear as the first keyword in that transaction.

If the first statement or request references a USING variable, the USING clause should immediately precede the BEGIN TRANSACTION statement (see Examples).

ANSI DateTime Considerations

Other than DATE values in INTEGERDATE mode, external values for DateTime and Interval data are expressed as fixed length CharFix character strings (in logical characters) in the designated client character set for the session.

The TYPE provided in a USING phrase for any client data to be used as a DateTime value can be defined either as the appropriate DateTime TYPE or as CHAR(n), where n is a character string having the correct length for the external form of a DateTime or Interval data type, and the character string is to be used internally to represent a DateTime or Interval value.

When a client creates USING phrases without being aware of the ANSI DateTime data types, those fields are TYPEd as CHAR(n). Then when those USING values appear in the assignment lists for INSERTs or UPDATEs, the field names from the USING phrase can be used directly.

An example follows.

```
USING (TimeVal CHARACTER(11), NumVal INTEGER, TextVal(CHARACTER(5))
INSERT INTO TABLE_1 (:TimeVal, :NumVal, :TextVal);
```

When you import ANSI DateTime values with a USING phrase and the values are to be used for actions other than an INSERT or UPDATE, you must explicitly CAST them from the external character format to the proper ANSI DateTime TYPE.

An example follows.

```
USING (TimeVal CHARACTER(11), NumVal INTEGER)
UPDATE TABLE_1
SET TimeField=:TimeVal, NumField=:NumVal
WHERE CAST(:TimeVal AS TIME(2)) > TimeField;
```

While you can use TimeVal CHAR(11) directly for assignment in this USING phrase, you must CAST the column data definition explicitly as TIME(2) in order to compare the field value TimeField in the table because TimeField is an ANSI TIME defined as TIME(2).

You can use both DateTime and Interval declarations to allow a USING phrase to directly indicate that an external character string value is to be treated as a DateTime or Interval value. To import such values, you import their character strings directly into the USING phrase.

If you move values into a USING phrase and the character string data cannot be converted into valid internal DateTime or Interval values as indicated by their TYPE definitions, then an error is returned to the application.

Example of ANSI DateTime and Interval With USING

The following USING phrase expects to see an 11 character field containing a string such as ‘21:12:57.24’.

```
USING ( TimeVal TIME(2), … )
```

You could import the same value using the following USING phrase; however, it is unclear that the data is a time value with two decimal places:

```
USING ( TimeVal CHARACTER(11), … )
```

ANSI DateTime and Parameterized Queries

A DataInfo parcel is used in both “Prepare then Execute” and “Parameterized Query” CLI modes to describe fields for export and import and to build a USING data descriptor without a USING phrase in the request text.

In this mode of operation, Teradata observes the following rules.

- DateTime and Interval exported values as seen by the client as CharFix data with the exception of DATE values when the session is in INTEGERDATE mode.
- Imported values that are to be DateTime or Interval values are seen by Teradata as CharFix values with the exception of DATE values when the session is in INTEGERDATE mode.

  When used in an INSERT or UPDATE statement, CharFix values can be implicitly cast in the appropriate DateTime or Interval value. In this case, Teradata ensures that the value being assigned is CharFix and that it has the right length to represent the DateTime or Interval value to which it is being assigned.

  Teradata CASTs the value and accepts the result if the contents are a valid representation of a value of the indicated data type. In other cases, you must explicitly CAST the USING parameter as the appropriate DateTime or Interval data type.

Character String Definitions in a USING Row Descriptor

Default case specificity for character string comparisons depends on the mode defined for the current session.

IF the session is in this mode ...	THEN the values default to this declaration for string comparisons ...
ANSI	CASESPECIFIC
Teradata	NOT CASESPECIFIC

You can add the explicit designation NOT CASESPECIFIC to the definition of a CHARACTER or VARCHAR field in the USING phrase to override the default.

The purpose of the NOT CASESPECIFIC attribute is to ease the transition to an ANSI mode of operation. Instead of using this transitional feature, you should instead use the ANSI-compliant UPPER function to perform case blind comparisons.

data_type_declaration Considerations for a Japanese Character Site

Be very careful with Japanese character data regarding the information that will be sent in the data parcel in client form-of-use. Consider the following example for illustrating the issues involved.

```
USING (emp_name CHARACTER(40) UPPERCASE, emp_number INTEGER)
INSERT INTO employee (:emp_name, :emp_number);
```

This SQL specifies that the data parcel contains 40 bytes of CHARACTER data for emp_name and 4 bytes of INTEGER data for emp_number.

Because the *data_type_declaration* describes the layout of the data parcel in terms of client form-of-use, this means that CHARACTER(40) indicates 40 *bytes* of CHARACTER information rather than 40 *characters*.

For single byte character external data sets such as ASCII and EBCDIC, bytes and characters represent the same byte count.

For character sets containing mixed single byte and multibyte characters (such as KanjiEUC) or only 2-byte characters, however, the numbers of bytes and numbers of characters are *not* identical.

Notice that the GRAPHIC and VARGRAPHIC data types have character counts that are always half the number of bytes.

For Kanji character sets containing only GRAPHIC multibyte characters, such as KanjiEBCDIC, character and byte units are identical.

You can specify graphic USING data using the [VAR]CHAR(n) CHARACTER SET GRAPHIC clause for the *data_type_declaration*. This is identical to specifying the *data_type_declaration* as [VAR]GRAPHIC(n).

There is no equivalent extension to CHARACTER SET syntax to account for CHARACTER data because the data is in client rather than internal form-of-use.

Data Limitations for INSERT

The following table describes the data limits on INSERTs for various CHARACTER and GRAPHIC character data types.

Character Set Type	USING Data Type	Maximum Size for USING Data	Character Data Type	Maximum Field Size for INSERT (Characters)
Single byte	CHARACTER	64K bytes	Latin	64K
			Kanji1	
			Unicode[a]	32K
			KanjiSJIS[a]	
Multibyte	CHARACTER	64K bytes	Latin[b]	32K
			Kanji1	64K
			Unicode[a]	32K
			KanjiSJIS[a]	32K
	GRAPHIC	32K characters	GRAPHIC	32 K
			Latin[b]	
			Kanji1	
			Unicode[a]	
			KanjiSJIS[a]	

a. If the USING data contains more than 32K logical characters (or more than 64K *bytes* when mixed single byte-multibyte strings are involved), the characters that exceed that limit are truncated.
b. If the USING data contains more than 32K logical characters (or more than 64K *bytes* when mixed single byte-multibyte strings are involved), the characters that exceed that limit are truncated.

While a Latin field can be defined as 64K, you cannot insert more than 32K multibyte characters because Teradata uses Unicode internally, and Unicode has a 32K limit.

The Load utilities are similarly limited.

Use Consistent CHARACTER Declarations

CHARACTER(*n*) and VARCHAR(*n*) definitions for the UNICODE character data set in a USING clause are defined in terms of bytes, where *n* designates 2*n* bytes.

In all other SQL declarations, such as a column definition within a CREATE TABLE statement, UNICODE CHARACTER(*n*) and VARCHAR(*n*) expressions are defined in terms of *n characters*.

Character String Assignment and GRAPHIC Columns

The following table describes the behavior of character strings assigned to GRAPHIC and non-GRAPHIC columns.

Under no circumstances are you permitted to import non-GRAPHIC data into character columns typed as GRAPHIC nor are you permitted to import GRAPHIC data into character columns not typed as GRAPHIC.

IF USING describes a character string for assignment to a column of this type . . .	THEN it must describe the USING string as this type . . .
Graphic	GRAPHIC(n) VARGRAPHIC(n) CHARACTER(n) CHARACTER SET GRAPHIC VARCHAR(n) CHARACTER SET GRAPHIC
Latin Unicode KanjiSJIS Kanji1	CHARACTER(n) VARCHAR(n)

UPPERCASE Option and Character Parameter Definition in USING

When you specify UPPERCASE for a character parameter in a USING clause, it applies only to the single byte characters of any mixed single byte/multibyte character set.

To ensure uniform application of uppercasing, do one of the following things:

- Define the column as UPPERCASE in CREATE/MODIFY TABLE
- Use the UPPER function to convert the lowercase characters to uppercase.

See "Example 4: Kanji Usage" on page 3-178.

Examples

The following examples illustrate the use of USING.

Example 1

In this example the USING modifier defines the client system variables :emp_name and :emp_number as, respectively, a character constant and a numeric constant. The variables are replaced by values from a client system data record when the accompanying INSERT statement is processed.

```
USING (emp_name CHARACTER(40), emp_number INTEGER)
INSERT INTO Employee (Name, EmpNo)
```

```
VALUES (:emp_name, :emp_number) ;
```

The INSERT statement (in record mode on an IBM Series/370) would be transmitted to Teradata with an appended 44-byte data record consisting of a 40-byte EBCDIC character string, followed by a 32-bit integer.

Example 2

Here, the USING modifier establishes three variables whose constant values are used both for data input and as WHERE conditionals in a multistatement request:

```
USING (var_1 CHARACTER, var_2 CHARACTER, var_3 CHARACTER)
   INSERT INTO TestTabU (c1) VALUES (:var_1)
;  INSERT INTO TestTabU (c1) VALUES (:var_2)
;  INSERT INTO TestTabU (c1) VALUES (:var_3)
;  UPDATE TestTabU
   SET c2 = c1 + 1
   WHERE c1 = :var_1
;  UPDATE TestTabU
   SET c2 = c1 + 1
   WHERE c1 = :var_2
;  UPDATE TestTabU
   SET c2 = c1 + 1
   WHERE c1 = :var_3 ;
```

Example 3

In this example, the USING modifier defines a variable for use with an explicit transaction that reads character strings from a disk file and inserts them in signed zoned decimal format.

The USING modifier precedes the BEGIN TRANSACTION statement, while the BEGIN TRANSACTION statement and the statement associated with the USING clause are entered as one multistatement request.

```
USING (zonedec CHARACTER(4))
BEGIN TRANSACTION
; INSERT INTO Dectest (Colz = :zonedec (DECIMAL(4), FORMAT
   '9999S')) ;

USING (zonedec CHARACTER(4))
INSERT INTO Dectest
   (Colz = :zonedec (DECIMAL(4), FORMAT '9999S')) ;

USING (zonedec CHARACTER(4))
INSERT INTO Dectest
   (Colz = :zonedec (DECIMAL(4), FORMAT '9999S')) ;
ET;
```

In BTEQ applications, you can combine USING with the .REPEAT command to perform multiple insertions automatically.

Example 4: Kanji Usage

This example is based on the following table definition and subsequent INSERT statement against that table.

```
CREATE TABLE table_1
(cunicode  CHAR(10) CHARACTER SET UNICODE
,clatin    CHAR(10) CHARACTER SET LATIN
,csjis     CHAR(10) CHARACTER SET KANJISJIS
,cgraphic  CHAR(10) CHARACTER SET GRAPHIC,
,cgraphic2 CHAR(10) CHARACTER SET GRAPHIC);

USING
cunicode(CHAR(10))
,clatin(CHAR(10))
,csjis(CHAR(10))
,cgraphic(GRAPHIC(10))
,cgraphic2(CHAR(10) CHARACTER SET GRAPHIC))
INSERT INTO table_1(:cunicode, :clatin, :csjis, :cgraphic,
 :cgraphic2);
```

Suppose the session character set is KanjiShift-JIS.

The USING clause indicates that the data parcel contains 10 KanjiShift-JIS *bytes* for the UNICODE field, 10 for LATIN, 10 for KANJISJIS, 20 for the first GRAPHIC field, and 20 for the second GRAPHIC field.

Individual fields in the data parcel are converted from client form-of-use to the Teradata server form-of-use.

After conversion the first three fields are treated as UNICODE literals and the last two fields are treated as GRAPHIC literals (see “Character Data Literals” in the chapter “Character Data Types” in *Teradata RDBMS SQL Reference, Volume 3*).

The column data is then converted to the target fields using implicit translation according to the rules listed under “Character Data Type Assignability Rules” in the chapter “Data Type Conversions” in *Teradata RDBMS SQL Reference, Volume 5*.

The conversion process is described by the following table.

Stage	Process			
	These bytes are converted to server form-of-use ...	**Treated as this kind of literal ...**	**Converted to this character data type ...**	**And stored in this column ...**
1	first 10 (1 - 10)	UNICODE	none	cunicode
2	next 10 (11 - 20)	UNICODE	LATIN	clatin
3	next 10 (21-30)	UNICODE	KANJISJIS	csjis
4	next 20 (31 - 50)	GRAPHIC	GRAPHIC	cgraphic
5	last 20 (51 - 70)	GRAPHIC	GRAPHIC	cgraphic2

Note that the meaning of the USING clause is independent of the session character set.

WHERE Clause

Selects rows in a SELECT query that satisfy a conditional expression.

See "WHERE Clause" on page 1-32 for a detailed discussion of the WHERE clause.

WITH Clause

Specifies summary lines and breaks (grouping conditions) that determine how selected results from a SELECT query are returned (typically used for subtotals).

See "WITH Clause" on page 1-85 for detailed information on the WITH clause.

Section 2:
Embedded SQL

Section Contents

Chapter 4:

Embedded SQL

This chapter describes special topics and SQL statements whose use is restricted to embedded SQL applications.

The following list names the topics described in the chapter:

- "Host Structures" on page 4-5
- "Host Variables" on page 4-7
- "Input Host Variables" on page 4-12
- "Output Host Variables" on page 4-15
- "SQL Character Strings as Host Variables" on page 4-19
- "Indicator Variables" on page 4-20
- "Multistatement Requests With Embedded SQL" on page 4-24
- "Miscellaneous Embedded SQL-Only Statement Syntax" on page 4-28
- "Dynamic SQL" on page 4-52
- "Dynamic SQL Statement Syntax" on page 4-54

You can find information about other embedded SQL-related topics in the following chapters:

- Chapter 5: "Client-Server Connectivity Statements"
- Chapter 6: "Multisession Asynchronous Programming With Embedded SQL"
- Chapter 7: "Cursors and Cursor Control Statements"
- Chapter 8: "Result Code Variables"

Embedded SQL

Definition

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

An embedded Teradata SQL client program consists of the following:

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

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

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

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

Special SQL Statements for Embedded SQL

Many SQL language constructs are required for embedded SQL that are not supported for interactive use of the language.

By contrast, with few exceptions any SQL statement that can be performed interactively can be used in an embedded SQL application. The exceptions are the non-ANSI Teradata extensions ECHO and USING.

Embedded SQL includes the following SQL components:

- Direct, or interactive, SQL
- Extensions providing host variable support
- Statements supporting the following constructs to support embedded SQL:
  - Declaratives
  - Dynamic SQL
  - Cursors

Supported Host Languages

Teradata supports embedded SQL for the following host application languages only:

- C
- COBOL
- PL/I

Embedded SQL Statements Must Be Preprocessed

Because client programming languages do not understand SQL, you must preprocess SQL-containing source code using a precompiler or preprocessor (the Preprocessor2 product is the Teradata precompiler and runtime SQL statement manager) that first comments out and then converts the SQL language elements into CLIv2 calls before compiling them using the appropriate C, COBOL, or PL/I compiler.

Consult *Teradata Preprocessor2 Programmer Guide* for examples of embedded SQL applications in the supported client languages.

Data Returning Statements

A data returning statement is an embedded SQL statement that returns one or more rows of data to the program.

The data returning SQL statements are as follows:

- CHECKPOINT
- COMMENT (Comment Returning Form)
- EXPLAIN
- HELP
- SELECT
- SHOW

Each data returning statement must specify the host output variables into which the returned data is placed.

IF . . .	**THEN** . . .
no more than one row of data can be returned	you can use an INTO clause with the statement to specify the host variables.
more than one row is expected	use the selection cursor method (see Chapter 7: "Cursors and Cursor Control Statements") and do not specify the INTO clause.
a data returning statement is performed dynamically	you *must* define a dynamic cursor, regardless of the number of response rows expected.

Rules and Characteristics of Embedded SQL

The rules and characteristics for embedding SQL within client language application programs are listed in the following bullets.

- All embedded SQL statements must be prefixed by EXEC SQL.
- All embedded SQL statements must be terminated. The terminator depends on the client application language.

FOR this language . . .	The SQL terminator is . . .
COBOL	END-EXEC
C	;
PL/I	

- Any *executable* SQL statement can appear anywhere that an executable client application language statement can appear.
- Embedded SQL statements can reference host variables.
- A host variable must be defined between BEGIN DECLARE SECTION and END DECLARE SECTION statements.
- A host variable must be defined prior to any SQL statement reference to it.
- All host variables should be drawn from the same domain as their target columns.
- Teradata host variables and columns can have the same names.
- All embedded SQL programs must contain one or both of the following host variables for communicating status between the Teradata database and the client application:
  - SQLSTATE (see "SQLSTATE" on page 8-2)
  - SQLCODE (see "SQLCODE" on page 8-6)

  The ANSI SQL-92 standard deprecates the use of SQLCODE and the ANSI SQL-99 standard no longer supports it; therefore, you should use the SQLSTATE variable for any applications that you intend to run under ANSI mode.
- You should always test the value for SQLCODE or SQLSTATE (or both) after performing an executable embedded SQL statement.

  See "WHENEVER" on page 4-48 for more information about testing the values of SQLCODE and SQLSTATE.

Host Structures

Definition

A host structure is an array of host variables that is declared outside of SQL in the host language of your embedded SQL application.

Example

Consider the following embedded SQL SELECT statement written for a COBOL application. The purpose of this statement is to produce a report on the ethnic demographics of the first 100 employees hired by the corporation.

```
EXEC SQL
 SELECT EmpNo, LastName, Ethnicity, BirthDate, SSN, DeptNo
 INTO :EmpNo, :LastName, :Ethnicity, :BirthDate,:SSN,:DeptNo
 FROM Employee
 WHERE EmpNo < '100'
END-EXEC
```

Rather than typing the names of the six host variables, you can create a named host structure that contains :EmpNo, :LastName, :Ethnicity, :BirthDate, :SSN, and :DeptNo as individual elements within the array and then substitute that name in the query for the individual host variables.

The same COBOL example could then be rewritten as follows, where :FounderEmployeeInfo is the name of the host structure that contains the host variables :EmpNo, :LastName, :Ethnicity, :BirthDate, :SSN, and :DeptNo.

```
EXEC SQL
 SELECT EmpNo, LastName, Ethnicity, BirthDate, SSN, DeptNo
 INTO :FounderEmployeeInfo
 FROM Employee
 WHERE EmpNo < '100'
END-EXEC
```

Host Structures Are Not Supported In ANSI Mode

ANSI mode does not support arrays; therefore, it does not support host structures nor does it support qualified host variables.

Host Structures Are Supported for Teradata Mode

Teradata mode supports IBM-style host structures to a maximum of two levels.

Teradata mode also supports qualified host variables to reference fields within host structures.

Fully qualified host variable references in embedded SQL statements are expressed in the same way as fully qualified SQL column references: with a FULLSTOP character separating the different levels of qualification.

This syntax is valid for all supported host languages.

This example uses COBOL to illustrate the point, using :SHIPMENT-RECORD.WEIGHT as a fully qualified host variable.

```
ADD 5 TO WEIGHT OF SHIPMENT-RECORD.
EXEC SQL
 DELETE FROM SHIPMENT_TABLE
 WHERE WEIGHT > :SHIPMENT-RECORD.WEIGHT
END-EXEC.
```

Host Variables

Definition

A host variable is one of the following items that is referenced in an embedded SQL statement:

- A host language variable that is defined directly with statements in that host language.
- A host language SQL-based construct that is generated by the SQL preprocessor and indirectly defined from within SQL.

In stored procedures, local variables and parameters perform the same function. See "Local Variables" on page 9-9 and "Parameters" on page 9-9.

Purpose of Host Variables

Host variables provide input values to SQL statements (see "Input Host Variables" on page 4-12) or receive output values from SQL requests (see "Output Host Variables" on page 4-15). They are identified by name in an embedded SQL statement (for example, Value-Var or HostIn-Var).

A host variable within an embedded SQL statement has a 1:1 relationship with a host variable of the same name declared in the host language of an application between the SQL BEGIN DECLARE SECTION and END DECLARE SECTION statements.

Classification of Host Variables

Host variables are classified into main and indicator categories.

A host . . .	Is a host variable . . .
main variable	used to send data to or receive data from the Teradata database.
indicator variable	that indicates any of the following: • A main variable is null on input • A main variable is null on output • For character or byte data, truncation on output

Host Variable Processing

At runtime, the preprocessor extracts values from the specified host input variables and sends them to the Teradata database, along with the SQL statement to be processed. The functionality is similar to the Teradata interactive SQL USING clause with associated data, or to the Teradata SQL EXEC statement for a macro with parameters.

When Teradata returns the results of an SQL data returning statement to the client application program, the preprocessor places the corresponding values into the specified host output variables, which are listed in an INTO clause, separated by commas.

A host main variable can also be associated with a host indicator variable. See "Indicator Variables" on page 4-20 for more information on host indicator variables.

Rules for Using Host Variables

A number of rules apply to host variable usage. Several of these rules are independent of the embedded SQL statement in which a host variable is used.

These statement independent rules are noted in the following list of rules.

- All host variables must be preceded by a COLON character.
- Pad characters before or after the COLON character are optional. COLON character usage with host variables is discussed with the individual statements in Chapter 3: "SQL Data Manipulation Language Statement Syntax.".
- Host variable names must not begin with a numeric.
- Host variable names should be unique within a program. This is mandatory for applications written in C and strongly recommended for applications written in COBOL and PL/I.
- In a WHERE clause, use a host variable to specify the value in a search condition or to replace a literal in the search expression.

  Indicator variables (see "Indicator Variables" on page 4-20) are allowed in a WHERE clause.

  The COLON character preceding the host variable is mandatory.

  Specify the COLON character to distinguish the use of a variable from a table column reference.

  ```
  SELECT * FROM table
  INTO :intofield1
  WHERE COL1 = :hostvar1
  ```

  When executed, the value of hostvar1 is substituted into the WHERE clause as though a constant were specified.

- You can use a host variable in a SELECT list either as part of an expression or by itself to represent a single column.

  Indicator variables (see "Indicator Variables" on page 4-20) are not allowed in the SELECT list.

  The COLON character preceding the host variable is mandatory to distinguish the host variable from a table column reference.

  ```
  SELECT :hostvar1,
  COL1, COL2 + :hostvar2
  INTO :intofield1,
  :intofield2 INDICATOR :indvar1,
  :intofield3 INDICATOR :indvar2
  FROM table
  WHERE COL3 = 'ABC'
  ```

  Upon execution, the values in hostvar1 and hostvar2 are substituted into the SELECT list as though a constant had been specified.
- You can use host and indicator variables (see "Indicator Variables" on page 4-20) in the VALUES clause of the INSERT statement or in the SET clause of the UPDATE statement.
- You can use host variables in CHECKPOINT, DATABASE and LOGON statements to complete the command (that is, as SQL strings).
- You can use host variables to identify the application storage areas to receive the output of data returning statements.

  In Teradata mode, the COLON character preceding the main host variables (intofield1 and intofield2 in the example) is optional, but its use is very strongly recommended.

  The COLON character is also mandatory before an indicator host variable (indvar1 and indvar2 in the example) specification.

  This usage is always associated with the INTO clause of data returning statements or the cursor-related FETCH statement.

  ```
  SELECT column_1,
  column_2
  INTO :intofield1 INDICATOR :indvar1,
  :intofield2 INDICATOR :indvar2
  FROM table
  WHERE column_3 = 'ABC'
  ```

  :intofield*n* contains the value of column_*n* when the statement is performed. If column_*n* is null, then the value of :intofield*n* is indeterminate.

  :indvar*n* indicates whether the associated :intofield*n* *is null*, for character and byte string data, whether any truncation occurred in returning the Teradata server data to the field.

  For additional information on handling nulls in an application, see "Indicator Variables" on page 4-20.

COLON Character Usage With Host Variables

The ANSI SQL standard mandates that all host variables be preceded by a COLON character to distinguish them from SQL column references that might otherwise be ambiguous.

Teradata SQL requires a preceding COLON character in some situations, but not all (see "Mandatory COLON Character Usage in Teradata Mode" on page 4-10 and "Optional COLON Character Usage in Teradata Mode" on page 4-11 for details).

The best practice is to precede *all* host variables with a COLON character, even when your session is running in Teradata transaction mode.

Mandatory COLON Character Usage in Teradata Mode

Host variable references in an SQL statement *must* be preceded by a COLON character under the following conditions in Teradata mode:

- The host variable name is an SQL reserved word.
- The host variable is used as an indicator variable (see "Indicator Variables" on page 4-20).
- The syntax usage is ambiguous such that the name could be either a column reference or a host variable reference.

  For example, in a WHERE clause, WHERE column_1 = field_1, field_1 either could be a column in a table or a host variable.
- The reference is in a DATABASE statement; that is, DATABASE :var1.
- The reference is the object of a SET CHARSET statement.
- The reference is an argument of a Teradata function; for example, ABS(:var_1).
- A COBOL variable name intended for use as an input variable begins with a numeric character (0-9) where a numeric constant could be expected.
- The reference occurs in a place other than in one of the items in the list under "Optional COLON Character Usage in Teradata Mode" on page 4-11.
- A preceding COLON character is mandatory for all host variables specified in the SET or WHERE clauses of an UPDATE statement and for all host variables specified in a *match_condition*, SET, or INSERT clause in a MERGE statement.

Optional COLON Character Usage in Teradata Mode

Host variable references in an SQL statement are optionally preceded by a COLON character when the reference is one of the following in Teradata mode:

- In an INTO clause.
- Either the id or password variable in a CONNECT statement.
- In the FOR STATEMENT clause of a DESCRIBE or PREPARE statement.
- In the USING clause of an EXECUTE or OPEN statement.
- In the DESCRIPTOR clause of an EXECUTE, FETCH or OPEN statement.
- The object of a LOGON statement.
- The object of an EXECUTE IMMEDIATE statement.
- In the VALUES clause of an INSERT statement.
- In the TO STATEMENT clause of a POSITION statement.

The best practice is to precede *all* host variables with a COLON character, even when your session is in Teradata transaction mode.

Host Variables in Dynamic SQL

Dynamic SQL does not support host variables in the same way they are supported in static SQL. Instead, dynamic SQL uses the ? placeholder, or parameter marker, token.

To illustrate the difference, consider the following parallel examples:

The first is a static SQL INSERT using host variables. The second is the same statement, but performed as a dynamic SQL statement using parameter markers instead of host variables.

```
INSERT INTO parts
VALUES (:part_no, :part_desc)

INSERT INTO parts
VALUES (?, ?)
```

See "PREPARE" on page 1-29 for additional information about using placeholders in dynamic SQL statements.

Input Host Variables

Introduction

When an SQL statement with host variable inputs is sent to the Teradata client for processing, the preprocessor extracts the input values from the corresponding host input variables and then sends them to Teradata for processing.

Within stored procedures, host input variables are referred to as IN parameters. Another class of stored procedure parameter, INOUT, can be used to pass data into and out of the procedure. See "Parameters" and "Parameter Rules" on page 9-10.

Rules

The following rules apply to input host variables:

- When an input main host variable is used in an expression in a WHERE clause, or as part of a SELECT list, the data type of the host variable and the data type of the expression must be drawn from the same domain.

  You *cannot* specify an indicator host variable for input main host variables used in this way.
- When you use an input main host variable to provide data for an INSERT or UPDATE, the data type of the host variable and the data type of the expression must be drawn from the same domain.

  Teradata SQL does allow mixing of character and numeric types.

  You *can* specify an indicator host variable for input main host variables used in this way. This data type rule does not apply to an input main host variable if an associated indicator host variable (see "Indicator Variables" on page 4-20) is specified and the indicator variable shows NULL.
- If an indicator variable (see "Indicator Variables" on page 4-20) is specified for an input main host variable that corresponds to a non-nullable table column, then the indicator variable must always indicate not NULL.
- Exercise caution in using CHARACTER host variables as input to Teradata VARCHAR fields.

  Teradata strips all trailing blanks from the host value, including a single blank if that is what the host variable contains.

  For example, a host variable typed as CHAR(3) with a value of A ('A ') loaded into a Teradata database field typed as VARCHAR(10) results in the value A ('A') with the varying length set to 1. The two trailing pad characters are lost.

Similarly, if the host variable has a length of 1 and the value of the field is blank, the Teradata VARCHAR field is neither blank nor null, but is a string having length 0.

This feature differs from other systems that preserve all of the pad characters that are passed to a VARCHAR field. To preserve pad characters for a Teradata VARCHAR field, define the host variable as a VARCHAR field with length *number_of_characters* + *number_of_pad_characters*. For example, a field containing '`A  `' should be defined as VARCHAR(3) rather than CHAR(3).

Static Request Input Host Variables

Specify input variables used in static SQL requests by referencing the variable name where it is appropriate in the SQL statement.

The following statement is an example of a static request using an input host variable:

```
EXEC SQL
 SELECT field1
 FROM table1
 WHERE field2 = :var1
```

:var1 represents a properly defined host variable that is used to determine the rows to be selected from the table.

The application can change the value of var1 between SQL statement executions.

Dynamic Request Input Host Variables

Use input variables only with the following types of dynamic requests:

- Those executed using an OPEN statement for a dynamic cursor
- Those executed using an EXECUTE statement

Do not use input host variables with EXECUTE IMMEDIATE.

Input host variables in a request are represented by the ? token, referred to as a parameter marker.

When the SQL statement is performed, the preprocessor substitutes the appropriate host variable for the ? token in one of the following ways:

- By use of host variable names
- By use of an input SQLDA to describe the variables

For example, assume that the following statement has been successfully prepared using a dynamic name of S1:

```
DELETE FROM table1
WHERE field1 = ?
```

To specify the variable to be substituted for the ?, the application code would contain one of the following statements:

```
EXEC SQL
 EXECUTE S1 USING :var1
```

or

```
EXEC SQL
 EXECUTE S1 USING DESCRIPTOR INPUTDA
```

where INPUTDA is a programmer-defined SQLDA structure.

The embedded SQL preprocessor extracts the value from the host variable when the statement is performed and passes it to the Teradata server in place of the ? parameter marker token.

Output Host Variables

Introduction

When the results of a data returning SQL statement are received from the Teradata database, the embedded SQL preprocessor extracts the values and places them into the corresponding host output variables.

Valid Data Type Combinations

When you use an output main host variable to receive data from a FETCH or SELECT statement, only certain combinations of SELECT list element and host variable data types are allowed.

The valid combinations are shown in the following table. No other combinations are valid.

Most other combinations can be used by forcing the table column to change to a data type that is compatible with the data type host variable.

The data types listed in the Host Variable Data Type column are generic.

Teradata Server Column Data Type	Host Variable Data Type
BYTE(m) VARBYTE(m)	BYTE(n) VARBYTE(n) CHAR(n) VARCHAR(n) BYTEINT SMALLINT INTEGER DECIMAL NUMERIC FLOAT REAL DOUBLE PRECISION GRAPHIC(n) VARGRAPHIC(n)
CHARACTER(m) VARCHAR(m)	CHARACTER(n) VARCHAR(n)
DATE (Teradata)	CHARACTER(n) VARCHAR(n) INTEGER DECIMAL NUMERIC FLOAT REAL DOUBLE PRECISION

Teradata Server Column Data Type	Host Variable Data Type
DATE (ANSI) TIME TIMESTAMP INTERVAL	CHARACTER(n)
BYTEINT SMALLINT INTEGER, DECIMAL NUMERIC FLOAT REAL DOUBLE PRECISION	BYTEINT SMALLINT INTEGER DECIMAL NUMERIC FLOAT REAL DOUBLE PRECISION
GRAPHIC(m) VARGRAPHIC(m)	GRAPHIC(n) VARGRAPHIC(n)

Assignment Rules

The following table explains assignment rules for output host variables.

Data Type	Condition	Description
BYTE	m < n	m bytes of data are moved to the host variable with (n - m) bytes of x'00' added.
	m = n	m bytes of data are moved to the host variable.
	m > n	n bytes of data are moved to the host variable; the indicator, if used, is set to m; SQLWARN1 in the SQLCA is set to 'W.' m represents the length of the data. n represents the length of the host variable. BYTEINT, SMALLINT and INTEGER have implied lengths of 1, 2 and 4, respectively. DECIMAL can have a length from 1 to 15 bytes. FLOAT can be single (4 bytes) or double (8 bytes). No data conversion is performed when a BYTE field is assigned to a host variable. The application is responsible for processing the value returned.

Data Type	Condition	Description
VARBYTE	m <= n	m bytes of data are moved to the host variable.
	m > n	n bytes of data are moved to the host variable; the indicator, if used, is set to m; SQLWARN1 in the SQLCA is set to 'W.' m represents the current length of the data; n represents the maximum length of the host variable. BYTEINT, SMALLINT and INTEGER have implied lengths of 1, 2 and 4, respectively. DECIMAL can have a length from 1 to 15 bytes. FLOAT can be single (4 bytes) or double (8 bytes). No data conversion is performed when a BYTE field is assigned to a host variable. The application is responsible for processing the returned value.
CHARACTER	m < n	m bytes of data are moved to the host variable with (n - m) bytes of blanks (x'40' in EBCDIC, x'20' in ASCII environments) added.
	m = n	m bytes of data are moved to the host variable.
	m >n	n bytes of data are moved to the host variable; the indicator, if used, is set to m; SQLWARN1 in the SQLCA is set to 'W.' m represents the length of the data; n represents the length of the host variable.
VARCHAR	m <= n	m bytes of data are moved to the host variable.
	m > n	n bytes of data are moved to the host variable; the indicator, if used, is set to m; SQLWARN1 in the SQLCA is set to 'W.' m represents the current length of the data; n represents the maximum length of the host variable.
DATE (Teradata)		**into a CHARACTER field**: If Teradata format is requested, n must be at least 8 bytes. All other formats require n to be at least 10 bytes. Remaining bytes are set to blanks (x'40' in EBCDIC, x'20' in ASCII environments). SQLCODE in the SQLCA is set to -304 if the host variable cannot contain the requested date format.
		into a numeric field: The value must be representable in the type specified without losing leading digits. SQLCODE in the SQLCA is set to -304 if the host variable cannot contain the data.

Data Type	Condition	Description
DATE (ANSI) TIME TIMESTAMP INTERVAL		If Teradata format is requested, n must be at least 8 bytes. All other formats require n to be at least 10 bytes. Remaining bytes are set to blanks (x'40' in EBCDIC, x'20' in ASCII environments). SQLCODE in the SQLCA is set to -304 if the host variable cannot contain the requested date format.
BYTEINT		The value must be representable in the type specified without losing leading digits. SQLCODE in the SQLCA is set to -304 if the host variable cannot contain the data.
SMALLINT		The value must be representable in the type specified without losing leading digits. SQLCODE in the SQLCA is set to -304 if the host variable cannot contain the data.
INTEGER		The value must be representable in the type specified without losing leading digits. SQLCODE in the SQLCA is set to -304 if the host variable cannot contain the data.
DECIMAL		The value must be representable in the type specified without losing leading digits. SQLCODE in the SQLCA is set to -304 if the host variable cannot contain the data.
NUMERIC		The value must be representable in the type specified without losing leading digits. SQLCODE in the SQLCA is set to -304 if the host variable cannot contain the data.
FLOAT		The value must be representable in the type specified without losing leading digits. SQLCODE in the SQLCA is set to -304 if the host variable cannot contain the data.
REAL		The value must be representable in the type specified without losing leading digits. SQLCODE in the SQLCA is set to -304 if the host variable cannot contain the data.
DOUBLE PRECISION		The value must be representable in the type specified without losing leading digits. SQLCODE in the SQLCA is set to -304 if the host variable cannot contain the data.

SQL Character Strings as Host Variables

Introduction

The preprocessor treats SQL character strings as a third kind of host variable that is neither input nor output.

Definition

An SQL string is a series of characters used to complete an embedded SQL command. It is not an input or an output variable because it does not correspond to a field in a row of a table.

Character Strings as Host Variables

SQL character strings are a distinct category of host variable because some host languages apply special rules to them. Those rules are detailed in the language-dependent chapters of *Teradata Preprocessor2 Programmer Guide*.

Character strings can require a leading COLON character when referenced in an embedded SQL statement. For details, see the individual statement syntax documentation in Chapter 3: "SQL Data Manipulation Language Statement Syntax" and in this chapter.

Statements That Use Strings as Host Variables

The following table lists embedded SQL statements that use SQL strings as host variables.

This SQL statement . . .	Uses an SQL string as a host variable . . .
CHECKPOINT	when the checkpoint label is expressed as a host variable.
DATABASE	when the database name is expressed as a host variable.
EXECUTE IMMEDIATE	when the SQL statement string is expressed as a host variable.
LOGON	for the logon string.
PREPARE	when the SQL statement string is expressed as a host variable.
SET CHARSET	when the character set name is expressed as a host variable.

Indicator Variables

Introduction

You can optionally associate an indicator host variable with any main host variable. The value of indicator variables is set by the sending agent (client application-to-Teradata database or Teradata database-to-client application) in a client-server data exchange to inform the receiving agent if the host variable is null.

Syntax



where:

Syntax element . . .	Specifies . . .
host_variable_name	the name of a host variable with which *indicator_variable_name* is associated.
INDICATOR	an optional keyword to help distinguish *indicator_variable_name* from *host_variable_name*.
indicator_variable_name	the name of the indicator variable

Rules and Guidelines for Indicator Variables

The following rules and guidelines apply to the use of indicator variables.

- All indicator variables must be preceded by a COLON character, whether a COLON character precedes its associated main host variable or not.
- Specify the indicator host variable immediately following the main host variable with which it is associated (for example, :MainVar:IndVar or : HostMainVar : HostIndVar).

  To avoid confusion, precede the indicator variable specification by the word INDICATOR (that is :MainVar INDICATOR :IndVar).
- Indicator variables can be used in WHERE clause conditions.

How Indicator Variables Are Used With Host Variables

For an input host variable, the application program uses an indicator variable to inform Teradata if the host variable is null.

For an output host variable, Teradata uses an indicator variable to inform the application program if the host variable is null or was truncated when the value was placed into the host variable.

The following table defines the relationship between indicator variable values and input host variables.

This indicator variable value . . .	Reports that its associated input host variable is . . .
-n where *n* is typically 1	null.
0	non-null and successfully returned to the output host variable.
+n	truncated. The truncation occurred when returning a character or byte string to the main variable associated with the indicator variable. The value *n* reports the original length of the string before truncation.

Processing of Indicator Variables

Indicator variables are processed by Teradata as follows:

- One indicator variable corresponds to each data item (field) of a response row.
- Each indicator variable occupies one bit of space.
- If there are *n* data fields, the first $(n + 7)/8$ bytes of a response row contain the indicator variables for the data in that row.

  For example, if a response row contains 19 data items, then $(19 + 7)/8 = 3$ bytes contain the indicator variables for that row.
- Indicator variables are held in the minimum number of 8-bit bytes required to store them.

  Unused bits are set to binary 0.

Internal Processing of Indicator Variables

Internally, Teradata uses CLIv2 Indicator mode to return data in the NullIndicators field of the Data field of a Record parcel in the internal format used by the client system.

Immediately preceding the first response row is a DataInfo parcel containing information on the total number of columns returned, the data type, and length of each column.

Each response row begins with indicator variables corresponding to each data item in that row.

Indicator Variables and DateTime and Interval Data

DateTime and Interval values are cast as CharFix, and the DataInfo parcel created for Indicator Variable output follows that rule with the exception of DATE values in INTEGERDATE (Teradata-style DATE) mode.

You can export values in IndicData mode and subsequently import in Data mode with a USING phrase built to properly type any DateTime or Interval values in the import records.

If the exported values are to be used as data for INSERT or UPDATE statements, Teradata implicitly casts USING values that are CharFix and have the right length for the target DateTime or Interval type.

See "USING Row Descriptor" on page 3-168.

Example 1

In this example, when the value for the indicator variable is -1, the employee number or department number is set to null.

When the indicator variable is 0, then the employee number or department number is set to the value reported to the host variable.

```
EXEC SQL
 INSERT INTO EMPLOYEE
 VALUES (:EMPNO INDICATOR :EMPNO-INDIC,:DEPTNO INDICATOR
 :DEPTNO-INDIC)
END-EXEC.
```

Example 2

In this example, Department Number is defined to be null.

```
MOVE -1 TO DEPTNO-INDIC.
EXEC SQL
 UPDATE EMPLOYEE
 SET DEPARTMENT_NUMBER = :DEPTNO
 INDICATOR :DEPTNO-INDIC
END-EXEC.
```

Multistatement Requests With Embedded SQL

Multistatement Requests Require Cursors

A multistatement request often returns a response having more than one row. Each statement in the request produces its own results (success/failure, activity count and so on), which are returned to the application program.

Because the results table for a multistatement request returns more than one response, you *must* declare a cursor to fetch and manipulate the results.

TO associate a . . .	YOU must . . .
static multistatement request with a request cursor	issue a DECLARE CURSOR statement for a request cursor.
dynamic multistatement request with a dynamic cursor	use a PREPARE statement with the statement string containing a multistatement request. The dynamic request is then associated with a dynamic cursor.

Using the FOR STATEMENT Clause With PREPARE and DESCRIBE

You can extend the syntax of PREPARE and DESCRIBE by using the FOR STATEMENT clause. FOR STATEMENT permits you to specify which of the statements in the multistatement request is to be described.

To describe all statements of a multistatement request, the DESCRIBE statement must be executed multiple times for each data returning statement within the request.

Even though the output SQLDA contains no column descriptions, it is always valid to DESCRIBE a non-data-returning statement.

For further information, see "DESCRIBE" on page 4-55 and "PREPARE" on page 4-63.

Using SQLDA to Track Statement Status

In processing the output from a multistatement request, you must know the success or failure of each statement and when the output from one request ends and output from the next begins.

The mechanism described by the following table, which is similar to that used for single statement requests, provides a framework for achieving this:

WHEN . . .	THEN . . .
the request cursor is opened	SQLCODE in SQLCA is set to reflect the success of the first statement of the request or the failure (failure defined as an IMPLICIT ROLLBACK occurrence) of the request as an entity. A failure condition overrides a success report. If successful, the activity count is reported to the application in the third SQLERRD element in the SQLCA.
the statement is in error (error defined as a non-implicit ROLLBACK)	the next FETCH returns the appropriate error code: SQLCODE in the SQLCA is < 0 and the error code in the first element of SQLERRD.
no rows are returned or the rows returned for a particular statement are exhausted	SQLCODE is set to +100 on return from the FETCH, just as with a single statement request.
the application needs to position to the next (or any) statement of the request	use the POSITION statement. POSITION moves control to the output for the specified statement of the request and sets the SQLCA information based on the success or failure of the OPEN request. The program can then use FETCH to retrieve the output of the statement.
the application needs to position to the beginning of all output for the request	use the REWIND statement. The REWIND statement is exactly equivalent to POSITION ... TO STATEMENT 1. REWIND moves control to the output for the specified statement of the request and sets the SQLCA information based on the success or failure of the OPEN request. The program can then use FETCH to retrieve the output of the statement.
you receive +100 SQLCODE for the current statement	use POSITION or REWIND to access the results of another (or even the same) statement. You do not need to wait until the +100 is received. You can issue POSITION or REWIND statements at any time.

See "DECLARE CURSOR" on page 7-17 for further information on using cursors with multistatement requests.

Multistatement Request Example

An example of a multistatement request is shown in the following passages. The SQL prefixes and terminators are omitted. This example assumes successful completion of the statements in the request.

```
DECLARE curs CURSOR FOR
'SELECT ent1,ent2,ent3 FROM tabx;
UPDATE ...;SELECT entt FROM tabl'

OPEN curs {SQLCA gets first SELECT result}
WHENEVER NOT FOUND GOTO updstmt

selstmt1:
FETCH curs INTO :vara,:varb,:varc
.
.
GOTO selstmt1

updstmt:
WHENEVER NOT FOUND CONTINUE
POSITION curs TO STATEMENT 2 {SQLCA gets UPDATE
result}
FETCH curs
.
.
WHENEVER NOT FOUND GOTO reread
POSITION curs TO STATEMENT 3 {SQLCA gets second
SELECT result}

selstmt2:
FETCH curs INTO :vars
.
.
GOTO selstmt2

reread:
REWIND curs {SQLCA gets first SELECT result}
WHENEVER NOT FOUND GOTO alldone

selstmt1x:
FETCH curs INTO :varaa,:varbb,:varcc
.
.
GOTO selstmt1x

alldone:
CLOSE curs
```

This section describes embedded SQL-only declarations and miscellaneous statements, including the following statements:

- BEGIN DECLARE SECTION
- COMMENT (Returning Form)
- DATABASE
- DECLARE STATEMENT
- DECLARE TABLE
- END DECLARE SECTION
- END-EXEC Statement Terminator
- EXEC
- EXEC SQL Statement Prefix
- INCLUDE
- INCLUDE SQLCA
- INCLUDE SQLDA

Additional embedded SQL statements are described in the following chapters:

- Chapter 3: “SQL Data Manipulation Language Statement Syntax”
- Chapter 5: “Client-Server Connectivity Statements”
- Chapter 6: “Multisession Asynchronous Programming With Embedded SQL”
- Chapter 7: “Cursors and Cursor Control Statements”

Dynamic SQL and its related SQL statements are described beginning with “Dynamic SQL” on page 4-52.

Miscellaneous Embedded SQL-Only Statement Syntax

SQL declarative and other miscellaneous embedded SQL-only statements are described in the following pages:

- "BEGIN DECLARE SECTION" on page 4-29
- "COMMENT (Returning Form)" on page 4-30
- "DATABASE" on page 4-32
- "DECLARE STATEMENT" on page 4-34
- "DECLARE TABLE" on page 4-35
- "END DECLARE SECTION" on page 4-37
- "END-EXEC Statement Terminator" on page 4-38
- "EXEC" on page 4-39
- "EXEC SQL Statement Prefix" on page 4-40
- "INCLUDE" on page 4-42
- "INCLUDE SQLCA" on page 4-44
- "INCLUDE SQLDA" on page 4-46
- "WHENEVER" on page 4-48

The following statements, also used only for embedded SQL, are used with positioned cursors. They are documented in Chapter 3: "SQL Data Manipulation Language Statement Syntax."

- "DELETE (Positioned Form)" on page 3-46
- "UPDATE (Positioned Form)" on page 3-157

BEGIN DECLARE SECTION

Purpose

Identifies the start of an embedded SQL declare section.

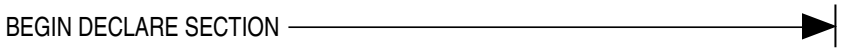
Invocation

Nonexecutable.

Preprocessor declaration.

Embedded SQL only.

Syntax



GW01A001

ANSI Compliance

BEGIN DECLARE SECTION is ANSI SQL-99-compliant.

Authorization

None.

Usage Notes

You must specify the complete statement, including the SQL prefix and terminator, on one line. Only a pad character can separate the words of the statement.

This statement and the END DECLARE SECTION statement (used to identify the end of an embedded SQL declare section: see "END DECLARE SECTION" on page 4-37) are mandatory for applications written in C.

The preprocessor issues a warning if either statement appears in a COBOL or PL/I application.

All host variables must be defined within the declare section.

Related Information

See "END DECLARE SECTION" on page 4-37.

COMMENT (Returning Form)

Purpose

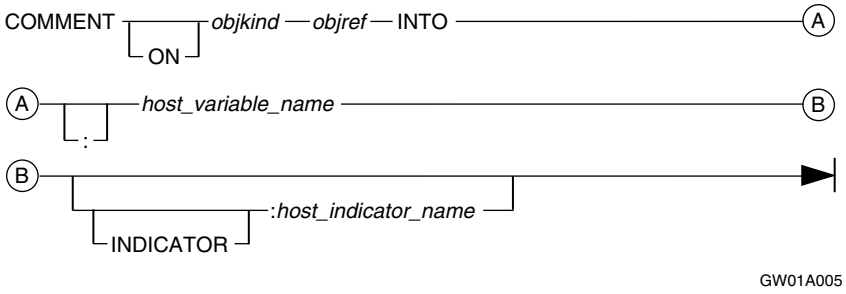
Returns the comment (if any) that belongs to an object.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax element …	Specifies …
object_kind	One of these objects: • COLUMN • DATABASE • MACRO • PROCEDURE • TABLE • TRIGGER • USER • VIEW
object_reference	One of these object references: • column name • database name • macro name • procedure name • table name • trigger name • user name • view name

Syntax element . . .	Specifies . . .
host_variable_name	the name of the host variable into which the comment is to be placed. The preceding COLON is optional; however, its use is strongly recommended. Blanks before and after the colon are optional. The rules for the name follow the client programming language conventions.
host_indicator_name	the name of the host indicator variable.

ANSI Compliance

COMMENT is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

COMMENT Returns Data

The returning form of COMMENT returns data.

Rules

The following rules apply to the returning form of COMMENT:

- The data type of *host_variable_name* must be VARCHAR(255).
- If no comment exists for the specific object, *host_indicator_name* returns NULL.
- Although the COMMENT statement returns only one data value (in effect, a single row containing a single column), you can use a selection cursor with a static COMMENT statement. Use the same procedure for the cursor as for a static selection cursor.
- If you perform a dynamic COMMENT statement, then you must use a dynamic cursor because data is returned. In this case, the same procedure is followed as with dynamic selection.
- If you use COMMENT with a cursor or as a dynamic SQL statement, then you must omit the INTO clause.

DATABASE

Purpose

Specifies a default database.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax element ...	Is the database name to be used with the statement as a ...
database_name	SQL identifier.
database_name_variable	host variable. The colon is mandatory.

ANSI Compliance

DATABASE is a Teradata extension to the ANSI SQL-99 standard.

Usage Notes

The following rules apply to the DATABASE statement:

- Whether specified as *database_name* or as *database_name_variable*, the database name must be a valid SQL identifier.
- If you use the *database_name_variable* form, the host variable must follow the rules for SQL strings for the client language.
- You must ensure that your DATABASE specification is consistent with your DATABASE or -db preprocessor specification.

  While the statements and the options need not name the same database, all unqualified object references in the application program *must* resolve at application execution time to objects that are compatible with the ones they resolve to at precompile time.
- Referenced objects in multiple databases should use fully qualified names. Name resolution problems may occur if referenced databases contain tables or views with identical names and these objects are not fully qualified. Name resolution problems may even occur if the identically named objects are not themselves referenced.
- DATABASE is not valid when you specify the TRANSACT(2PC) option to the preprocessor.

DECLARE STATEMENT

Purpose

Declares the names used to identify prepared dynamic SQL statements.

Invocation

Nonexecutable.

Preprocessor declaration.

Embedded SQL only.

Syntax



where:

Syntax element …	Specifies …
statement_name	the name associated with a previously prepared statement.

ANSI Compliance

DECLARE STATEMENT is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Usage Notes

DECLARE STATEMENT is used for program documentation only.

You can specify multiple statement names, but each must be separated by a comma.

DECLARE TABLE

Purpose

Declares tables used in the embedded SQL statements within the application.

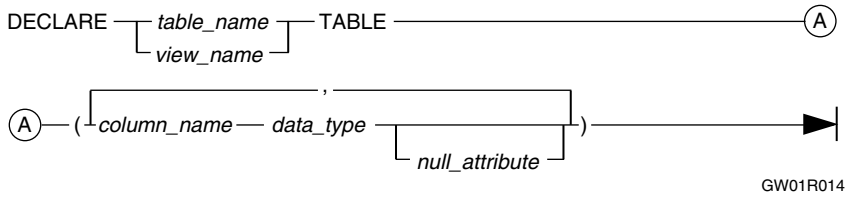
Invocation

Nonexecutable.

Preprocessor declaration.

Embedded SQL only.

Syntax



where:

Syntax element …	Specifies …
table_name	the name of the table to be declared. If the same name is used in a CREATE TABLE statement in your program, the description of the table in the CREATE TABLE statement and the DECLARE TABLE statement must be identical.
view_name	the name of the view to be declared. If the same name is used in a CREATE TABLE statement in your program, the description of the table in the CREATE TABLE statement and the DECLARE TABLE statement must be identical.
column_name	the name of a column or columns being declared for the table.
data_type	the data type for *column_name*.
null_attribute	the nullability specification for *column_name*.

IF null_attribute is …	THEN nulls are …
NOT NULL	not permitted and there is no default value specified.
NOT NULL WITH DEFAULT	not permitted and a default value is specified.
not specified	permitted.

ANSI Compliance

DECLARE TABLE is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Usage Notes

DECLARE TABLE is useful for program documentation.

The preprocessor accepts DECLARE TABLE, but treats it as a comment.

The preprocessor performs no verification of the syntax or correctness of the field definitions other than identifying, in order:

1. The DECLARE keyword
2. The presence of a *table_name* or *view_name*
3. The TABLE keyword

END DECLARE SECTION

Purpose

Identifies the end of an embedded SQL declare section.

Invocation

Nonexecutable.

Preprocessor declaration.

Embedded SQL only.

Syntax



ANSI Compliance

END DECLARE SECTION is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Usage Notes

The complete statement (including the SQL prefix and terminator) must be specified on one line. Only a pad character can separate the words of the statement.

END DECLARE SECTION is mandatory for applications written in C. The preprocessor issues a warning if this statement is used in a COBOL or PL/I application.

Related Information

See "BEGIN DECLARE SECTION" on page 4-29.

END-EXEC Statement Terminator

Purpose

Terminates an SQL statement in an embedded SQL client COBOL application program.

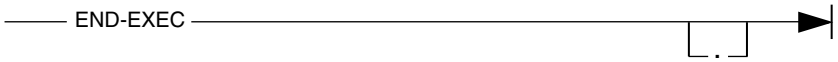
Invocation

Nonexecutable.

Preprocessor declaration.

Embedded SQL only.

Syntax



FF07D287

ANSI Compliance

END-EXEC is ANSI SQL-99-compliant.

Authorization

None.

Rules

The following rules apply to END-EXEC:

- END-EXEC is mandatory for all SQL statements embedded in a client COBOL application program.
- END-EXEC cannot be used with interactive SQL statements.
- The SQL statement terminator for C and PL/I is the semicolon.

Related Information

See "EXEC SQL Statement Prefix" on page 4-40.

EXEC

Purpose

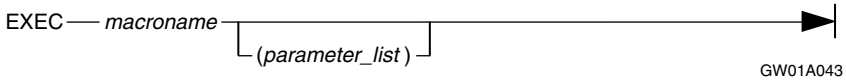
Performs a Teradata SQL macro.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax element …	Specifies …
macro_name	the name of the macro to be performed.
parameter_list	the Teradata SQL macro parameters.

ANSI Compliance

EXEC is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Rules

The following rules apply to EXEC.

- The statement *must* be spelled EXEC to distinguish it from the dynamic SQL statement EXECUTE.
- The macro specified by *macro_name* can contain no more than a single Teradata SQL statement.
- The macro specified by *macro_name* cannot return data.
- You must use a macro cursor to perform the following types of macros:
  - Multistatement
  - Data returning

EXEC SQL Statement Prefix

Purpose

Denotes the beginning of an SQL statement in an embedded SQL application.

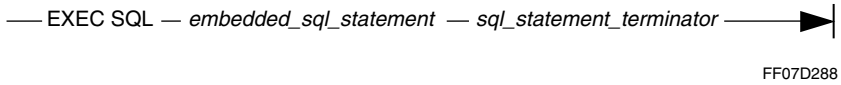
Invocation

Nonexecutable.

Preprocessor declaration.

Embedded SQL only.

Syntax



where:

Syntax element ...	Specifies ...
embedded_sql_statement	the embedded SQL statement to be performed by the client application program.
sql_statement_terminator	the SQL statement terminator for the client language.

FOR this client language ...	The SQL statement terminator is ...
COBOL	END-EXEC
C	;
PL/I	

ANSI Compliance

EXEC SQL is ANSI SQL-99-compliant.

Authorization

None.

Rules

The following rules apply to EXEC SQL:

- EXEC SQL must precede each SQL statement embedded in a client application program.
- The words EXEC SQL must appear together on a single line.
- The SQL statement that follows can begin immediately following the words EXEC SQL or it can begin on a new line.
- EXEC SQL cannot be used with interactive SQL statements.

Related Information

See "END-EXEC Statement Terminator" on page 4-38.

INCLUDE

Purpose

Incorporates an external source file into the application program.

Invocation

Nonexecutable.

Preprocessor declaration.

Embedded SQL only.

Syntax



GW01A020

where:

Syntax element ...	Specifies ...
text_name	the name of the source file to be included. If *text_name* is SQLCA or SQLDA, a special case results. INCLUDE SQLCA and INCLUDE SQLDA are not instances of INCLUDE and are defined separately (see "INCLUDE SQLCA" on page 4-44 and "INCLUDE SQLDA" on page 4-46).

ANSI Compliance

INCLUDE is ANSI SQL-99-compliant.

Authorization

None.

Rules

The following rules apply to the INCLUDE statement:

- Under OS/390, the library input PDS (SYSLIB) is searched for a member which has the specified name.
- Under VM/CMS, the standard CMS disk path is searched for a file with the specified name and a language specific file type:

For this language ...	The INCLUDE file type is ...
C	CCOPY
COBOL	COBCOPY
PL/I	PLICOPY

- For network-attached platforms, the directory path is searched for a file having the specified name and a language specific file extension:

For this language ...	The INCLUDE file type is ...
C	pc
COBOL	pb
PL/I	pi

- The INCLUDE statement is effectively replaced by the included text in the application program input to the preprocessor.
- The included text can contain any embedded SQL statements, except another INCLUDE because INCLUDE statements cannot be nested.
- You *can* specify INCLUDE SQLCA and INCLUDE SQLDA statements in the included text.

INCLUDE SQLCA

Purpose

Defines the SQL Communications Area (SQLCA) in a client embedded SQL application program.

Invocation

Nonexecutable.

Preprocessor declaration.

Embedded SQL only.

Syntax



ANSI Compliance

INCLUDE SQLCA is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Rules

The following rules apply to INCLUDE SQLCA:

- When operating in Teradata mode, you must have an SQL Communications Area defined in your embedded SQL application program.
- When operating in ANSI mode, the INCLUDE SQLCA statement is flagged as an error.

  ANSI SQL requires you to explicitly define a variable named SQLCODE.

  If you are operating in ANSI mode, you can also define an SQLSTATE variable to receive error codes.
- Exactly one declaration of the SQL Communications Area is required in every application program. Either an INCLUDE SQLCA statement or an equivalent user-supplied declaration can be used.

- In COBOL, the SQL Communications Area declaration must appear in the WORKING STORAGE SECTION.
- The complete statement (including the SQL prefix and terminator) must be specified on one line. Only a pad character can separate the words of the statement. No other statements can appear on the line.
- The INCLUDE SQLCA statement is effectively replaced by the appropriate language definition of the SQL Communications Area in the application program input to the SQL compatible preprocessor. No library path (OS/390), disk path (VM/CMS) or directory (network) search is performed for this statement.

Related Information

See Appendix C: “SQL Communications Area (SQLCA)”.

INCLUDE SQLDA

Purpose

Defines the SQL Descriptor Area (SQLDA) in the application program.

Invocation

Nonexecutable.

Preprocessor declaration.

Embedded SQL only.

Syntax



ANSI Compliance

INCLUDE SQLDA is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Rules

The following rules apply to INCLUDE SQLDA:

- In PL/I, the SQLDA declaration is defined as a based structure with a varying (REFER) substructure. This makes it suitable for use with multiple SQL Descriptor Areas.
- The INCLUDE SQLDA statement is not used in COBOL because COBOL does not support based structures.

  If SQL Descriptor Areas are needed in a COBOL program, you must code them yourself and insert them into the WORKING STORAGE SECTION of your program.
- The complete statement (including the SQL prefix and terminator) must be specified on one line.

  Only a pad character can separate the words of the statement.

  Additional statements cannot appear on the same line.
- Either an INCLUDE SQLDA statement or an equivalent user-supplied declaration of the SQL Descriptor Area is required in every application program that uses SQL data descriptions.
- The preprocessor replaces the INCLUDE SQLDA statement with the appropriate language definition of the SQL Descriptor Area.

  No library path (OS/390), disk path (VM/CMS) or directory (network) search is performed for this statement.

Related Information

See Appendix B: “SQL Descriptor Area (SQLDA)”.

WHENEVER

Purpose

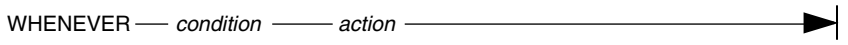
Specifies the action to be taken when an exception condition occurs.

Invocation

Executable.

Embedded SQL only.

Syntax



GW01R035

where:

Syntax element …	Specifies …
condition	a status keyword that indicates the type of condition for which the indicated action is to be undertaken. The valid *condition* keywords and their definitions are listed in the following tables. Following each keyword definition is a table that describes the values for the SQLCODE and SQLSTATE variables when the condition occurs.

Keyword	Definition
SQLERROR	a condition in which an SQL error occurs.

This variable …	Has this value …
SQLCODE	< 0
SQLSTATE	See *"SQLSTATE" on page 8-2*.
NOT FOUND	a condition in which no data is found.

<table>
<tr><th>Syntax element …</th><th>Specifies …</th></tr>
<tr><td>condition
(continued)</td><td>
<table>
<tr><th>THIS variable …</th><th>Has this value …</th></tr>
<tr><td>SQLCODE</td><td>+100</td></tr>
<tr><td>SQLSTATE</td><td>02xxx
See “SQLSTATE” on page 8-2.</td></tr>
</table>
SQLWARNING is a non-ANSI Teradata extension.
<table>
<tr><th>Keyword</th><th>Definition</th></tr>
<tr><td>SQLWARNING</td><td>a condition in which an SQL warning occurs.</td></tr>
</table>
<table>
<tr><th>This variable …</th><th>Has this value …</th></tr>
<tr><td>SQLCODE</td><td>a positive number other than +100.</td></tr>
<tr><td>SQLSTATE</td><td>not defined.</td></tr>
</table>
</td></tr>
<tr><td>action</td><td>
the action to be performed when condition occurs.
The valid actions are:

CONTINUE
GO TO :host_label
GOTO :host_label
PERFORM code
CALL function_call

where:
<table>
<tr><th>Syntax element …</th><th>Specifies …</th></tr>
<tr><td>:host_label</td><td>a valid target of a client language GO TO statement.
Use of a preceding colon is strongly recommended.</td></tr>
<tr><td>code</td><td>the name of a section or paragraph in the application to be performed when the exception condition occurs.
The PERFORM action is valid only for COBOL.</td></tr>
<tr><td>function_call</td><td>the function to be called when the exception condition occurs.</td></tr>
</table>
</td></tr>
</table>

ANSI Compliance

WHENEVER is ANSI SQL-99-compliant with extensions.

Authorization

None.

Rules

The following rules apply to the WHENEVER statement:

- If the precompiler SQLFLAGGER option is set to ENTRY, WHENEVER SQLWARNING causes a precompiler warning.
- The rules for the object of a GO TO are language-dependent. See *Teradatan Preprocessor2 Programmer Guide* for details.
- The initial implied exception declaration is always CONTINUE.
- An exception declaration applies to a particular SQL statement only if that statement follows the exception declaration in the text of the program and there are no intervening exception declarations for the same exception condition.

IF an exception condition applies and the action is . . .	THEN the application program continues execution at the . . .
CONTINUE	next sequential instruction. The exception condition is ignored.
GOTO	specified target location. *host_label* must be such that a client language GO TO statement specifying that target is valid at every SQL statement to which the exception declaration applies.
CALL	next sequential instruction only after the specified subprogram has been performed (called) and control has been returned to the calling program. A corresponding client statement (CALL function call for COBOL and PL/I or function call for C) must be valid at every SQL statement to which the exception declaration applies.
PERFORM	next sequential instruction only after the specified COBOL paragraphs or sections have been performed. A corresponding COBOL statement (PERFORM code) must be valid at every SQL statement to which the exception declaration applies.

- The following SQLCODE definitions apply.

IF SQLCODE has this value following the performance of an SQL statement . . .	THEN the following exception condition applies . . .
any negative number	SQLERROR
a positive number other than +100	SQLWARNING
+100	NOT FOUND

Dynamic SQL

Dynamic SQL[1] is a facility that permits an interactive user to submit SQL statements dynamically to an application written using embedded SQL or stored procedures.

Dynamic SQL is useful for situations where you do not know the full text of some or all of the SQL statements your application will require at runtime. An example might be a spreadsheet-driven application where when a user interactively types one or more formulas into the cells. These formulas are then translated into SQL statements that retrieve the data needed to perform the calculations specified by each spreadsheet formula. Because you cannot know in advance what SQL statements are required to support the ad hoc formulas the user might type, you would code the application using dynamic SQL features to support the dynamic requirements of the spreadsheet.

You can perform SQL statements dynamically in either prepared[2] or immediate form, as described by the following table:

Dynamic SQL Statement Type	Description
Prepared	Valid statement text is prepared and preserved for the duration of the session and the statement can be performed as many times as the application requires. There is some overhead in preparing an SQL statement, somewhat similar to the overhead required to compile and preprocess static SQL for an embedded SQL application. See "EXECUTE (Dynamic SQL Form)" on page 4-59 and "PREPARE" on page 4-63 for more information about preparing dynamic SQL statements in embedded SQL applications.
Immediate	Valid statement text is performed one time only. If a statement must be performed more than one time in an application, then you must incur the overhead of preparing and performing it each time. See "EXECUTE IMMEDIATE" on page 4-61 for more information about immediate preparation and performance of dynamic SQL statements in embedded SQL applications.

1 When contrasted with dynamic SQL, the facilities provided by ordinary SQL are often referred to as static SQL.

2 Stored procedures only support the immediate form of dynamic SQL. See"Using Dynamic SQL in Stored Procedures" on page 9-31.

An overview of stored procedure support for dynamic SQL is provided in "Using Dynamic SQL in Stored Procedures" on page 9-31.

You should always use static SQL if possible because dynamic SQL has a significant processing overhead that often retards system performance.

Support for dynamic SQL within embedded SQL applications is provided by the following set of SQL statements:

- "DECLARE CURSOR (Dynamic SQL Form)" on page 7-19
- "DESCRIBE" on page 4-55
- "EXECUTE (Dynamic SQL Form)" on page 4-59
- "EXECUTE IMMEDIATE" on page 4-61
- "PREPARE" on page 4-63

Dynamic SQL Statement Syntax

The set of SQL statements unique to dynamic SQL is described in the following pages. Those statements provided in the following bulleted list:

- "DESCRIBE" on page 4-55
- "EXECUTE (Dynamic SQL Form)" on page 4-59
- "EXECUTE IMMEDIATE" on page 4-61
- "PREPARE" on page 4-63

There is also a form of DECLARE CURSOR that is specific to dynamic SQL (see "DECLARE CURSOR (Dynamic SQL Form)" on page 7-19).

Stored procedures also support dynamic SQL, but in a completely different way than embedded SQL. See "Using Dynamic SQL in Stored Procedures" on page 9-31 for further information.

DESCRIBE

Purpose

Obtains information about the data to be returned when a previously prepared dynamic SQL statement is performed.

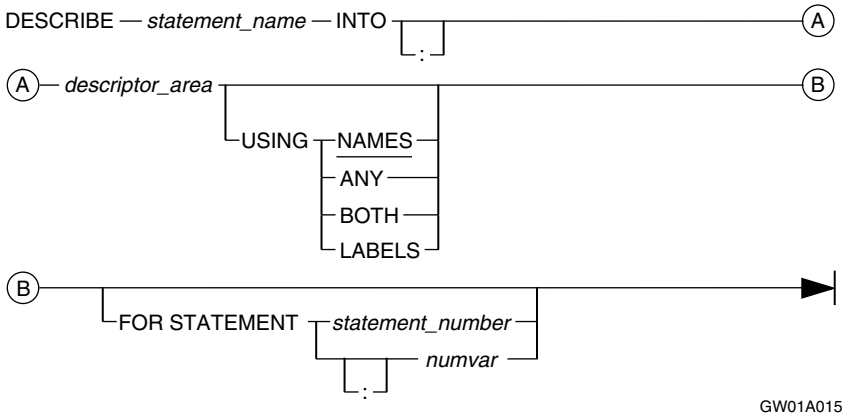
Invocation

Executable.

Embedded SQL only.

Dynamic SQL.

Syntax



where:

Syntax element ...	Specifies ...
statement_name	the name associated with the previously prepared statement. Must be a valid SQL identifier, not enclosed in quotes.
descriptor_area	the area to receive the information about the data which will be returned when the previously prepared statement is performed. Must identify an SQLDA. You can specify *descriptor_area* in C programs as a name or as a pointer reference (*sqldaname) when SQLDA is declared as a pointer. See Teradata *Preprocessor2 Programmer Guide* for additional details.

Syntax element . . .	Specifies . . .
statement_number	the statement number within the request for which the information is required. Must be a valid integer numeric literal.
numeric_variable	the statement number within the request for which the information is required. Must identify a host variable which conforms to INTEGER or SMALLINT.

ANSI Compliance

DESCRIBE is a Teradata extension to the ANSI SQL-99 standard.

DESCRIBE performs an analogous function to the ANSI statements DESCRIBE INPUT and DESCRIBE OUTPUT.

Authorization

None.

General Rules

The following rules apply to the DESCRIBE statement:

- An SQLDA must be defined.
- The statement specified by *statement_name* must be prepared within the same transaction.
- If the prepared statement is a non-data returning statement, no useful information is obtained other than verification that the prepared statement is not a data returning statement.
- DESCRIBE cannot be performed as a dynamic SQL statement.

USING Clause Rules

The USING clause affects the information in SQLDA as follows:

IF this form of USING is specified . . .	THEN column . . .
NAMES	names are placed in the SQLNAME fields of the SQLDA. This also happens if the USING clause is omitted.
LABELS	titles are placed in the SQLNAME fields of the SQLDA. If a TITLE clause is specified for a column on the SELECT, that title is returned. If a column was defined with a TITLE at CREATE time and no title appears on the SELECT, that title is returned. In the absence of either, the default Teradata title (column name) is returned.
BOTH	names and titles are placed in the SQLNAME fields of the SQLDA. In this case, the SQLVAR array must have two elements per column (2 * n elements). The first set of elements contains the names and column information, while the second set contains the titles of the columns.
ANY	titles or names are placed in the SQLNAME fields of the SQLDA. If a column has a title, then that title is placed in the SQLNAME field If a column has no title, then the column name is placed in the SQLNAME field.

FOR STATEMENT Clause Rules

The following rules apply to the FOR STATEMENT clause.

- The FOR STATEMENT clause is intended to support dynamic multistatement requests, but can also be used for single statement requests.
- If the FOR STATEMENT clause is not included, the descriptive information is returned for the first or only SQL statement of the prepared dynamic SQL request.
- If the FOR STATEMENT clause is included, the descriptive information is returned for that SQL statement of the prepared dynamic SQL request which is identified by the integer numeric operand of the clause.

  If there is no such statement (for example, if FOR STATEMENT 3 is coded and the request contains only two statements), the value -504 is returned in SQLCODE, and no information is returned.

Related Information

See:

- “EXECUTE (Dynamic SQL Form)” on page 4-59
- “EXECUTE IMMEDIATE” on page 4-61
- “PREPARE” on page 4-63

EXECUTE (Dynamic SQL Form)

Purpose

Performs a prepared dynamic SQL statement.

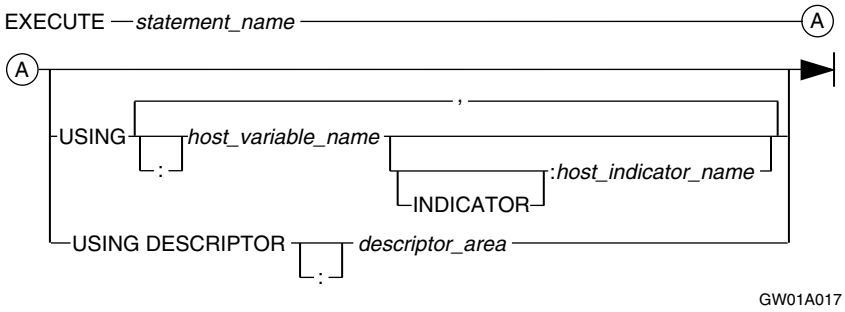
Invocation

Executable.

Embedded SQL only.

Dynamic SQL statement.

Syntax



where:

Syntax element . . .	Specifies . . .
statement_name	the name associated with the previously prepared statement.
host_variable_name	the variable used as input data for the prepared statement. The colon preceding the name or names is optional.
host_indicator_name	the indicator variable. The colon preceding the name is mandatory.
descriptor_area	an SQL Descriptor Area (SQLDA). You can specify *descriptor_area* in C programs as a name or as a pointer reference (*sqldaname) when the SQLDA structure is declared as a pointer. See Appendix B: "SQL Descriptor Area (SQLDA)," for additional details.

ANSI Compliance

The dynamic SQL form of EXECUTE is ANSI SQL-99-compliant.

Authorization

The privileges required depend on the SQL statement and tables accessed.

Rules

The following rules apply to the dynamic SQL form of EXECUTE.

- an SQLDA should be defined.
- The statement specified by *statement_name* must have been previously prepared successfully within the same transaction.
- EXECUTE cannot be used with a dynamic data returning statement, a dynamic macro or with a dynamic multistatement request. For these cases, a dynamic cursor must be declared and the application program should access the results via the FETCH statement.
- The following rules apply to the USING clause, if present:
  - The USING clause identifies variables used as input to the SQL statement by *statement_name*.
  - The host variable name must be a valid client language variable declared prior to the EXECUTE statement that will be used as an input variable. A client structure can be used to identify the input variables.

    The number of variables specified must be the same as the number of parameter markers (the QUESTION MARK character) in the identified statement. The nth variable corresponds to the nth marker.
  - The descriptor name identifies an input SQLDA structure previously defined by the application which contains all necessary information about the input variable or variables.

    The number of variables identified by the SQLD field of the SQLDA must be the same as the number of parameter markers (the QUESTION MARK character) in the identified statement. The nth variable described by the SQLDA corresponds to the nth marker.
- EXECUTE itself cannot be performed as a dynamic SQL statement.

Related Information

See:

- “DESCRIBE” on page 4-55
- “EXECUTE IMMEDIATE” on page 4-61
- “PREPARE” on page 4-63

EXECUTE IMMEDIATE

Purpose

Prepares and executes a dynamic SQL statement.

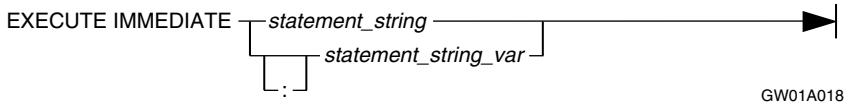
Invocation

Executable.

Embedded SQL only.

Dynamic SQL statement.

Syntax



where:

Syntax element ...	Specifies ...
statement_string	the text of the dynamic SQL statement as a string expression.
statement_string_variable	the text of the dynamic SQL statement as a host variable. The preceding COLON character is strongly recommended.

ANSI Compliance

EXECUTE IMMEDIATE is ANSI SQL-99-compliant.

Authorization

The privileges required depend on the SQL statement and tables accessed.

Usage Notes

The following rules apply to the EXECUTE IMMEDIATE statement:

- Whether specified as a string expression or as a host variable, the dynamic SQL statement can be as long as 32,000 characters.
- If specified as a host variable, the host variable must follow the rules for SQL strings for the client language.

IN this language . . .	*statement_string* is a . . .
COBOL	non-numeric literal.
C	
PL/I	character string expression.

- The dynamic SQL statement must be a single SQL statement; it cannot be a multistatement request.
- Performing an EXECUTE IMMEDIATE is equivalent to performing a PREPARE followed by an EXECUTE of an (unnamed) dynamic single non-macro, non-data returning statement.

  EXECUTE IMMEDIATE simply provides for this special case.
- The dynamic SQL statement cannot:
  - be any of the following specific SQL statements:
    - ABORT
    - BEGIN TRANSACTION
    - CHECKPOINT
    - CLOSE
    - COMMIT
    - CONNECT
    - DESCRIBE
    - ECHO
    - END TRANSACTION
    - EXECUTE
    - EXECUTE IMMEDIATE
    - FETCH
    - LOGOFF
    - LOGON
    - OPEN
    - POSITION
    - PREPARE
    - REWIND
    - ROLLBACK
  - be a data returning statement.

    Such statements require a dynamic cursor when performed dynamically.
  - be a preprocessor declarative.
  - include host variable references.

Related Information

See:

- “DESCRIBE” on page 4-55
- “EXECUTE (Dynamic SQL Form)” on page 4-59
- “PREPARE” on page 4-63

PREPARE

Purpose

Prepares a dynamic SQL statement for execution and assigns a name to it.

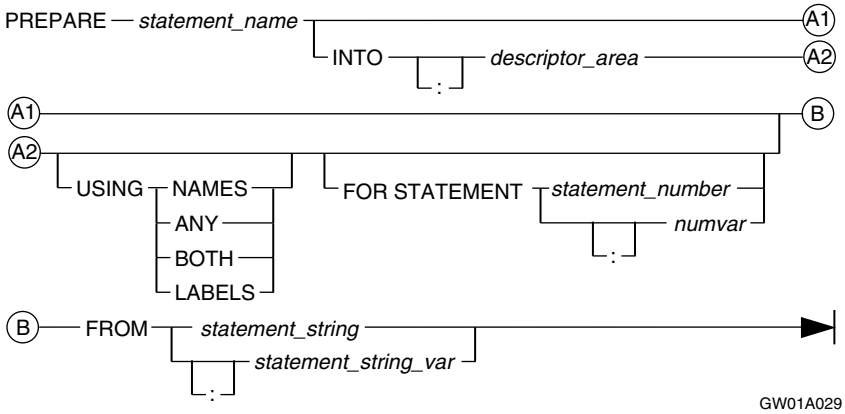
Invocation

Executable.

Embedded SQL only.

Dynamic SQL.

Syntax



where:

Syntax element ...	Specifies ...
statement_name	the name to be associated with the prepared statement. *statement_name* must be a valid SQL identifier and must not be enclosed in quotes.
descriptor_area	specifies the SQLDA to receive the descriptive information about the data returned when the prepared statement is performed. You can specify *descriptor_area* in C programs as a name or as a pointer reference (*sqldaname) when the SQLDA structure is declared as a pointer.
statement_string	the text of the dynamic SQL statement or statements as a string expression.
statement_string_variable	the dynamic SQL statement text as a host variable. The colon before *statement_string_variable* is optional.

Syntax element . . .	Specifies . . .
statement_number	a valid integer numeric literal that identifies the statement number within the request for which the descriptive information is requested
numeric_variable	a host variable conforming to type INTEGER or SMALLINT that represents the statement number in the request for which the descriptive information is requested. The preceding colon is optional.

ANSI Compliance

PREPARE is ANSI SQL-99-compliant.

Authorization

None.

What It Means to Prepare an SQL Statement

To prepare an SQL statement dynamically is to compile it on the fly.

Using the syntax elements defined here, the following table minimally describes the process.

Stage	Process
1	Declare the variable *statement_string* in the client application language.
2	Define *statement_string* as the literal character text of the SQL statement to be performed, still in the client application language.
3	Perform the PREPARE statement from within SQL, defining the host variable *statement_string* as the SQL variable *statement_name*. PREPARE compiles the source code from *statement_name* into executable object code.
4	Perform EXECUTE or EXECUTE IMMEDIATE for *statement_name*.
5	The database software posts return codes to SQLCODE and SQLSTATE.

Rules

The following rules apply to the PREPARE statement:

- an SQLDA should be defined whenever you use dynamic SQL.
- *statement_name* cannot exceed 18 characters.
- Whether specified as a string expression or as a host variable, the dynamic SQL statement text can be as long as 32 kbytes (including SQL, USING data and parcel overhead).
- If specified as a host variable, *host_variable* must follow the rules for SQL strings for the client language. See *Teradata Preprocessor2 Programmer Guide* for details.

IN this language . . .	***statement_string* is a . . .**
COBOL	non-numeric literal.
C	
PL/I	character string expression.

- The dynamic SQL statement text in *statement_string* can:
  - be a single statement or a multistatement request
  - incorporate an EXEC statement
  - incorporate a data returning statement
- The dynamic SQL statement cannot be:
  - any of the following specific SQL statements:
    - ABORT
    - BEGIN TRANSACTION
    - CHECKPOINT
    - CLOSE
    - COMMIT
    - CONNECT
    - DESCRIBE
    - ECHO
    - END TRANSACTION
    - EXECUTE
    - EXECUTE IMMEDIATE
    - FETCH
    - LOGOFF
    - LOGON
    - OPEN
    - POSITION
    - PREPARE
    - REWIND
    - ROLLBACK
  - a preprocessor declarative.

- The dynamic SQL statement can include parameter markers, or placeholder tokens (the QUESTION MARK character), where any literal, particularly a host variable, reference is legal.

  Values are supplied to the statement by means of the USING clause of the OPEN and EXECUTE statements.
- Placeholders are of two types: typed and untyped.
  - A typed placeholder has its data type explicitly cast. For example, part_no is a typed placeholder in the following UPDATE statement.

    ```
    UPDATE parts
    SET part_no = (CAST(? AS INTEGER))
    WHERE vendor_no = ?
    ```

    This action establishes that the data type of the variable at runtime will either be the type cast for the placeholder or one that is convertible to that type.
  - An untyped placeholder is one for which the data type is determined by its context.

    For example, in the following statement, the type of the untyped placeholder in the WHERE clause is the same as that of vendor_no.

    ```
    UPDATE parts
    SET part_no = (CAST(? AS INTEGER)
    WHERE vendor_no = ?
    ```
- Use of placeholders within a CASE expression as the result of a THEN/ELSE clause is valid only when at least one other result in the CASE expression is neither a placeholder nor the NULL keyword.
- The following usage of untyped placeholders is not valid:
  - As the operand of a monadic operator.

    For example, `+ ?` is not valid.
  - As both operands of a dyadic operator.

    For example, `? + ?` is not valid.
  - As both operands of a comparison operator.

    For example, the following SELECT statement is not valid.

    ```
    SELECT *
    FROM table_name
    WHERE ? = ?
    ```

    Note that if you were to replace either placeholder with 0+, the comparison becomes valid because such a value provides enough context to determine that the data type is numeric.

    By extension, it is also true that placeholders cannot be used to denote corresponding fields in a comparison.

    For example, the following comparison is not valid because the first value in each pair has an unknown type that cannot be determined from the context at PREPARE time.

    ```
    (?,X) > (?,Y)
    ```

At the same time, the following comparison is valid because the types can be determined from the context at PREPARE time.

```
(?,X) > (Y,?)
```

- As both operands in a POSITION function.
- As the only operand in an UPPER function.
- As the only operand in a LOWER function.
- As either the second or third operand in a TRIM function.
- As the FROM operand in an EXTRACT function.
- As the first operand in a TRANSLATE function.
- As the argument for any aggregate function.
- As any component of the left hand operand of IS [NOT] NULL.
- As the second component of either operand of an OVERLAPS comparison.
- As solitary select item expressions.

  For example, the following SELECT is not valid.

  ```
  SELECT ? AS alias_name
  FROM table_name
  ```

  Note that the following SELECT statement is valid because the placeholder is used as part of an expression and not as the entire expression in the SELECT list.

  ```
  SELECT 0 + ? AS alias_name
  FROM table_name
  ```

- Executing a PREPARE statement with an INTO clause (and optionally a USING clause) is exactly equivalent to the following:
  - Executing the PREPARE statement without the INTO and USING clauses.
  - Executing a DESCRIBE statement for the prepared statement using the INTO and USING clauses.
- For additional rules for the INTO, USING and FOR STATEMENT clauses of the PREPARE statement, see the description of the DESCRIBE statement.
- PREPARE cannot be performed as a dynamic SQL statement.

Related Information

See:

- “DESCRIBE” on page 4-55
- “EXECUTE (Dynamic SQL Form)” on page 4-59
- “EXECUTE IMMEDIATE” on page 4-61

Chapter 5:

Client-Server Connectivity Statements

This chapter documents the statements used to perform and maintain the connectivity between a client application program performing embedded SQL statements and the Teradata server.

Topics include a brief description of how the Teradata embedded SQL preprocessor makes connections to the Teradata server both during precompilation and at runtime and documentation of the individual SQL statements concerned with connecting and maintaining a client application to a Teradata server.

All SQL statements described in this chapter are for use in embedded SQL applications only. They cannot be used interactively.

The client-server connectivity topics described in this chapter are the following:

- "Connecting a Client Application to the Teradata Server" on page 5-2
- "CONNECT" on page 5-5
- "GET CRASH" on page 5-8
- "LOGOFF" on page 5-9
- "LOGON" on page 5-11
- "SET BUFFERSIZE" on page 5-13
- "SET CHARSET" on page 5-15
- "SET CONNECTION" on page 5-17
- "SET CRASH" on page 5-19

Connecting a Client Application to the Teradata Server

Introduction

The Teradata embedded SQL preprocessor, which runs on a client system, needs to make a connection to the database on the Teradata server. Connections are required both during precompilation and at runtime.

There is no relationship between the preprocessor connection to Teradata made at precompile time and the connection made by an application at runtime. They are separate events.

LOGON and CONNECT statements embedded within the SQL of a host application program have no effect on the preprocessor connection.

Preprocessor Connection

You can run the preprocessor against an application without connecting to Teradata:

IF you specify the SQLCHECK or -sc option as...	THEN the preprocessor...
NOSYNTAX	does not require connection to Teradata.
• FULL (or use FULL as the default) • SQLFLAGGER(ENTRY)	requires connection to Teradata.

You can establish a preprocessor connection to the Teradata server by using the TDPID or -t and USERID or -u preprocessor options.

If you do not provide a user ID and the preprocessor is operating in the IBM mainframe environment, then an implicit connection is attempted.

Runtime Execution Connection

Runtime connection to Teradata is made either explicitly or implicitly.

The TRANSACT or -tr preprocessor transaction mode setting for a session is established when the connection (either explicit or implicit) is made.

The transaction mode is based on the TRANSACT or -tr preprocessor option setting for the application that established the session.

See *Teradata Preprocessor2 Programmer Guide* for additional information about preprocessor invocation options.

Completion Conditions

A successful runtime connection returns the following completion codes:

- SQLCODE = 0
- SQLSTATE = 00000
- SQLCA fields SQLWARN0 and SQLWARN2 = W (Teradata mode only)

Explicit Connections

An application can specify its connection to Teradata explicitly via the CONNECT or the LOGON statement.

Explicit connection permits precise control over which TDP and user ID to connect with, while implicit connection uses system defaults for the TDP and user ID. For this reason, any time you need to connect to a non-default TDP or user ID, you must make an explicit connection.

Explicit connections are preferable because they provide precise control, even when default TDPs and user IDs are sufficient to make a connection.

IF an explicit connection request is made AND...	THEN...
the application is already connected to Teradata	the previous connection is dropped before the new connection is attempted.
a transaction is active	• the connection request is rejected with a SQLCODE of -752. • The application must terminate the current transaction explicitly using one of the following before attempting to issue a new explicit connection request: • COMMIT • ROLLBACK (or ABORT) • LOGOFF

Any explicit connection to the Teradata server requires the following three pieces of information.

- TDP ID
- User ID
- Password

TDP ID and user ID preprocessor options do not affect the application logon at execution time.

User ID security, TDP IDs, and user IDs are described in *Teradata TDP Reference*.

Default TDP ID

If you do not specify a tdpid, then the connection is made using the system default tdpid.

IF your application runs on this platform …	THEN the default TDP is …
IBM mainframe	obtained from the HSHSPB data area module (see *Teradata Call-Level Interface Version2 for Channel-Attached Systems* for details.)
network-attached system	the mtdpid, obtained from the user-defined clispb.dat file or the CLI2SPB data area.

Implicit Connection

If an embedded SQL application running in an IBM mainframe environment submits a SQL request without specifying an explicit connection to Teradata, an implicit connection is attempted based on the job or session under which the application is running.

LAN-attached platforms do *not* permit implicit connections. See *Teradata Call-Level Interface Version2 for Channel-Attached Systems* for details of the implicit connection mechanism.

CONNECT

Purpose

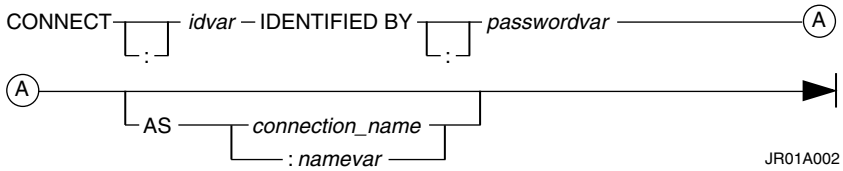
Explicitly connects a client application program to the Teradata server.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax Element . . .	Specifies . . .
idvar	the host variable that contains the Teradata user ID to be used for connection. The user ID is restricted to eight characters.
passwordvar	the host variable that contains the password for the specified user ID. The password is restricted to eight characters. Use of a preceding colon with this variable is optional. The tdpid used for the connection is the system default. You cannot specify an explicit tdpid for CONNECT.
connection_name	the name of the connection.
:namevar	the host variable that contains the connection name. The preceding colon is mandatory.

ANSI Compliance

CONNECT is a Teradata extension to the ANSI SQL-99 standard.

A CONNECT statement is defined in the ANSI SQL-99 standard, but the ANSI form has slightly different syntax.

Authorization

None.

Difference Between CONNECT and LOGON

The difference between CONNECT and LOGON is that LOGON allows specification of any of the possible elements of a Teradata SQL logon string, such as TDP ID and account ID, while CONNECT allows only the user ID and password to be specified.

SQL CONNECT and Preprocessor Connection to the Teradata Server

CONNECT statements have no effect on preprocessor connections to the Teradata server. See *Teradata Preprocessor2 Programmer Guide* for further information.

Why Make an Explicit Connection?

Explicit connection permits you precise control over which TDP and user ID to connect to, while implicit connection uses system defaults for the TDP and user ID. For this reason, any time you need to connect to a non-default TDP or user ID, you must make an explicit connection.

Explicit connections are preferable because they provide precise control, even when default TDPs and user IDs are sufficient to make a connection.

Rules

The following rules apply to the CONNECT statement:

- Use of the CONNECT statement is optional. If the application program executes any SQL statement which requires access to Teradata and the program is not currently connected to Teradata, an implicit connection is attempted.
- If the application program executes a CONNECT statement while already connected to Teradata, the previous connection is dropped.
- Both *idvar* and *passwordvar* must be host variables defined as fixed length character strings of eight characters. (If the user ID or password is less than eight characters long, use pad characters to extend the user ID or password to eight characters).
- CONNECT cannot be performed as a dynamic statement.
- The application program is connected to Teradata using the specified user ID and password.

The following rules apply to the AS (*connection_name* | *:namevar*) clause:

- *connection_name* must be unique (up to 30 bytes) and is case sensitive.
- If the current active connection did not have a connection name, then the next connection must not include a connection name.

  A runtime error is returned indicating the connection attempt has been rejected.

  The current active connection remains unchanged.
- *:namevar* must be a fixed or varying length character variable no longer than 30 bytes.

Related Information

See "LOGON" on page 5-11 for an alternative way to connect to the Teradata server from a client application program.

See *Teradata Preprocessor2 Programmer Guide* for information about how the preprocessor can connect to the Teradata server without using CONNECT or LOGON statements.

GET CRASH

Purpose

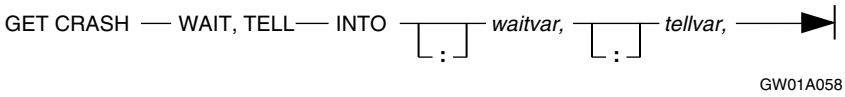
Displays the crash maintenance options established by the SET CRASH statement.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax Element...	Specifies...
waitvar	a one byte character to receive the current wait_across_crash (WAC) option setting value.
tellvar	a one byte character to receive the current tell_about_crash (TAC) option setting value.

ANSI Compliance

GET CRASH is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

GET CRASH Restricted to Workstation Platforms

GET CRASH is valid only for workstation platforms. Mainframe precompilers generate an error if this statement is submitted to the preprocessor.

LOGOFF

Purpose

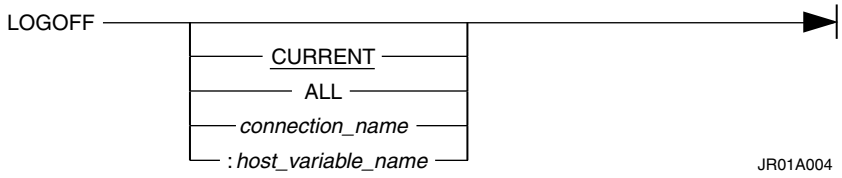
Disconnects an embedded SQL application program from the Teradata server.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax Element...	Specifies...
CURRENT	to log off the current session only.
ALL	to log off all sessions connected with the current user.
connection_name	the name of the connection.
:host_variable_name	the host variable that contains the connection name. The preceding colon is mandatory.

ANSI Compliance

LOGOFF is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Rules

The following rules apply to LOGOFF:

- The LOGOFF statement is optional. If omitted, a disconnect from the Teradata server is implicitly performed when the application program terminates.
- LOGOFF can be used without regard to whether the connection to the Teradata server was established by means of a CONNECT statement, a LOGON statement or an implicit connection.
- If the application is executing in COMMIT mode, LOGOFF causes any outstanding transaction to be committed (see COMMIT above).

  If an application is running in any of the other transaction modes, LOGOFF does not cause outstanding transactions to be committed.
- LOGOFF cannot be performed as a dynamic statement.
- LOGOFF ALL disconnects all active connections.
- LOGOFF CURRENT disconnects the current active connection. (This is the default when no disconnect object is specified.)
- LOGOFF *connection_name* disconnects the named connection. Each connection name must be unique (up to 30 bytes) and is case sensitive.
- LOGOFF *:namevar* disconnects the named connection stored in namevar. The namevar variable must be a fixed or varying length character variable no longer than 30 bytes.

LOGON

Purpose

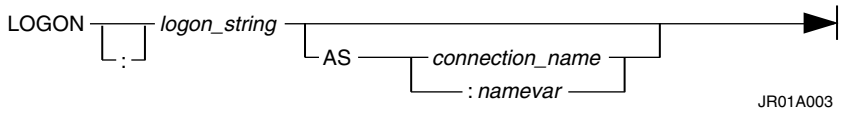
Explicitly connects an embedded SQL application program to the Teradata server.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax Element...	Specifies...
logon_string	the variable containing the logon string to be used.
connection_name	the specified name of the connection.
:namevar	the host variable that contains the connection name. The preceding colon is mandatory.

ANSI Compliance

LOGON is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Difference Between LOGON and CONNECT

The difference between LOGON and CONNECT is that LOGON allows specification of any of the possible elements of a Teradata SQL logon string, such as TDP ID and account ID, while CONNECT allows only the user ID and password to be specified.

Rules

The following rules apply to LOGON:

- LOGON is optional.

  If it is used, LOGON must be the first SQL statement performed by the application.

  If omitted, connection to the Teradata server is made implicitly.
- *logon_st**ring*** entry identifies a host variable that contains the logon string to be used. It must obey the rules for SQL strings for the client language. Use of the colon before the host variable is optional.
- The application program is connected to the Teradata server using the user ID and password, if any, contained in *logon_string*. If either of these is missing, an implicit connection is attempted.
- If *logon_string* contains a TDP ID, it must appear first and must be separated from the rest of the logon string by a slash (/). If present, it determines the TDP used for the connection; otherwise, the installation defined default TDP is used.
- LOGON cannot be performed as a dynamic statement.

The following rules apply to the AS (*connection_name* | *:namevar*) clause:

- The *connection_name* must be unique (up to 30 bytes) and is case sensitive.
- If the current active connection does not have a connection name, then the next connection must not include a connection name. A runtime error is returned indicating the connection attempt has been rejected. The current active connection remains unchanged.
- The *:namevar* variable must be a fixed or varying length character variable no longer than 30 bytes.

Related Information

See "CONNECT" on page 5-5 for an alternative way to connect to the Teradata server from a client application program.

See *Teradata Preprocessor2 Programmer Guide* for information about how the preprocessor can connect to Teradata without using CONNECT or LOGON statements.

SET BUFFERSIZE

Purpose

Specifies the response buffer length to be used when processing an SQL request.

Invocation

Nonexecutable.

Preprocessor declarative.

Embedded SQL only.

Syntax



where:

Syntax Element . . .	Specifies . . .
size	a valid integer numeric literal that defines the response buffer length to be used when processing subsequent SQL requests. The value for *size* must be 0 or an integer in the range 256 to 64K.

ANSI Compliance

SET BUFFERSIZE is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Usage Notes

Preprocessor requests that follow the SET BUFFERSIZE statement sequentially use the value for buffer size specified by the *size* variable. The request buffer size is not affected by this statement

If no SET BUFFERSIZE statement is used, the default response buffer length is used for all requests.

Rules

The following rules apply to SET BUFFERSIZE:

- The value of *size* must be either zero or a valid number within the range of 256 to 65 535. A value of zero specifies the default response buffer length.

On this type of platform . . .	Default buffer size is defined in . . .
IBM mainframe	HSHSPB
workstation	clispb.dat

- If the value of *size* is outside the valid range or is non-numeric, the default response buffer length is used and Teradata displays preprocessor warning message SPP1500.

SET CHARSET

Purpose

Specifies a character set to be used for translation of data to and from the Teradata server at program execution.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax Element . . .	Specifies . . .
set_name	the name of the character set to be used in translating data between the client and the Teradata server.
set_name_var	the name of a host variable that contains the name of the character set to be used in translating data between the client and the Teradata server. Use of the colon is mandatory.

ANSI Compliance

SET CHARSET is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Usage Note

SET CHARSET overrides the default character set used for communication between the application and Teradata at runtime.

Rules

The following rules apply to SET CHARSET:

- Whether specified as *set_name* or as *set_name_var*, the character set name must be a valid Teradata character set identifier.
- If used to identify the character set name rather than the character set code, *set_name* must be enclosed in apostrophes or quotes, based on the APOST/QUOTE preprocessor option setting.
- If used to identify the character set name, *set_name_var* must follow the rules for SQL strings for the client language. See Chapter 2: “Preprocessor Functions,” for details.
- If used to identify the character set code, *set_name_var* must be defined as a small integer host variable.
- Specification of the SET CHARSET statement does not affect preprocessor processing, just the data sent and retrieved from Teradata at execution time.

SET CONNECTION

Purpose

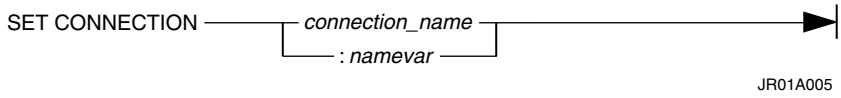
Changes the existing connection to a new connection.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax Element. . . .	Specifies . . .
connection_name	the name of the connection variable to which the current connection is being changed.
:namevar	the host variable that contains the connection name. The preceding colon is mandatory.

ANSI Compliance

SET CONNECTION is ANSI SQL-99-compliant.

Authorization

None.

Rules

The following rules apply to SET CONNECTION:

- SET CONNECTION is not valid in the following cases:
  - in single session mode because the current session does not have a connection name. A runtime error occurs and the current connection remains in effect.
  - within cursor requests specified by the DECLARE CURSOR statement.
  - within dynamic requests specified by the PREPARE or EXECUTE IMMEDIATE statement.
- If the attempted SET CONNECTION fails, then there is no current session unless the current connection does not have a *connection_name*.
- If the current connection is disconnected, then a SET CONNECTION statement must be performed to make a background connection the current one.
- Each connection name must be unique (up to 30 bytes) and is case sensitive.
- The namevar variable must be a fixed or varying length character variable no longer than 30 bytes.

SET CRASH

Purpose

Sets the wait_across_crash (WAC) and tell_about_crash (TAC) options for handling node crashes.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax Element ...	Specifies ...
WAIT_NOTELL	that WAC is to be set to Y and that TAC is to be set to N at runtime. This is the default crash option setting.
NOWAIT_TELL	that WAC is to be set to N and TAC is to be set to Y at runtime.

ANSI Compliance

SET CRASH is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

SET CRASH Is Workstation-Only

SET CRASH is enabled only for workstation platforms. Mainframe precompilers generate an error if this statement is precompiled.

The crash options are in effect for all embedded SQL statements performed after SET CRASH is performed, including LOGON and CONNECT requests, until another SET CRASH is performed.

Preprocessor Behavior When a Node Resets

The following table describes the behavior of the preprocessor when a node resets.

IF the Preprocessor is running on a ...	THEN it ...
Resetting node	aborts. Preprocessing must be restarted after the node resets. This is equivalent to the situation where a utility or application is initiated on an external client that fails.
• Non-Resetting node • LAN-attached client • Channel-attached client	reconnects its session (if connected to a Teradata server) and the current SQL statement undergoing a syntax check returns an error. Restart preprocessing to ensure complete syntax checking.

Application Behavior When a Node Resets

The behavior of embedded SQL applications when a node resets depends on:

- the type of node on which the preprocessor is running
- the crash notification setting

If an embedded SQL application is running on a resetting node, then it aborts and must be restarted after the node resets.

This is equivalent to the situation where a utility or application is initiated on an external client that fails.

Application Behavior When SET CRASH = WAIT_NOTELL

The behavior of embedded SQL applications when a node resets, SET CRASH = WAIT_NOTELL, and the application is running on any of the following environments is explained below.

- Non-resetting node
- LAN-attached client
- Channel-attached client

The application reconnects its session and returns one of the following error codes from the Teradata server and the embedded SQL application takes action appropriate to the error condition:

Code	Description
Error 2825	No record of the last request found after Teradata restart.
Error 2826	Request completed but all output was lost due to Teradata restart.
Error 2828	Request was rolled back during system recovery.
Error 3120	Request aborted because of a Teradata recovery.

Application Behavior When SET CRASH = NOWAIT_TELL

The behavior of embedded SQL applications when a node resets and SET CRASH = NOWAIT_TELL is explained below for the following environments:

- Non-resetting node
- LAN-attached client
- Channel-attached client

The application immediately disconnects the session and the application receives one of the following CLI error codes:

Code	Description
Error 219 (EM_DBC_CRASH_B)	Server connection lost (network or server problem)
Error 220 (EM_DBC_CRASH_A)	

An implicit CLI DISCONNECT request is issued to release any CLI resources tied to the crashed request and session. Any outstanding cursor and dynamic statement resources tied to the crashed session are also released.

Chapter 6:

Multisession Asynchronous Programming With Embedded SQL

This chapter describes the features that support multisession asynchronous programming with embedded SQL in Teradata.

The following topics are described:

- "Multisession Programming With Embedded SQL" on page 6-2
- "Embedded SQL Statements Supporting Multisession Asynchronous Request Programming" on page 6-5
- "ASYNC Statement Modifier" on page 6-6
- "TEST" on page 6-9
- "WAIT" on page 6-13

Multisession Programming With Embedded SQL

Introduction

You can program an embedded SQL application to perform parallel request processing using more than one Teradata session. Such an application can transmit several requests simultaneously, one per each session.

A multisession application is more complicated to implement, debug, and maintain than a single session application, so before you implement multisession programming, you should determine whether multistatement requests on a single session satisfy your throughput and response time requirements.

If you decide the situation calls for multisession programming, then the preprocessor provides facilities to implement multisession applications.

How Multiple Sessions Work

The following table describes how multiple Teradata sessions work.

USE this statement . . .	**TO . . .**
CONNECT or LOGON with an AS *session_id* clause	uniquely name each of the Teradata sessions. This action differentiates the multiple sessions. When more than one session is to be used, the application must name each one explicitly.
SET CONNECTION	switch between each of the named sessions using the unique session identifier specified in the CONNECT or LOGON statements.
LOGOFF *session_id*	disconnect an application from a specific named session.
LOGOFF ALL	disconnect an application from all sessions.

How Asynchronous Requests Work

The following table summarizes how asynchronous requests work.

WHEN you . . .	THEN the session . . .
add an ASYNC clause to the executable SQL request	initiates a uniquely named request. The session returns control to the application without waiting for the completion of the asynchronous request.
use the WAIT statement	waits for the completion of ANY, ALL, or a list of asynchronous requests.
use the TEST statement	tests for the completion of the asynchronous request and returns the results of the request after it has completed.

ASYNC Statement Modifier

Each asynchronous request can be identified to the preprocessor by using the unique asynchronous request identifier specified in the ASYNC statement modifier preceding the executable SQL request.

When the ASYNC modifier is added to the executable SQL request, the request is initiated on the current session and returns control back to the application without waiting for the completion of the asynchronous request.

For more information on ASYNC, see "ASYNC Statement Modifier" on page 6-3.

WAIT Statement

An application program can have requests pending on several sessions simultaneously.

Use the WAIT statement to wait for the completion of ANY, ALL or a list of asynchronous requests, as described below:

- An application can call an asynchronous wait using the WAIT ANY syntax
  The wait ends when any outstanding asynchronous request completes, and returns the session identifier and the asynchronous request identifier.
- An application can wait for all asynchronous requests to complete using the WAIT ALL syntax.
  The wait ends when all outstanding asynchronous requests complete.
- An application can call a synchronous wait using the WAIT *asynchronous_request_id_list* syntax, specifying the asynchronous request identifier of any active asynchronous requests.
  The wait ends when all specified requests complete.

For more information on WAIT, see "WAIT" on page 6-13.

TEST Statement

The TEST statement tests for the completion of an asynchronous request. Once an asynchronous request has completed, TEST is used to retrieve the status of the execution of the asynchronous request.

TEST can also be used to asynchronously test whether an outstanding asynchronous request has completed without having to wait for the request to complete. If the request has not completed, TEST returns an SQL 'not yet complete' message.

TEST can be executed only once against the asynchronous request, and only after the request has completed.

See "TEST" on page 6-9 for more information.

Status Variables and Data Structures for Embedded SQL Applications

Stored procedures and embedded SQL applications use several standardized status variables and data structures to communicate between the application and Teradata.

The following standard host variables that receive completion and exception status codes are described in Chapter 8: "Result Code Variables":

- SQLSTATE (see "SQLSTATE" on page 8-2)
- SQLCODE (see "SQLCODE" on page 8-6)

The ANSI-compliant structure called the SQL Descriptor Area, or SQLDA, is described in Appendix B: "SQL Descriptor Area (SQLDA)."

The Teradata analog of SQLCODE and SQLSTATE, called the SQL Communications Area (SQLCA), is described in Appendix C: "SQL Communications Area (SQLCA).".

The activity count, an enumeration of the number of rows returned by a query, is also useful for many applications. The activity count is reported in the third word in the SQLERRD array for embedded SQL applications and in the status variable declared as ACTIVITY_CODE for stored procedures.

For further information about activity counts, see the following topics:

- "ACTIVITY_COUNT" on page 8-10.
- Appendix D: "SQLSTATE Mappings."

Embedded SQL Statements Supporting Multisession Asynchronous Request Programming

This section describes the embedded SQL statements that support multisession programming:

- ASYNC
- TEST
- WAIT

ASYNC Statement Modifier

Purpose

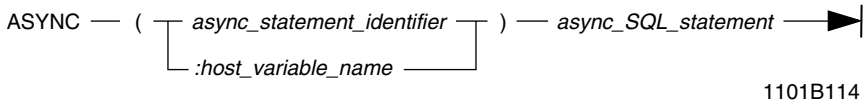
Initiates the asynchronous performance of an executable SQL statement.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax element . . .	Specifies . . .
async_statement_identifier	a unique identifier assigned to the asynchronously performed SQL statement so it can be accessed and its status can be tested and reported by the TEST and WAIT statements.
host_variable_name	a host variable to supply multiple *async_statement_identifier* values. Using a host variable permits a single ASYNC statement to support multiple asynchronous sessions simultaneously.
async_SQL_statement	the executable SQL statement. *async_SQL_statement* can be passed to ASYNC indirectly through dynamic SQL using a host variable.

ANSI Compliance

The ASYNC clause is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Rules

The following rules apply to the ASYNC modifier:

- Only one asynchronous statement can execute per connection.

  Before another statement can be processed asynchronously on a connection, the previous asynchronous statement must have completed; otherwise, a runtime error occurs.
- Each *async_statement_identifier* must be unique (up to 30 bytes) across all active connections and is case sensitive.
- ASYNC is not valid within cursor requests specified by the DECLARE CURSOR statement.
- ASYNC is not valid within dynamic requests specified by the PREPARE or EXECUTE IMMEDIATE statement; however, you can use dynamic SQL to pass an *async_SQL_statement* to ASYNC indirectly through a host variable (see "Example 4" on page 6-8).
- ASYNC is not valid with any embedded SQL declarative statements:
  - BEGIN DECLARE SECTION
  - DECLARE CURSOR
  - DECLARE STATEMENT
  - DECLARE TABLE
  - END DECLARE SECTION
  - INCLUDE
  - INCLUDE SQLCA
  - INCLUDE SQLDA
  - WHENEVER
- ASYNC is not valid with the following executable embedded SQL statements:
  - ABORT
  - BEGIN TRANSACTION
  - COMMIT
  - CONNECT
  - DATABASE
  - DESCRIBE
  - END TRANSACTION
  - FETCH
  - GET CRASH
  - LOGOFF
  - LOGON
  - POSITION
  - REWIND
  - ROLLBACK
  - SET BUFFERSIZE
  - SET CHARSET
  - SET CRASH

Example 1

This example submits an asynchronous request to open a cursor.

```
ASYNC (request_1) OPEN cursor_1
```

Example 2

This example submits an asynchronous request to perform a searched update of a table.

```
ASYNC (request_1) UPDATE table_1
SET A = :A
```

Example 3

This example submits an asynchronous request to perform a macro.

```
ASYNC (request_1) EXEC macro_1
```

Example 4

This example uses dynamic SQL to pass the asynchronous SQL statement to ASYNC through a host variable.

```
strcpy (SQL_STATEMENT.arr,"DELETE FROM TABLE1 WHERE FIELD1 = ?");
       SQL_STATEMENT.len = strlen (SQL_STATEMENT.arr);

EXEC SQL PREPARE S1 FROM :SQL_STATEMENT;

EXEC SQL ASYNC (stmt01) EXECUTE S1 USING :VAR1;
```

Related Information

See "TEST" on page 6-9 for more information about testing the completion status of an asynchronous request.

See "WAIT" on page 6-13 for more information about waiting for an asynchronous request to complete.

See "Dynamic SQL" on page 4-52 for more information about dynamic SQL and the descriptions of "EXECUTE (Dynamic SQL Form)" on page 4-59 and "PREPARE" on page 4-63 for information about how to prepare and execute an SQL statement dynamically.

TEST

Purpose

Tests the completion status of the asynchronous SQL statement identified by *async_statement_identifier.*

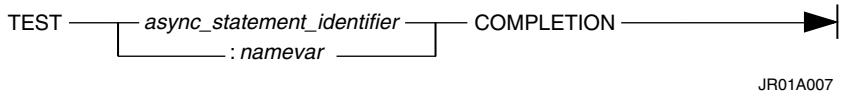
When used with the WAIT statement, returns the completion status of the asynchronous SQL statement identified by *async_statement_identifier* or by *host_variable_name*, but does not wait if the request has not completed.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax element . . .	Specifies . . .
async_statement_identifier	a unique, application-supplied identifier for an asynchronously performed SQL statement assigned by the ASYNC modifier.
host_variable_name	the host variable that contains the connection request ID name. The preceding colon is mandatory.

ANSI Compliance

TEST is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Rules

The following rules apply to TEST:

- Each *async_statement_identifier assignment* is case sensitive and must be unique across all active connections.
- The maximum length of each *async_statement_identifier* is 30 bytes.
- The value for *host_variable_name* must be a fixed or varying length character variable no longer than 30 bytes.
- If there is no outstanding asynchronous SQL statement, then the exception condition “no outstanding asynchronous SQL statement” is raised.

  SQLCODE is set to -650.

  SQLSTATE is set to 04000.
- If the specified asynchronous SQL statement has not completed, then the exception condition ”SQL statement not yet complete” is raised.

  SQLCODE is set to -651.

  SQLSTATE is set to 03000.
- If the specified asynchronous SQL statement has completed, then the following things happen:

  1 The runtime finishes processing the request and returns the completion status via the SQLCODE or the SQLSTATE.

  2 The SQL statement named by *async_statement_identifier* can no longer be referenced.
- TEST is not permitted within the following request types:
  - Cursor requests specified by the DECLARE CURSOR statement.
  - Dynamic requests specified by the PREPARE or EXECUTE IMMEDIATE statement.

Example 1

This example tests the statement identified by the name ‘REQ_1’ for completion and returns the appropriate exception or completion code to SQLCODE or SQLSTATE.

The name ‘REQ_1’ is defined by the ASYNC clause using the *async_statement_modifier* variable.

```
TEST REQ_1 COMPLETION
```

Example 2

This example tests the statement identified by the host variable 'REQID_VAR' for completion and returns the appropriate exception or completion code to SQLCODE or SQLSTATE.

The name contained within REQID_VAR is defined by the ASYNC clause using the *async_statement_modifier* variable.

```
TEST :REQID_VAR COMPLETION
```

Example 3

This example uses TEST with WAIT. The program asynchronously spawns 2 update requests, REQ_1 and REQ_2, respectively, then waits until both REQ_1 and REQ_2 have completed before proceeding.

TEST statements monitor SQLCODE to determine when both REQ_1 and REQ_2 have returned successful completion codes (SQLCODE = 0) before continuing with the rest of the main program.

If either request has not completed (SQLCODE = -651), then the wait continues.

When both statements have completed, then the main program continues processing.

The non-SQL statements are pseudocode to indicate crudely a general idea of how the WAIT and TEST SQL statements might fit into a host main program.

```
…
     EXEC-SQL
       ASYNC REQ_1
       UPDATE TABLE_A
       SET A = :A;
     EXEC-SQL
       ASYNC REQ_2
       UPDATE TABLE_B
       SET B = :B;
…
100 EXEC-SQL
       WAIT REQ_1, REQ_2 COMPLETION;
…
     EXEC-SQL
       TEST REQ_1 COMPLETION;
    IF SQLCODE = -651 THEN GOTO 100
    IF SQLCODE = 0 THEN CONTINUE
     EXEC-SQL
       TEST REQ_2 COMPLETION;
    IF SQLCODE = -651 THEN GOTO 100
    IF SQLCODE = 0 THEN CONTINUE
…
```

Related Information

See "ASYNC Statement Modifier" on page 6-6 for information about how to submit an asynchronous request.

See "WAIT" on page 6-13 for information about how to wait on *and* test the status of an asynchronous request.

WAIT

Purpose

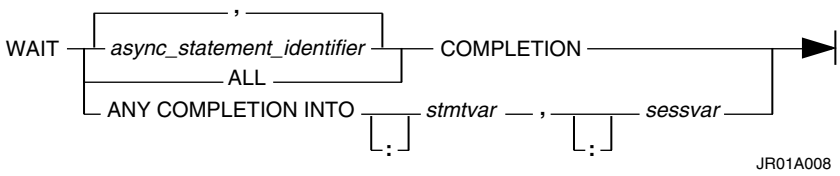
Pauses execution of the invoking program and waits for the completion of one or more asynchronous SQL statements.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax element …	Specifies …
async_statement_identifier	the unique identifier for an asynchronously performed SQL statement assigned by the ASYNC modifier.
host_variable_name	the name of the host variable that contains a value to be passed to the WAIT statement as an *async_statement_identifier*. This permits an application to supply multiple values of an *async_statement_identifier* to WAIT by means of a host variable.
ALL	to pause execution for all current asynchronously performed SQL statements.
async_statement_variable	the name of the host variable into which the asynchronous statement identifier for the completed request is to be written.
session_variable	the name of the host variable into which the ID of the session in which *async_statement_variable* completed is to be written.

ANSI Compliance

WAIT is a Teradata extension to the ANSI SQL-99 standard.

Authorization

None.

Rules

The following rules apply to WAIT:

- Each *async_statement_identifier* must be unique (up to 30 bytes) across all active connections and is case sensitive.
- If there is no outstanding asynchronous SQL statement, then the exception condition “no outstanding asynchronous SQL statement” is raised.
- If ALL is specified, then the WAIT statement returns when all asynchronous statements have completed.
- If ANY is specified, then the WAIT statement returns when any of the outstanding asynchronous statements have completed.

  The asynchronous statement identifier is returned to the host variable *async_statement_variable* in the INTO clause, and the session identifier is returned to the host variable *session_variable*.

  The host variables *async_statement_variable* and *session_variable* must be defined as a fixed or varying length character variable no longer than 30 bytes.
- WAIT is not valid within the following request types:
  - Cursor requests specified by the DECLARE CURSOR statement.
  - Dynamic requests specified by the PREPARE or EXECUTE IMMEDIATE statement.

Example 1

The following example shows a basic WAIT statement. WAIT returns control to the program when the SQL request named REQ_1 completes.

```
WAIT REQ_1 COMPLETION
```

Example 2

The following example shows a more complicated WAIT statement that specifies 2 asynchronous statement identifiers. WAIT returns control to the program when both REQ_1 and REQ_2 complete.

```
WAIT REQ_1, REQ_2 COMPLETION
```

Example 3

The following example waits on all outstanding asynchronous SQL statements and returns control to the program when they have all completed.

```
WAIT ALL COMPLETION
```

Example 4

The following example waits on any outstanding asynchronous SQL request to complete and returns control to the program when any one of them completes, returning the value for the completed asynchronous statement identifier to REQID_VAR and the value for the ID of the session in which it ran to completion to SESSID_VAR.

```
WAIT COMPLETION INTO :REQID_VAR, :SESSID_VAR
```

Related Information

See "ASYNC Statement Modifier" on page 6-6 for information about how to submit an asynchronous request.

See "TEST" on page 6-9 for information about how to test the status of an asynchronous request without waiting for it to complete.

Section 3:

Constructs Common to Embedded SQL and Stored Procedures

Section Contents

Chapter 7:

Cursors and Cursor Control Statements

This chapter describes SQL cursors, their declaration, and their control by embedded SQL applications and stored procedures.

The following topics are described in the chapter:

- Cursors
- Updatable Cursors
- "CLOSE" on page 7-15
- "DECLARE CURSOR" on page 7-17
- "DECLARE CURSOR (Dynamic SQL Form)" on page 7-19
- "DECLARE CURSOR (Macro Form)" on page 7-21
- "DECLARE CURSOR (Request Form)" on page 7-23
- "DECLARE CURSOR (Selection Form)" on page 7-26
- "DECLARE CURSOR (Stored Procedures Form)" on page 7-29
- "FETCH (Embedded SQL Form)" on page 7-33
- "FETCH (Stored Procedures Form)" on page 7-37
- "OPEN (Embedded SQL Form)" on page 7-42
- "OPEN (Stored Procedures Form)" on page 7-45
- "POSITION" on page 7-47
- "REWIND" on page 7-49

Cursors

Definition

A cursor is a structure used at runtime by the SQL preprocessor and by stored procedures to point to the result rows in a response set returned by an SQL query.

Cursors are only applicable to the following SQL application types:

- Client applications written in C, COBOL, or PL/I that contain embedded SQL statements and have been precompiled using the Teradata SQL preprocessor
- Stored procedures

The syntax of various cursor control statements varies depending on whether they are used for embedded SQL or for stored procedures. Variant syntax forms are documented with the individual cursor control statements. Several cursor control statements are valid only with embedded SQL.

Cursors are not valid for sessions conducted interactively from a terminal using a query manager like BTEQ.

Why Cursors Are Necessary

An embedded or stored procedure SQL SELECT statement is permitted to retrieve at most one row of data at a time. It is an error for a SELECT statement to retrieve more than one row of data in embedded SQL or within a stored procedure.

Without knowing the number of rows to be retrieved from a request, it is impossible to know the number of host variables required to hold the results of the SELECT. Thus, only a single result row is allowed.

This is not a problem for so-called singleton SELECTs: SELECT statements that are coded to return only one row. However, SQL queries frequently return multiple rows in the form of a result table or response set. This situation is one that typical programming languages are not equipped to handle.

Another way of expressing this is to say that traditional programming languages such as COBOL, C, and PL/I are record-oriented, while relational databases are inherently set-oriented.

Because of this mismatch, which is sometimes referred to as an impedance mismatch, some mechanism must be used to enable a record-oriented language to process set-oriented data.

This mechanism is the cursor. You can think of a cursor as a pointer to a single data row in a result table.

Cursors use SQL statements unique to embedded SQL and stored procedures to step through the result table, which is stored as a spool file, one row at a time.

Types of Cursors

Cursors can be classified in several different ways.

- Dynamic
- Macro
- Request
- Selection
- Stored Procedure
- Positioned (updatable)
- Non-positioned

The first five types refer to ways a cursor can be declared in a DECLARE CURSOR statement. Of these, only Stored Procedure-type cursors are supported in stored procedures.

The last two types refer to ways a cursor can be used to manipulate data. Both are supported in embedded SQL and stored procedures.

Cursor States and Positions

The state of a cursor at any given time is either open or closed.

If open, a cursor can point to a returned row from the result table returned by its associated SELECT. This row is said to be the current row.

While the relational model does not support the concept of ordered rows, the mechanism of processing a result set one row at a time necessitates a redefinition for this situation only.

An open cursor can be in one of three possible positions:

- Before a row

  The cursor is positioned before the first row if the cursor was OPENed but no rows were FETCHed.
- On a row

  The cursor is on a row following a FETCH of the row.

  When a cursor is pointing at a row, that row is referred to as the current row of the cursor.
- After the last row in a result set.

  When there are no further rows in the result set to FETCH, the cursor points immediately after the last row in the retrieved set.

Cursors are also used to manage inserts, updates, execution of multistatement requests and SQL macros.

How Cursors Are Incremented

Cursors are incremented using a FETCH statement as needed to step through the response set rows.

Stored procedure cursors have some additional facilities that enable more flexibility than embedded SQL cursors possess. The following bullets list these additional facilities:

- The FIRST and NEXT options of the FETCH statement and the SCROLL and NO SCROLL options of the DECLARE CURSOR declaration enable cursors to either scroll forward to the next row in the spool or to scroll directly to the first row in the spool.
- Within a FOR loop, a cursor moves to the next row with each iteration of the loop.

Cursor Rules for Embedded SQL

The following rules apply to the use of cursors in an embedded SQL application program:

- No more than 16 cursors can be open at any one time in a given application.
- Once an application has 16 cursors open, it can only issue one of the following as the next statement:
  - CLOSE
  - COMMIT (if in COMMIT mode)
  - FETCH
  - LOGOFF
  - POSITION
  - REWIND
- Cursor and dynamic statement identifiers are constructed from the following list of valid characters:

    - uppercase letters
    - lowercase letters
    - $
    - @
    - #
    - digits
    - underscores
- Cursor and dynamic statement identifiers must begin with a national character and cannot exceed 18 characters.
- A cursor and dynamic statement identifier cannot be an SQL keyword.
- For purposes of comparison between identifiers, the case of letters is not significant.

  The preprocessor accepts statements in uppercase, lowercase or mixed case.
- To support multibyte character sets, cursor and dynamic statement names can have multibyte characters, and they can be expressed in internal hexadecimal notation.

Cursor Rules for Stored Procedures

The following rules apply to the use of cursors in a stored procedure:

- No more than 15 cursors can be open at any one time in a given stored procedure.
- Cursors can be defined in two ways:
    - In a DECLARE CURSOR declaration.
    - In a FOR loop control statement.
- The scope of a cursor declared with DECLARE CURSOR is the compound statement and its nested compound statements, if any.
- The scope of column name or its correlation name in a FOR loop cursor is restricted to the body of the FOR statement.
- Cursor names are constructed from the following list of valid characters:
    - uppercase letters
    - lowercase letters
    - $
    - @
    - #
    - digits
    - underscores
- The FOR *cursor_name* statement opens a cursor for the SELECT statement specified as the cursor specification.

- For a cursor defined in a FOR loop control statement, open cursor, fetch cursor, and close cursor operations are performed implicitly as part of the execution of the FOR loop.

  The next row, if it exists, is fetched for an open cursor with each iteration of the FOR statement.
- For a cursor defined by a DECLARE CURSOR declaration, open cursor, fetch cursor, and close cursor operations must be specified explicitly using OPEN, FETCH, and CLOSE statements.
- The OPEN *cursor_name* statement opens a cursor for the SELECT statement specified as the cursor specification.
- The FOR *cursor_name* loop control statement implicitly opens a cursor for the SELECT statement specified as the cursor specification.
- To create a positioned cursor, specify a FOR UPDATE clause in the cursor specification of the DECLARE CURSOR declaration.

Cursor Support Statements in the SQL Preprocessor

The following list explains how the various SQL statements that support cursors fit into a coherent whole in embedded SQL.

- DECLARE *cursor_name* CURSOR FOR *data_returning_statement* associates a cursor name with a multirow data returning statement. You do not need to use a cursor to process a singleton SELECT.

  You can then use the following statements to manipulate the declared cursor.
- OPEN *cursor_name* performs the request (or requests) defined by the DECLARE CURSOR statement.
- FETCH *cursor_name* INTO uses the opened cursor to retrieve successive individual rows from the result set into host variables, using host language statements to increment the cursor based on a WHENEVER statement or on testing the value of status codes returned to SQLCODE after each FETCH.
- DELETE … WHERE CURRENT OF *cursor_name* deletes the currently fetched row from its base table.
- UPDATE … WHERE CURRENT OF *cursor_name* updates the currently fetched row.
- POSITION *cursor_name* moves the cursor either forward or backward to the first row of the specified statement.

  A REWIND statement specifying the cursor name moves the cursor to the first row of the first (or only) statement of a request.
- CLOSE *cursor_name* closes the open *cursor_name* and terminates the data returning statement specified by the DECLARE CURSOR statement.

The following table summarizes the actions taken by cursors and related statements describes the outcomes of those actions.

Action	SQL Statement	Result
Define a statement or request to be associated with a cursor.	DECLARE CURSOR	Defines the association between a cursor and an SQL data returning statement.
Open a cursor.	OPEN	Performs the SQL data returning statement defined in DECLARE CURSOR.
Retrieve the next row in the result table.	FETCH	Retrieves a row from the result table.
Move the cursor to the first row of a specific SQL statement.	POSITION REWIND	Positions the cursor to the first row of the result table of the named statement.
Update a row.	UPDATE … WHERE CURRENT OF	Changes the contents of the current row.
Delete a row.	DELETE … WHERE CURRENT OF	Removes the current row from the table.
Close the cursor.	CLOSE	Terminates the retrieval process.

Cursor Support in Stored Procedures

Support is somewhat different depending on whether a cursor is opened by a FOR loop statement or by a cursor declared by a DECLARE CURSOR statement:

- The following dummy iteration statement opens a cursor for the specified cursor.

  ```
  FOR for_loop_variable AS [cursor_name CURSOR FOR]
  cursor_specification DO statement END FOR;
  ```

  where *cursor_specification* is a single SELECT statement and *statement* can be one or more SQL control or DML statements.

  The FOR statement performs as follows:

  1. Fetches one row of data from the result set into the *for_loop_variable* on each iteration .
  2. Increments the cursor on each iteration, fetching the next row of data, if it exists.

The WHERE CURRENT OF forms of DELETE and UPDATE perform as follows:

- DELETE … WHERE CURRENT OF *cursor_name* deletes the currently fetched row from its base table.
- UPDATE … WHERE CURRENT OF *cursor_name* updates the currently fetched row in its base table.

- For cursors defined by a DECLARE CURSOR statement, you must submit explicit OPEN cursor_name and FETCH cursor_name statements.

For more details on the use of cursors in stored procedures, see "FOR" on page 9-111.

Transactions and Cursors

The following bullets explain how the various SQL statements that terminate transactions process cursors.

- COMMIT terminates all open cursors and commits the changes made by the cursors while a transaction was in progress. (ANSI mode only)
- ROLLBACK (ANSI and Teradata modes) or ABORT (Teradata mode only) terminates all open cursors within the current transaction and discards any changes made by the cursors while the transaction was in progress.
- END TRANSACTION terminates all open cursors within the current transaction and commits any changes made by the cursors while the transaction was in progress (Teradata mode only).

Positioned Cursors

Introduction

The ANSI SQL standard defines an updatable, or positioned, cursor. This means that an application can define a cursor for a query and then update results rows using the same cursor.

The ANSI SQL-99-standard DELETE and UPDATE statements do not identify a search condition. Instead, they identify a cursor with a row to be updated or deleted.

Process Flow for Updating a Row Using a Cursor in the SQL Preprocessor

The following table shows the general process flow for updating a row using a cursor in the SQL Preprocessor:

Stage	Process
1	Declare a cursor for a SELECT statement.
2	Open the cursor using an OPEN statement.
3	Retrieve a row using the FETCH statement.
4	Update or delete the fetched row using the WHERE CURRENT OF clause with an UPDATE or DELETE statement, respectively.
5	Close the cursor using a CLOSE statement.

Process Flows for Using a Positioned Cursor to Update or Delete a Row in a Stored Procedure

The following table shows the general process flow for updating or deleting a row using a FOR loop cursor in a stored procedure:

Stage	Process
1	Specify a FOR statement with the appropriate cursor specification.
2	Fetch one row with each iteration of the FOR statement. The cursor then points to the next row in the results set.
3	Update or delete the fetched row using the WHERE CURRENT OF clause with an UPDATE or DELETE statement, respectively.
4	Continue the FOR iteration loop until the last row is fetched.
5	Close the cursor by terminating the FOR statement.

The following table shows the general process flow for updating or deleting a row using a DECLARE CURSOR-style cursor in a stored procedure:

Stage	Process
1	Specify a DECLARE CURSOR statement with the appropriate cursor specification.
2	Open the cursor by performing an OPEN *cursor_name* statement.
3	Perform FETCH statements to fetch one row at a time from the response set. The next cursor movement depends on the scrollability option you specify.
4	Update or delete the fetched row using the WHERE CURRENT OF clause with an UPDATE or DELETE statement, respectively.
5	Close the cursor by performing a CLOSE *cursor_name* statement.

IF the fetch option specified is …	THEN the cursor moves to the …
FIRST	first row in the result set.
NEXT	next row in the result set.

Features Supporting Positioned Cursors

Several features enable ANSI SQL-99-standard positioned cursor functionality, including:

- WHERE CURRENT OF clause in DELETE and UPDATE statements.
- FOR CHECKSUM clause of the LOCKING modifier (embedded SQL only).
- FOR UPDATE clause in SELECT statements (stored procedures only).

WHERE CURRENT OF Clause

Once a cursor declaration is defined as updatable, the DELETE and UPDATE statements must have some mechanism for acting on the row pointed to by the cursor. The WHERE CURRENT OF clause performs that function.

FOR CHECKSUM Clause

Updatable cursors do not recognize resource locking levels. Instead, they assume that all actions involving a cursor are done within a single transaction and that terminating that transaction closes any open cursors.

To this functionality, Teradata adds the FOR CHECKSUM clause of the LOCKING modifier, a Teradata extension to the ANSI SQL-99 standard.

When no LOCKING modifier is specified, all SELECT statements use a READ level lock. Positioned updates and deletes default to a WRITE level row hash lock.

The FOR CHECKSUM clause is not supported for stored procedures.

How CHECKSUM Locking Works

CHECKSUM locking is similar to ACCESS locking, but it adds checksums to the rows of a results table to allow a test of whether a row in the cursor has been modified by another user or session at the time an update is being made through the cursor.

If an application specifies an ACCESS lock and then issues a cursor UPDATE or DELETE, the row to be changed might have been altered by another application between the time the first application read the row and the time it issued the cursor UPDATE or DELETE statement.

If the checksum changes because another application has updated the row since it was last read by the current application, the current application receives an error.

An error is returned to the application whenever any of the following requirements for CHECKSUM locking are not met:

- The object locked must be a table
- The LOCKING modifier must be followed by an updatable cursor SELECT
- The table specified in the LOCKING modifier must be the same as the table referenced in the FROM clause of the SELECT statement that follows it

CHECKSUM locks are valid only when used with a SELECT statement opened by an updatable cursor.

Example: LOCKING with CHECKSUM

This example uses LOCKING with CHECKSUM on the table named T.

```
LOCKING TABLE T
FOR CHECKSUM
SELECT I, TEXT
FROM T;
```

Rules for Positioned Cursors

Positioned cursors must observe a specific set of rules that constrain the options of the SELECT statement to which they apply.

The rules for defining a positioned cursor are as follows:

- Positioned UPDATEs and DELETEs must be in the same transaction as the SELECT that opened the cursor they are using.
- The following items are not updatable:
  - Dynamic cursors
  - Multistatement requests
- The following items are not allowed in an SQL statement controlled by an updatable cursor:
  - Tables with triggers defined on them
  - Joins between multiple base tables
  - DISTINCT keyword
  - GROUP BY clause
  - HAVING clause
  - WITH clause
  - ORDER BY clause
  - Aggregate operators:
  - Set operators
  - Correlated subqueries
  - Select lists having any of the following:
    - duplicate column names
    - expressions
    - functions
- For positioned cursors in a stored procedure, the Parser determines whether a cursor is updatable or not as follows:

IF the SELECT statement specified by a cursor declaration ...	THEN the cursor is ...
specifies a FOR UPDATE clause	updatable.
does not specify a FOR UPDATE clause	not updatable. The SELECT statement is still processed, but Teradata returns a warning message stating that the cursor is not updatable.

- Multiple updates of a currently fetched row or updates followed by a delete of the currently fetched row are allowed.
- Positioned updates and deletes must occur within the same transaction that contains the SELECT statement that defined the cursor.

- When the application attempts a positioned UPDATE or DELETE against a row selected by a nonvalid SELECT, Teradata returns an error and rolls back the affected transaction.
- When a program attempts to UPDATE or DELETE a row using a WHERE CURRENT OF clause for a cursor that is not updatable, Teradata returns an error stating that the cursor is not updatable.
-

FOR stored procedures created under this mode ...	Positioned cursors are ...
ANSI	valid
Teradata	not valid

Usage Guidelines for Positioned Cursors

The following guidelines describe ways to optimize the performance of positioned cursors.

- Because of the number of row hash locks required to implement cursor updates on large sets of data, you should use positioned updates and deletes for small sets of data only. When too many row hash locks are imposed, the transaction fails and aborts with a lock table overflow error.

  Either avoid long duration transactions when using updatable cursors or use CHECKSUM locking[1] to avoid locking conflicts that might prevent other applications from reading or updating the tables used by your application.

  When the number of row hash locks becomes excessive and a lock table overflow error occurs, Teradata issues a transaction level abort for any SQL application that receives the error.

1 CHECKSUM locking is not supported for stored procedures.

- Cursor conflicts can occur within a single transaction. Such a conflict occurs when any of the following conditions exist:
  - There are two cursors opened on the same table at the same time within a transaction and one of the cursors attempts a positioned update or delete on a row in the table that is currently the subject of a positioned update or delete request by the other cursor.
  - A cursor is opened on a table, a searched update or delete is made on that table, and then the cursor attempts to perform a positioned update or delete on the table.
  - A cursor is opened on a table, a positioned update or delete is made through the cursor, and then a searched update or delete is attempted on the same table.

  All three situations return a cursor conflict warning to the user but perform the requested deletion or update.

Further Information

See *Teradata Preprocessor2 Programmer Guide* for examples of implementing updatable cursors in an embedded SQL client application.

See "FOR" on page 9-111 for examples of implementing updatable cursors in stored procedures.

CLOSE

Purpose

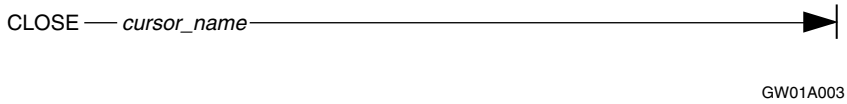
Closes an open cursor and releases the resources held by the cursor while it was open.

Invocation

Executable.

Embedded SQL and stored procedures only.

Syntax



GW01A003

where:

Syntax element …	Specifies …
cursor_name	the name of an open cursor to be closed.

ANSI Compliance

CLOSE is ANSI SQL-99-compliant.

Authorization

None.

Usage Notes

The following rules apply to CLOSE:

- The cursor identified by *cursor_name* must have been previously declared.
- The cursor identified by *cursor_name* must be open.
  If *cursor_name* is not open at the time CLOSE is submitted within a stored procedure, the following run time exception occurs:
  - SQLCODE is set to 7631
  - SQLSTATE is set to '24501'
- When control passes from a compound stored procedure statement, all open cursors declared within the body of that compound statement are closed implicitly.
- CLOSE cannot be performed as a dynamic SQL statement.

The COMMIT and ROLLBACK statements also close open cursors (see "COMMIT" on page 3-33 and "ROLLBACK" on page 3-141).

Example 1

The following stored procedure example is valid because the cursor identified by cursor name ProjCursor is open at the time it is closed with a CLOSE statement:

```
CREATE PROCEDURE sp1 (OUT Par1 INTEGER, OUT Par2 CHAR(30))
BEGIN
  DECLARE ProjCursor CURSOR FOR
    SELECT *
    FROM Project
    ORDER BY ProjId;
  OPEN ProjCursor;
  Label1:
  LOOP:
    FETCH ProjCursor INTO Par1, Par2;
    IF (SQLSTATE = '02000') THEN
      LEAVE Label1;
    END IF;
  END LOOP Label1;
  CLOSE ProjCursor;
END;
```

Example 2

In the following example, ProjCursor is closed explicitly with the CLOSE statement. EmpCursor is opened and no CLOSE statement is specified. In this case, EmpCursor is closed implicitly when the stored procedure terminates.

```
CREATE PROCEDURE sp1 (IN Par1 CHAR(5))
BEGIN
  DECLARE ProjCursor CURSOR FOR
    SELECT *
    FROM Project
    ORDER BY ProjId;
  DECLARE EmpCursor CURSOR FOR
    SELECT *
    FROM Employee
    WHERE Dept_Code = Par1;
  OPEN ProjCursor;
  OPEN EmpCursor;
  CLOSE ProjCursor;
END;
```

Related Information

See "OPEN (Embedded SQL Form)" on page 7-42.

DECLARE CURSOR

Purpose

Defines and assigns a name to a cursor.

Invocation

Nonexecutable.

Preprocessor declaration.

Embedded SQL and stored procedures only.

Usage Notes

Five forms of DECLARE CURSOR are defined as follows:

- The dynamic SQL form of DECLARE CURSOR (see "DECLARE CURSOR (Dynamic SQL Form)" on page 7-19) associates a cursor with a dynamic SQL statement

  The dynamic SQL statement can be any of the following:

  - a data returning statement
  - a Teradata SQL macro
  - an arbitrary request containing any combination of supported statements, including those listed in the previous 2 bullets
  - a Teradata stored procedure
- The macro form of DECLARE CURSOR (see "DECLARE CURSOR (Macro Form)" on page 7-21) associates a cursor with a Teradata SQL macro.
- The request form of DECLARE CURSOR (see "DECLARE CURSOR (Request Form)" on page 7-23) associates a cursor with an arbitrary Teradata SQL request, typically a multistatement request specified within an SQL string literal.
- The selection form of DECLARE CURSOR (see "DECLARE CURSOR (Selection Form)" on page 7-26) associates a cursor with a SELECT or other data returning statement.
- The stored procedures form of DECLARE CURSOR (see "DECLARE CURSOR (Stored Procedures Form)" on page 7-29) associates a cursor with a SELECT or other data returning statement within the body of a stored procedure FOR statement.

The following rules apply to all forms of the DECLARE CURSOR statement:

- Each cursor declaration within a client program must specify a different cursor name.
- The cursor declaration for a particular cursor name must precede any references to that cursor name in other embedded SQL or stored procedure statements.
- In COBOL, the DECLARE CURSOR statement can be specified either in the DATA DIVISION or in the PROCEDURE DIVISION.
- A cursor name cannot exceed 18 characters.

DECLARE CURSOR (Dynamic SQL Form)

Purpose

Defines and assigns a name to a cursor for a prepared dynamic SQL statement.

Invocation

Nonexecutable.

Preprocessor declaration.

Embedded SQL only.

Syntax



GW01A012

where:

Syntax element . . .	Specifies . . .
cursor_name	any valid SQL identifier.
statement_name	the name associated with a previously prepared statement.

ANSI Compliance

The dynamic SQL form of DECLARE CURSOR is ANSI SQL-99-compliant.

Authorization

None.

Usage Notes

The prepared dynamic SQL statement can be any of the following:

- A single, non-data-returning, non-macro statement.
- A single SELECT statement (which must be specified without an INTO clause).
- A single EXEC macro_name statement.
- A multi-statement request, which can include any of the foregoing statements.

The following rules apply to the Dynamic DECLARE CURSOR statement:

- Before the dynamic cursor can be opened, you must have previously prepared the statement specified by *statement_name* within the same transaction.
- You can declare only one dynamic cursor for a given *statement_name*.

Example

Dynamic DECLARE CURSOR statements take the following form:

```
DECLARE Ex CURSOR FOR DynStmt7
```

DECLARE CURSOR (Macro Form)

Purpose

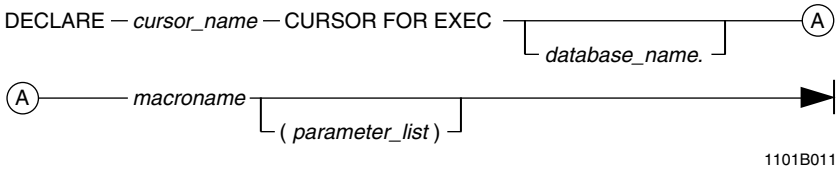
Defines and assigns a name to a macro cursor.

Invocation

Nonexecutable.

Preprocessor and stored procedure declaration.

Syntax



where:

Syntax element ...	Specifies ...
cursor_name	any valid SQL identifier.
database_name	the database to be used with the statement.
macro_name	the name of the Teradata SQL macro to be performed.
parameter_list	the Teradata SQL macro parameters.

ANSI Compliance

The macro form of DECLARE CURSOR is a Teradata extension to the ANSI SQL-99 standard because macros are not defined in ANSI SQL.

Authorization

None.

Rules

The following rules apply to the Macro DECLARE CURSOR statement:

- When the cursor is opened, the macro is performed. Once the macro has been performed, the results of macro execution are accessed by the application program as the results of a *request* cursor.
- None of the statements in the specified macro can be preprocessor or stored procedure declaratives.
- The macro cannot include any of the following SQL statements:

  - CHECKPOINT
  - CLOSE
  - COMMIT
  - CONNECT
  - DATABASE
  - DESCRIBE
  - ECHO
  - EXECUTE
  - EXECUTE IMMEDIATE
  - FETCH
  - LOGON
  - OPEN
  - POSITION
  - PREPARE
  - REWIND
  - SET BUFFERSIZE
  - SET CHARSET
  - SET SESSION

Example

Structure the Macro DECLARE CURSOR statement as follows:

```
DECLARE Ex CURSOR FOR EXEC NewEmp
```

DECLARE CURSOR (Request Form)

Purpose

Defines and assigns a name to a request cursor.

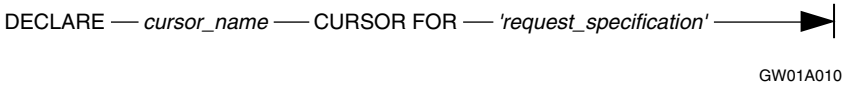
Invocation

Nonexecutable.

Preprocessor declaration.

Embedded SQL only.

Syntax



GW01A010

where:

Syntax element ...	Specifies ...
cursor_name	any valid SQL identifier.
request_specification	a literal character string enclosed in apostrophes comprised of any number of SQL statements separated by semicolons. The string is delimited by apostrophes by default. You can override this default using the QUOTESQL preprocessor parameter. The apostrophes syntactically distinguish the declaration of a request cursor from the other categories of cursor.

ANSI Compliance

The request form of DECLARE CURSOR is ANSI SQL-99-compliant.

Authorization

None.

Rules

The following rules apply to the Request DECLARE CURSOR statement:

- None of the statements in the *request_specification* can include any of the following SQL statements:
  - CHECKPOINT
  - CLOSE
  - COMMIT
  - CONNECT
  - DATABASE
  - DESCRIBE
  - ECHO
  - EXECUTE
  - EXECUTE IMMEDIATE
  - FETCH
  - LOGON
  - OPEN
  - POSITION
  - PREPARE
  - REWIND
  - SET BUFFERSIZE
  - SET CHARSET
  - SET SESSION
- The *request_specification* can be continued from line to line according to the syntax for continuation of quoted string literals in the client language.
- None of the statements in the *request_specification* can be preprocessor declaratives (embedded SQL only).
- When the cursor is opened (via the OPEN statement), the SQLCA is updated to reflect the success (SQLCODE in the SQLCA = 0) of the *first statement* of the request or failure (failure defined as an IMPLICIT ROLLBACK occurrence) of the request as an entity. A failure condition always overrides a success report. If successful, the activity count is shown in the third SQLERRD element in the SQLCA. To obtain the results of executing other statements of the request, use the POSITION statement (embedded SQL only).
- If any of the statements in the *request_specification* are data returning statements, retrieval of the data which is returned requires that the application program first position to the appropriate result set using the POSITION statement, OPEN automatically sets the position to the first statement of the request; the POSITION statement is not required in this case. Use a FETCH statement with an appropriate host variable list (INTO clause) or output SQLDA (USING DESCRIPTOR clause) (embedded SQL only).

Example

The following is an example of the Request DECLARE CURSOR statement.

This example omits the details of continuation of a literal character string from line to line, the rules for which are determined by the client language.

```
DECLARE Ex CURSOR FOR
   'UPDATE Employee SET Salary = Salary * 1.08
      WHERE DeptNo = 500;
   SELECT DeptName, Name, Salary FROM Employee,
          Department
      WHERE Employee.DeptNo = Department.DeptNo
      ORDER BY DeptName, Name'
```

DECLARE CURSOR (Selection Form)

Purpose

Defines and assigns a name to a selection cursor.

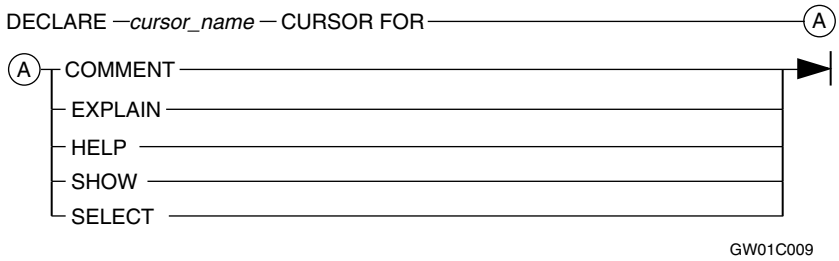
Invocation

Nonexecutable.

Preprocessor declaration.

Embedded SQL.

Syntax



where:

Syntax element ...	Specifies ...
cursor_name	the name you assign to this cursor. The name can be any valid SQL identifier.
COMMENT	a valid SQL comment-returning COMMENT statement. See "COMMENT (Returning Form)" on page 4-30.
EXPLAIN	a valid SQL EXPLAIN statement. See "EXPLAIN Modifier" on page 3-55.
HELP	a valid SQL HELP statement.
SHOW	a valid SQL SHOW statement. See "SHOW" in the chapter "SQL Help and Database Object Definition Tools: HELP and SHOW" of *Teradata RDBMS SQL Reference, Volume 4*.
SELECT	a valid embedded SQL or stored procedure SELECT statement. Note the restrictions listed under "Usage Notes" on page 7-27.

ANSI Compliance

The selection form of DECLARE CURSOR is ANSI SQL-99-compatible with extensions.

Authorization

None.

Usage Notes

The following rules apply to the Selection DECLARE CURSOR statement:

- The Teradata SQL WITH...BY clause cannot be specified.
- The SELECT privilege is required on all tables specified in the cursor declaration or on the database containing the tables.
- Each host variable referenced in the cursor specification must be defined prior to the selection DECLARE CURSOR statement.
- If the table identified in the *from_clause* is a grouped view (a view defined using a GROUP BY clause), the *table_expression* cannot contain a *where_clause*, a *group_by_clause* or a *having_clause*.
- If UNION is specified, the description of each result table in the union must be identical, except for column names. All columns of the temporary table formed by the union of the result tables are unnamed.
- If ORDER BY is used, each column specification in the ORDER BY clause must specify a column of the temporary table by name.

  Each unsigned integer in the ORDER BY clause must specify a column of the temporary table by relative number.

  A named column can be referenced either by a column specification or an unsigned integer; an unnamed column must be referenced by an unsigned integer.
- If UNION is specified, the result is the union of the individual result tables for each query in the union, with duplicate rows eliminated.

Examples

Four examples of use of the Selection DECLARE CURSOR statement are shown below.

Example 1

```
DECLARE Ex1 CURSOR FOR
SELECT * FROM Project ORDER BY Proj_Id
```

Example 2

```
DECLARE Ex2 CURSOR FOR
EXPLAIN SELECT DeptName, Name FROM Employee, Department
      WHERE Employee.DeptNo = Department.DeptNo
      ORDER BY DeptName, Name
```

Example 3

```
DECLARE Ex3 CURSOR FOR
SELECT a, b, 'X' FROM TableX WHERE a > b UNION
( SELECT a, b, 'Y' FROM TableY WHERE a > b INTERSECT
SELECT a, b, 'Y' FROM TableZ WHERE a > b ) ORDER BY 1,2
```

Example 4

```
DECLARE Ex4 CURSOR FOR
HELP TABLE Employee
```

DECLARE CURSOR (Stored Procedures Form)

Purpose

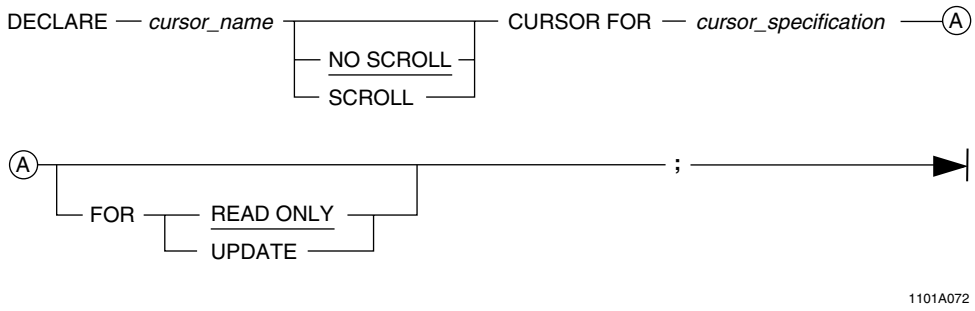
Defines and assigns a name to a cursor.

Invocation

Nonexecutable.

Stored procedures only.

Syntax



where:

Syntax element …	Specifies …
cursor_name	the name of the cursor to be declared.
[NO] SCROLL	whether the declared cursor can fetch the next row in the results set only or also be able to fetch the first row in the results set from any location in that set. See "FETCH (Stored Procedures Form)" on page 7-37.

IF you specify …	THEN the cursor …
SCROLL	can do either of the following: • scroll forward to the next row in the results set • scroll directly to the first row in the results set.
NO SCROLL	can only scroll forward to the next row in the results set. This is the default.

Syntax element …	Specifies …
READ ONLY	that you can only use the cursor to read the rows in the results set. This is the default.
UPDATE	that you can use the cursor to update the rows in the results set. In this case, the word update refers to both the delete and update operations.
cursor_specification	the SELECT statement that retrieves the rows the cursor reads or updates.

ANSI Compliance

The stored procedures form of DECLARE CURSOR is ANSI SQL-99-compliant.

Authorization

None.

Rules for the DECLARE CURSOR Statement

The following rules apply to the use of the DECLARE CURSOR statement within a stored procedure:

- A cursor declaration must be specified after any local declarations and before any handler declarations.
- The cursor name must be unique within the declaration of the same compound statement.
- If you do not specify an explicit scrollability clause, then NO SCROLL is the default and the cursor can only scroll forward.
- If you do not specify an explicit updatability clause, then FOR READ ONLY is the default.
- If you specify an explicit FOR UPDATE clause, then the cursor can be used for delete and update operations on its results rows.

Comparison of DECLARE CURSOR With FOR Statement

FOR statements contain DECLARE CURSOR statements.

DECLARE CURSOR and FOR differ in the following ways:

- The scope of a cursor is the entire compound statement and all its nested compound statements in which it is defined.

  The scope of a column or alias of a FOR statement is the body of the FOR statement.
- OPEN, FETCH, and CLOSE statements must be specified explicitly for cursors.

  OPEN, FETCH, and CLOSE operations are implicit in the performance of a FOR loop.

Example 1

The following example illustrates the correct usage of a cursor in a stored procedure. The declarations occur at lines 6 and 10.

```
CREATE PROCEDURE spSample1()
BEGIN
  L1: BEGIN
      DECLARE vName CHARACTER(30);
      DECLARE vAmt INTEGER;
      DECLARE EmpCursor CURSOR FOR
        SELECT EmpName, Salary
        FROM EmpDetails
        ORDER BY DeptCode;
      DECLARE DeptCursor CURSOR FOR
        SELECT DeptName
        FROM Department;
      DECLARE CONTINUE HANDLER FOR SQLSTATE '42000'
        BEGIN
          OPEN EmpCursor;
          …
        END;
        …
        …
  END L1;
END;
```

Example 2

The following example illustrates an implicit FOR READ ONLY cursor. An updatability clause is not specified in the declaration of EmpCursor, so its updatability is FOR READ ONLY by default.

```
CREATE PROCEDURE sp1()
BEGIN
  DECLARE EmpCursor CURSOR FOR
    SELECT *
    FROM Employee
    WHERE DeptCode = 101
    ORDER BY EmpID;
  …
END;
```

Example 3

The following example illustrates an explicitly declared FOR READ ONLY cursor.

```
CREATE PROCEDURE sp1()
BEGIN
  DECLARE EmpCursor CURSOR FOR
    SELECT *
    FROM Employee
    WHERE DeptCode = 101
  FOR READ ONLY;
    …
END;
```

Example 4

The following example illustrates the declaration of an updatable cursor specified by FOR UPDATE in the declaration of EmpCursor.

```
CREATE PROCEDURE sp1()
BEGIN
  DECLARE EmpCursor CURSOR FOR
    SELECT *
    FROM Employee
    WHERE DeptCode = 101
  FOR UPDATE;
  …
END;
```

FETCH (Embedded SQL Form)

Purpose

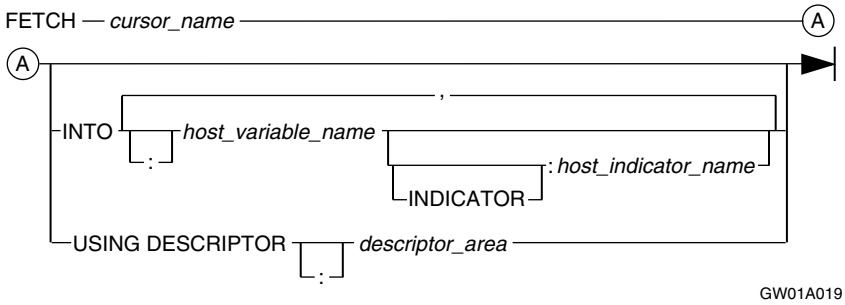
Positions a cursor to the next row of a response set and assigns the values in that row to host variables.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax element …	Specifies …
cursor_name	the name of an open selection cursor from which one or more rows are to be fetched.
host_variable_name	the variable to which the current row column value is to be assigned. The colon preceding the name is optional.
host_indicator_variable	the indicator variable. The colon preceding the name is mandatory.
descriptor_area	an SQL Descriptor Area (SQLDA). You can specify *descriptor_area* in a C program as a name or as a pointer reference (*sqldaname) when the SQLDA structure is declared as a pointer. See *Teradata Preprocessor2 Programmer Guide* for additional details.

ANSI Compliance

FETCH is ANSI SQL-99-compliant.

Authorization

None.

Rules

The following rules apply to FETCH:

- an SQLDA should be defined.
- FETCH cannot be performed as a dynamic SQL statement.
- The cursor identified by *cursor_name* must have been previously declared.
- The INTO clause is used with cursors which have been declared with either static or dynamic SQL statements. The USING DESCRIPTOR clause is intended for use with selection cursors which have been declared with dynamic SQL statements.
- The number of columns returned by the request must match the number of host variable specifications or the number of elements used in the SQLVAR array of the SQLDA (that is, must equal the value of the SQLD field).
- The main host variable specified by a host variable specification, or in the SQLVAR array of the SQLDA, and the corresponding column in the returned data must be of the same data type group, except that if the main host variable data type is approximate numeric, the temporary table column data type can be either approximate numeric or exact numeric.
- If the USING DESCRIPTOR clause is used, it is the responsibility of the programmer to verify that the SQLDATA and SQLIND pointer fields in SQLDA have been set to point to the appropriate host variables.

  Because the COBOL language provides no support for setting pointer values, Teradata supplies a service routine that can be called to do this task.

  The IBM dialect VS COBOL II provides a variant of the SET statement to set pointer values. Those using this dialect should consider this feature where appropriate.
- The cursor identified by *cursor_name* must be open.

- The cursor identified by *cursor_name* is positioned on the next row and values are assigned to host variables according to the following rules:

IF the cursor ...	THEN ...
has just been positioned	FETCH returns any of the following responses. **Condition** / **Returned Response** successful data returning statement / requested data no data / +100 non-rollback inducing SQLCODE < 0 / error
• is positioned on or after the last row, or • if there is no returned data	• the cursor is positioned after the last row • +100 is assigned to SQLCODE • the host variables remain unchanged
is positioned before a row	• the cursor is positioned on that row • values from the row are assigned to the host variables specified in the INTO clause or in the output SQLDA.
is positioned on a row other than the last row	• the cursor is positioned on the row immediately following that row • values from the new current row are assigned to the host variables specified in the INTO clause or in the output SQLDA.

- Values are assigned to the host variables specified in the INTO clause or in the SQLVAR array in the SQLDA in the order in which the host variables were specified. A value is assigned to SQLCODE last.
- If an error occurs in assigning a value to a host variable, one of the values -303, -304, or -413 is assigned to SQLCODE, and no further assignment to host variables occurs.

-

IF a field in the returned data is NULL and a corresponding host variable is …	THEN …
specified	-1 is assigned to the host indicator variable.
not specified	-305 is assigned to SQLCODE.

  In either case, the host variables remain unchanged.
- If a column value in the temporary table row is NOT NULL and a corresponding indicator host variable is specified, the indicator host variable is set as indicated in the following table.

IF …	THEN the host indicator value is set to …
the column and main host variables are typed CHARACTER and the column value is longer than the main host variable	the length of the column value.
anything else	0.

- Column values are set in the corresponding main host variables according to the rules for host variables.

FETCH (Stored Procedures Form)

Purpose

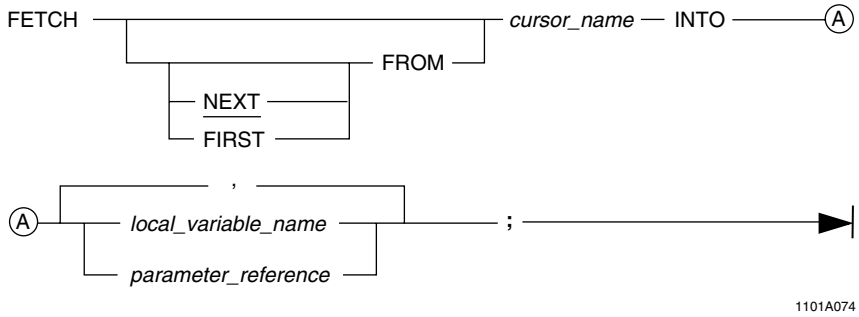
Positions a cursor to the next row of a response set and assigns the values in that row to local variables or parameters.

Invocation

Executable.

Stored procedures only.

Syntax



where:

Syntax element …	Specifies …
NEXT	to fetch the next row from the response set.
FIRST	to fetch the first row from the response set.
cursor_name	the name of an open selection cursor from which a row is to be fetched.
local_variable_name	the name of the local variable into which the fetched row is to be assigned.
parameter_reference	the name of the INOUT or OUT parameter into which the fetched row is to be assigned.

ANSI Compliance

The stored procedures form of FETCH is ANSI SQL-99-compliant.

Authorization

None.

When There Are No Rows In The Response Set

If there are no rows in the response set at the time the FETCH is performed, the following run time exception is raised:

- SQLCODE is set to 7362
- SQLSTATE is set to '02000'

This runtime warning condition can be handled within the procedure. If it is not handled, then the procedure continues from the next statement following the failed fetch operation.

Assignment Order for Fetched Rows

Row values are assigned to local variables or output parameters in the order those variables and parameters are declared in the INTO list.

Rules for the FETCH Statement

The following rules apply to the use of the FETCH statement within a stored procedure:

- The specified cursor must be open when FETCH is submitted.

  If the cursor is not open, the following run time exception is raised:
  - SQLCODE is set to 7631
  - SQLSTATE is set to '24501'
- The number of values returned by the fetch must match the number of local variables and output parameters declared in the fetch INTO list.

IF the mismatch is identified at ...	THEN Teradata responds with the following ...
compilation	compilation error SPL1027.
runtime	runtime exception: • SQLCODE is set to x • SQLSTATE is set to 'x'

- The data types of the fetched columns must match the data types specified for the local variables or outparameters to which they are assigned.
- You cannot specify a simple target specification that names table columns in the fetch INTO list. If you specify a non-valid fetch INTO list, error SPL1028 is reported during compilation.

  Instead, you must specify output parameters (INOUT and OUT) or local variables.

-

IF you specify …	THEN this row is fetched from the response set …
NEXT	the next, if one exists.
FIRST	the first.

- If you do not specify NEXT or FIRST, or if you specify NEXT, and the cursor is positioned on or after the last row in the response set, or if there is no data, then the cursor is positioned after the last row and the following completion condition is raised:
  - SQLCODE is set to 7632
  - SQLSTATE is set to '02000'

  The output parameter or local variable specified in the fetch INTO list for this value is not changed in this case.
- If you specify FIRST, then the declaration for the referenced cursor must specify SCROLL.

  If SCROLL is not specified, then error SPL1132 is reported at compilation.
- If you specify FIRST, but there is no data, then the following completion condition is raised:
  - SQLCODE is set to 7632
  - SQLSTATE is set to '02000'

  The output parameter or local variable specified in the fetch INTO list for this value is not changed in this case.

Example 1

The following example demonstrates that the cursor referenced by the FETCH statement is a valid cursor specification that returns columns correctly assigned to local variables with matching data types.

```
CREATE PROCEDURE sp1()
BEGIN
  DECLARE var1 INTEGER;
  DECLARE var2 CHARACTER(30)
  DECLARE ProjCursor CURSOR FOR
    SELECT projid, projectdesc FROM project ORDER BY projid;
  OPEN ProjCursor;
   WHILE (SQLCODE=0)
    FETCH ProjCursor INTO var1, var2;
   END WHILE;
  CLOSE ProjCursor;
END;
```

Example 2

In the following example, the FETCH statement after the WHILE loop raises completion condition SQLCODE 7632 and SQLSTATE '02000' and returns the message “no rows found” because the cursor is already positioned after the last row in the result set:

```
CREATE PROCEDURE sp1 (OUT par1 CHAR(50))
BEGIN
  DECLARE var1 INTEGER;
  DECLARE ProjCursor CURSOR FOR
    SELECT ProjId, ProjectDesc FROM Project;
  OPEN ProjCursor;
   WHILE (SQLCODE = 0)
    FETCH ProjCursor INTO var1, par1;
   END WHILE;
  FETCH ProjCursor INTO var1, par1;
  CLOSE ProjCursor;
END;
```

Example 3

The following example illustrates the usage of fetch orientation in the FETCH statement. Assume that Project contains 10 rows at the time performance of sp1 begins.

The first FETCH statement returns the first row, and the second FETCH returns the second row if no other rows have been fetched since ProjCursor was opened.

```
CREATE PROCEDURE sp1 (OUT Par1 INTEGER)
BEGIN
  DECLARE var1 CHARACTER(5);
  DECLARE var2 INTEGER;
  DECLARE ProjCursor SCROLL CURSOR FOR
    SELECT ProjectStatus FROM Project;
  OPEN ProjCursor;
    FETCH FIRST ProjCursor INTO var1;
    …
    FETCH NEXT ProjCursor INTO var1;
    …
  CLOSE ProjCursor;
END;
```

Example 4

The following example illustrates the usage of FETCH FIRST. Assume that Project contains 0 rows at the time performance of sp1 begins.

The FETCH statement raises the completion condition SQLCODE 7632 and SQLSTATE '02000' and returns the message “no rows found” because the table does not contain any rows.

```
CREATE PROCEDURE sp1 (OUT par1 INTEGER)
BEGIN
  DECLARE var1 CHARACTER(5);
  DECLARE var2 INTEGER;
  DECLARE ProjCursor SCROLL CURSOR FOR
    SELECT ProjectStatus FROM Project;
  OPEN ProjCursor;
    FETCH FIRST ProjCursor INTO var1;
    …
  CLOSE ProjCursor;
END;
```

OPEN (Embedded SQL Form)

Purpose

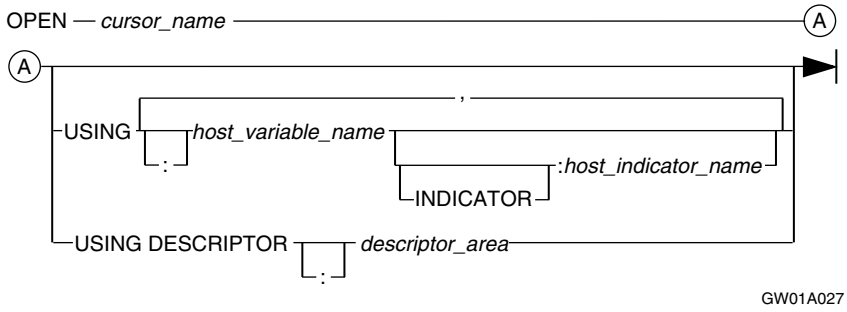
Opens a closed cursor and performs the SQL statement specified in the declaration for that cursor.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax element . . .	Specifies . . .
cursor_name	the name of the cursor to be opened.
host_variable_name	the variable to be used as input data for the cursor request. The colon preceding the name or names is optional.
host_indicator_variable	the indicator variable. The colon preceding the name is mandatory.
descriptor_area	an SQL Descriptor Area (SQLDA). You can specify *descriptor_area* in a C program as a name or as a pointer reference (*sqldaname) when the SQLDA structure is declared as a pointer. See *Teradata Preprocessor2 Programmer Guide* for additional details.

ANSI Compliance

OPEN (Embedded SQL Form) is ANSI SQL-99-compliant.

Authorization

None.

Rules

The following rules apply to the OPEN statement:

- an SQLDA should be defined.
- The cursor identified by *cursor_name* must have been previously declared.
- The cursor identified by *cursor_name* must be closed.
- Once the cursor is open, the associated static or dynamic SQL statement or statements are performed. The result spool file is then created and the cursor is positioned before the first row of the spool file.
- OPEN processing returns a zero in the SQLCODE field of the SQLCA, unless a failure (implicit rollback) occurs. For an SQLCODE of zero, the activity count is placed in the third SQLERRD element of the SQLCA structure.
- The following apply to the USING clause:
  - The USING clause identifies variables used as input to the SQL statement by *cursor_name*.
  - *host_variable_name* must be a valid client language variable declared prior to the OPEN statement, to be used as an input variable.

    A client structure can be used to identify the input variables.

    The number of variables specified must be the same as the number of parameter markers (the QUESTION MARK character) in the identified statement.

    The nth variable corresponds to the nth marker.

    Use of the colon before *host_variable_name* is optional.
  - *descriptor_area* identifies an input SQLDA structure previously defined by the application containing necessary information about the input variable or variables.

    The number of variables identified by the SQLD field of the SQLDA must be the same as the number of parameter markers (the QUESTION MARK character) in the identified statement.

    The nth variable described by the SQLDS corresponds to the nth marker.

- If the cursor is updatable or a REWIND or POSITION TO STATEMENT request exists for the cursor within a C or COBOL application program, then the OPEN statement is performed with KEEPRESP; else it is performed with NOKEEPRESP. For PL/I applications, KEEPRESP is the default.
- OPEN cannot be performed as a dynamic SQL statement.
- No more than 16 cursors can be OPEN at one time because the number of open cursors affects the processing of non-cursor-related statements.

  If an application has 16 cursors open, no other request can be issued until one or more cursors are closed.

Related Information

See "CLOSE" on page 7-15.

OPEN (Stored Procedures Form)

Purpose

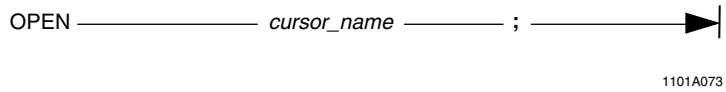
Opens a closed cursor and performs the SQL statement specified in the declaration for that cursor.

Invocation

Executable.

Stored procedures only.

Syntax



where:

Syntax element …	Specifies …
cursor_name	the name of the cursor to be opened.

ANSI Compliance

OPEN (Stored Procedures Form) is ANSI SQL-99-compliant.

Authorization

None.

Example

The following example is valid because the OPEN statement follows a valid cursor declaration statement in the same scope.

```
CREATE PROCEDURE sp1()
BEGIN
  DECLARE EmpCursor CURSOR FOR
    SELECT *
    FROM Employee
    ORDER BY EmpID;
  OPEN EmpCursor;
  …
END;
```

Related Information

See "CLOSE" on page 7-15.

POSITION

Purpose

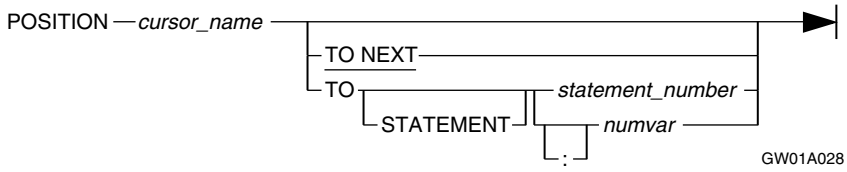
Positions a cursor to the beginning of the next statement or to the results of a specified statement.

Invocation

Executable.

Embedded SQL only.

Syntax



where:

Syntax element …	Specifies …
cursor_name	the name of an open cursor other than an Insertion cursor.
statement_number	an integer numeric for the statement number to which positioning is desired.
numeric_variable	a host variable conforming to type INTEGER that represents the statement number to which positioning is desired. Use of colon is optional.

ANSI Compliance

POSITION is ANSI SQL-99-compliant.

Authorization

None.

Rules

The following rules apply to the POSITION statement:

- The cursor is repositioned before the first result row (if any) of the statement specified and SQLCODE and other SQLCA values are set.
- With POSITION TO NEXT, the cursor is positioned to the next statement; with POSITION TO STATEMENT, the cursor is positioned to the statement specified.
- If the specified statement number does not exist (for example, TO STATEMENT 3 is coded and the cursor controls only two statements), the value -501 is assigned to SQLCODE, leaving the position of the cursor undefined.
- The cursor can be positioned either forward or backward from the current statement and can be repositioned to a given statement as many times as the application requires.

  For COBOL programs with multiple compile units, the cursor can only be positioned backward if a REWIND or POSITION TO STATEMENT occurs in the same compile unit as the declaration and the opening of the cursor.
- POSITION is used with any cursor except an Insertion cursor. For additional information, see DECLARE CURSOR, above.
- The statement or statements found by the cursor are *not* re-executed, but the cursor position in the spool file is adjusted.
- POSITION cannot be performed as a dynamic SQL statement.

Related Information

See "REWIND" on page 7-49.

REWIND

Purpose

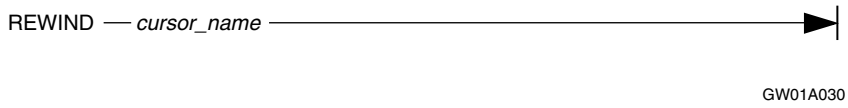
Positions or repositions a cursor to the beginning of the results of its first or only statement.

Invocation

Executable.

Embedded SQL only.

Syntax



GW01A030

where:

Syntax element ...	Specifies ...
cursor_name	the name of an open cursor other than an Insertion cursor.

ANSI Compliance

REWIND is a Teradata extension to the ANSI SQL-99standard.

Authorization

None.

Usage Note

The statement REWIND *cursor_name* is equivalent to the statement POSITION *cursor_name* TO STATEMENT 1.

Related Information

See "POSITION" on page 7-47.

Chapter 8:

Result Code Variables

This chapter describes a set of result code variables, also known as status parameters, shared by stored procedures and embedded SQL applications.

The result code variables described in this chapter are the following:

- "SQLSTATE" on page 8-2
- "SQLCODE" on page 8-6
- "ACTIVITY_COUNT" on page 8-10

SQLSTATE

Introduction

SQLSTATE is a variable (declared explicitly as a host variable in embedded SQL applications and implicitly as a status variable in stored procedures) that receives SQL statement status information.

ANSI Compliance

SQLSTATE is ANSI SQL-99-compliant. You should use it instead of SQLCODE any new applications you develop.

The SQLSTATE status variable used by stored procedure programs is non-ANSI SQL-99 standard.

Either SQLSTATE or SQLCODE is required for all embedded SQL applications written for ANSI mode.

SQLSTATE is not valid for embedded SQL applications running in Teradata transaction mode.

Where SQLSTATE Gets Its Information

SQLSTATE receives error codes from these subsystems:

- CLI/TDP
- Teradata database
- Preprocessor runtime manager

Structure of SQLSTATE

SQLSTATE is a five character string value divided logically into a two character class and a three character subclass. "SQLSTATE Class Definitions" on page D-2 lists the ANSI SQL-99-defined SQLSTATE classes.

Subclass values can be any numeric or simple uppercase Latin character string.

SQLSTATE Type Definition for Embedded SQL

The Teradata SQL preprocessor requires the following SQLSTATE data type definitions for the indicated client application language.

Client Language	Data Type Definition
• COBOL • PL/I	CHARACTER(5)
C	CHARACTER(6) The sixth character is always a null terminator.

SQLSTATE Type Definition for Stored Procedures

For stored procedures, the data type for SQLSTATE is predefined as CHARACTER(5).

Declaring SQLSTATE

SQLSTATE declaration is different for embedded SQL and stored procedure applications:

- SQLSTATE must be declared explicitly within an SQL DECLARE SECTION for embedded SQL applications.
- SQLSTATE is declared implicitly within a stored procedure application.

Using Both SQLSTATE and SQLCODE

You can define both SQLSTATE and SQLCODE in the same embedded SQL compilation unit.

You can also test the values of both SQLSTATE and SQLCODE variables within the same stored procedure.

In either case, both structures contain valid result codes.

How Teradata Error Codes Map to SQLSTATE

Unless otherwise specified, CLI/TDP, preprocessor runtime, and Teradata error codes map into SQLSTATE values using the ANSI-defined option of implementation-defined classes.

- Unmapped CLI/TDP errors are class T0.

  Their subclass contains the 3-digit CLI error code.

  For example, CLI error 157 (invalid Use_Presence_Bits option) produces class T0 and subclass 157.
- Unmapped Teradata errors are class T1 through class T9.

  The differentiating digit in the class number corresponds to the first digit of the Teradata error code.

  The subclass contains the remaining 3-digit Teradata error code.

  For example, Teradata error 3776 (unterminated comment) produces class T3 and subclass 776

The complete set of SQLSTATE class definitions and mappings for embedded SQL and stored procedure applications is provided in Appendix D: "SQLSTATE Mappings."

See *Teradata RDBMS Messages* for complete information on Teradata error codes.

SQLCODE to SQLSTATE Exception Mapping

Some SQLCODE values are generated by errors not originating within CLI, TDP, or Teradata SQL.

The exception mappings for these codes is provided in Appendix C: "SQL Communications Area (SQLCA)."

SQLSTATE Usage Constraints in Stored Procedures

The following usages of SQLSTATE are valid within a stored procedure:

- When specified as the operand of a SET statement.

  For example, the following statements are valid:

```
SET hErrorMessage = 'SQLSTATE' || sqlstate;

SET hSQLSTATE = SQLSTATE;
```

- When specified as an expression in an SQL statement within a stored procedure.

  For example, the following statements are valid.

```
INSERT INTO table_1 (column_1)
VALUES (:SQLSTATE);
UPDATE table_1
SET column_1 = column_1 + :ACTIVITY_COUNT;
```

The following uses of SQLSTATE are not valid within a stored procedure.

- SQLSTATE cannot be declared explicitly.
- SQLSTATE cannot be SET to a value or an expression.

  For example, the following statement is not valid.

  ```
  SET SQLSTATE = h1 + h2;
  ```
- SQLSTATE cannot be specified in the INTO clause of a SELECT statement.

  For example, the following statement is not valid.

  ```
  SELECT column_1 INTO :SQLSTATE FROM table_1;
  ```
- SQLSTATE cannot be specified in place of the INOUT and OUT parameters of a CALL statement.

SQLCODE

Introduction

In ANSI transaction mode, SQLCODE is a host variable (embedded SQL) or status variable (stored procedures) that receives SQL statement status information. The status codes permit an application program to test whether an executable embedded SQL statement completed successfully or not.

In Teradata transaction mode (for embedded SQL), SQLCODE is a field within SQLCA (see "SQL Communications Area (SQLCA)" on page C-1.

ANSI Compliance

SQLCODE is not ANSI SQL-99-compliant. SQLCODE was deprecated in the ANSI SQL-92 standard and is not defined in the SQL-99 standard. The ANSI SQL committee recommends that new applications be written using SQLSTATE (see "SQLSTATE" on page 8-2) in place of SQLCODE.

SQLCODE is required for all embedded SQL applications written for ANSI transaction mode that do not specify a SQLSTATE host variable. In other words, you must specify one or the other (or both) for any embedded SQL application you write, and SQLSTATE is the preferred choice.

A stored procedure application can test the status of either SQLCODE or SQLSTATE or both.

The SQLCODE field within the SQLCA is also not defined in the ANSI SQL-99 standard, nor is SQLCA. Stored procedures do not use SQLCA.

SQLCODE in ANSI Transaction Mode

SQLCODE is defined as a 32-bit signed integer.

If SQLCODE is not defined within an SQL DECLARE SECTION in an embedded SQL application, then the preprocessor assumes that a valid SQLCODE is defined within the program.

You can test the status values of either SQLCODE or SQLSTATE for stored procedure applications.

SQLCODE In Teradata Transaction Mode

The SQLCODE field within SQLCA communicates the result of performing an SQL statement to an embedded SQL application program. Stored procedures do not use SQLCA.

When the preprocessor option SQLFLAGGER or -sf is set to NONE, then SQLCODE is defined in an embedded SQL application program via SQLCA. Otherwise, you must define SQLCODE explicitly in your application.

You can test the status values of either SQLCODE or SQLSTATE for stored procedure applications.

SQLCODE Value Categories

The SQLCODE value returned to an application after an embedded SQL or stored procedure statement is performed always falls into one of three categories, as explained by the following table:

This SQLCODE value . . .	Indicates that . . .
negative	an error occurred during processing. The nature of the error is indicated by the numeric value of the code.
0	processing was successful.
positive	termination was normal. Positive values other than 0 and +100 indicate Teradata warnings. For example, an SQLCODE value of +100 indicates one of the following results: • No rows were selected. • All selected rows have been processed.

When SQLCODE Is Updated

SQLCODE is updated during runtime after each executable statement has been processed. You must write your own application code to test the status codes written to the SQLCODE variable.

- For embedded SQL applications, see "WHENEVER" on page 4-48 for information about condition handling.
- For stored procedure applications, see "Completion Condition and Exception Condition Handlers" on page 9-25 for information about condition handling.

When to Test SQLCODE

Test SQLCODE after each performance of an executable SQL statement to ensure that the statement completes successfully or that an unsuccessful statement is handled properly.

You must also write code to resolve unacceptable SQLCODE values.

- For embedded SQL applications, see "WHENEVER" on page 4-48 for information about condition handling.
- For stored procedure applications, see "Completion Condition and Exception Condition Handlers" on page 9-25 for information about condition handling.

SQLCODE Testing Example

Suppose an application creates a temporary table and then populates it using an INSERT … SELECT statement.

You would write your application code to perform an SQLCODE check immediately after performing the CREATE TABLE statement.

If this statement fails to create the table successfully, there is no reason to process the INSERT … SELECT statement that follows it, so you would code WHENEVER statements to perform some appropriate action to prevent performing the INSERT … SELECT or, if all goes as planned, to continue with processing.

You should also test the INSERT … SELECT statement to ensure that subsequent references to the temporary table are valid.

For example, the SQLCODE value might be 0, indicating that one or more rows were successfully selected and inserted.

The value might also be +100, indicating that no rows were selected or inserted, and the table is empty. Any subsequent references to the empty temporary table would be inaccurate in that case, so some action needs to be taken to ensure that further references to the empty temporary table do not occur.

How Teradata Error Messages Map to SQLCODE Values

For information on mapping SQLCODEs to Teradata error message numbers, see Appendix C: "SQL Communications Area (SQLCA)."

SQLCODE Usage Constraints for Stored Procedures

The following uses of SQLCODE are valid within a stored procedure:

- When specified as the operand of a SET statement.

  For example, the following statement is valid.

  ```
  SET h1 = - SQLCODE;
  IF SQLCODE = h1 THEN
  …
  …
  END IF;
  ```

- When specified as an expression in an SQL statement within a stored procedure.

  For example, the following statements are valid.

  ```
  INSERT INTO table_1 (column_1)
  VALUES (:SQLCODE);
  UPDATE table_1
  SET column_1 = column_1 + :SQLCODE;
  ```

The following usages of SQLCODE are not valid within a stored procedure:

- SQLCODE cannot be declared explicitly.
- SQLCODE cannot be SET to a value or an expression.

  For example, the following statement is not valid.

  ```
  SET SQLCODE = h1 + h2;
  ```

- SQLCODE cannot be specified in the INTO clause of a SELECT statement.

  For example, the following statement is not valid.

  ```
  SELECT column_1 INTO :SQLCODE FROM table_1;
  ```

- SQLCODE cannot be specified in place of the INOUT and OUT parameters of a CALL statement.

ACTIVITY_COUNT

Introduction

The ACTIVITY_COUNT status variable returns the number of rows affected by an SQL DML statement in an embedded SQL or stored procedure application.

It provides the same function as the Activity Count word in the SQLERRD array of SQLCA for embedded SQL applications (see Appendix C: "SQL Communications Area (SQLCA)").

ANSI Compliance

ACTIVITY_COUNT is a Teradata extension to the ANSI SQL-99 standard.

When ACTIVITY_COUNT Is Set

ACTIVITY_COUNT is initialized to 0 when a stored procedure or embedded SQL application begins execution and is updated during runtime after each executable SQL statement is processed. You must write your own code to test the count it receives.

- For embedded SQL applications, see "WHENEVER" on page 4-48 for information about condition handling.
- For stored procedure applications, see "Completion Condition and Exception Condition Handlers" on page 9-25 for information about condition handling.

When to Test ACTIVITY_COUNT

Test ACTIVITY_COUNT after each performance of an executable SQL statement for which you need to know the number of rows affected to ensure proper error handling.

You must write your own code to handle error processing based on ACTIVITY_COUNT values.

- For embedded SQL applications, see "WHENEVER" on page 4-48 for information about condition handling.
- For stored procedure applications, see "Completion Condition and Exception Condition Handlers" on page 9-25 for information about condition handling.

Usage Constraints on ACTIVITY_COUNT

The following usages of ACTIVITY_COUNT are valid within a stored procedure or embedded SQL application:

- ACTIVITY_COUNT can be specified as the operand of a SET statement.

  For example, the following statement is valid.

  ```
  SET h1 = h1 + ACTIVITY_COUNT;
  ```

- ACTIVITY_COUNT can be specified as an expression in an SQL statement.

  For example, the following statements are valid.

  ```
  INSERT INTO table_1 (column_1)
  VALUES (:ACTIVITY_COUNT);
  UPDATE table_1
  SET column_1 = column_1 + :ACTIVITY_COUNT;
  ```

The following usages of ACTIVITY_COUNT are not valid:

- ACTIVITY_COUNT cannot be declared explicitly within a stored procedure.
- ACTIVITY_COUNT cannot be SET to a value or an expression.

  For example, the following statement is not valid.

  ```
  SET ACTIVITY_COUNT = h1 + h2;
  ```

- ACTIVITY_COUNT cannot be specified in the INTO clause of a SELECT statement.

  For example, the following statement is not valid.

  ```
  SELECT column_1 INTO :ACTIVITY_COUNT FROM table_1;
  ```

- ACTIVITY_COUNT cannot be specified in place of the INOUT and OUT parameters of a CALL statement in a stored procedure.

Section 4:
SQL Stored Procedures

Section Contents

Chapter 9:

SQL Stored Procedures

This chapter describes SQL-based stored procedures.

Topics include:

- Stored procedure overview
- Creating / replacing a stored procedure
- Grant privileges on stored procedures
- Performing a stored procedure
- Restrictions on stored procedures
- Stored procedure lexicon
- DDL statements in stored procedures
- DML statements in stored procedures
- DCL statements in stored procedures
- SQL operations on stored procedures
- Control statements in stored procedures
- Completion / exception condition handlers in stored procedures
- Cursor declarations
- Rules for using SQL statements with stored procedures
- Using dynamic SQL in stored procedures
- Self-referencing stored procedures
- Archive and restore operations with stored procedures
- Control statements in stored procedures
- Tactical queries and stored procedures
- Debugging stored procedures
- Sample stored procedure

Stored Procedure Overview

What is a Stored Procedure?

A stored procedure is a combination of SQL statements and control and conditional handling statements that provides an interface to Teradata.

A stored procedure is a database object performed on the Teradata server. Typically, a stored procedure consists of a procedure name, input and output parameters, and a procedure body.

Each stored procedure has a corresponding "stored procedure table" in the database, containing the stored procedure body the user specifies and the corresponding stored procedure object code.

The parameters and attributes of the procedure object are stored in the Data Dictionary tables.

Definitions: Procedure Body and Source Text

The following terms are useful in understanding the structure of a stored procedure.

Term	Definition
Procedure body	The set of statements constituting the main tasks of the stored procedure. The procedure body can be a single control statement or SQL statement, or a BEGIN - END compound statement, or block. Compound statements can also be nested.
Source text	The entire definition of a stored procedure, including the CREATE/REPLACE PROCEDURE statement, parameters, procedure name, and the stored procedure body.

Elements in a Stored Procedure Body

A stored procedure can have a procedure body containing the following elements:

Stored procedure body of this type …	Can contain the following …
Single statement	one SQL statement or control statement, including dynamic SQL. **Note**: The following elements are *not* allowed: • Any declaration (local variable, cursor, or condition handler) statement • A cursor statement (OPEN, FETCH, or CLOSE)
Compound statement	• Local variable declarations • Cursor declarations • Condition handler declaration statements • Control statements • SQL DML, DDL, and DCL statements supported by stored procedures, including dynamic SQL.

It is not mandatory to enclose the procedure body within BEGIN - END keywords (that is, in a compound statement) if the procedure body contains only one statement and does not contain any declarations.

What Does a Stored Procedure Provide?

A stored procedure provides control and condition handling statements, in addition to multiple input and output parameters and local variables, that make SQL a computationally complete programming language.

Applications based on stored procedures:

- Perform well by reducing network traffic in the client-server environment.
- Allow encapsulation and enforcement of business rules on the server, contributing to improved application maintenance.
- Provide better transaction control and improved application security
- Provide better security by restricting user access to the procedures rather than the data tables.
- Enable all SQL and control statements embedded in a stored procedure to be performed on the server through one CALL statement.

  Nested CALL statements extend performance by combining all transactions and complex queries in the nested procedures into one explicit transaction, and handling errors internally in the nested procedures.

DDL Statements for Stored Procedure Objects

The following stored procedure-related DDL statements are documented in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*:

- ALTER PROCEDURE
- CREATE/REPLACE PROCEDURE
- DROP PROCEDURE

Granting Privileges on Stored Procedures

The privileges to create, drop, execute, or alter a procedure can be granted using the GRANT statement and revoked using the REVOKE statement.

This privilege ...	Can be granted to this level of database object ...
CREATE PROCEDURE	• database • user
ALTER PROCEDURE DROP PROCEDURE EXECUTE PROCEDURE	• database • user • stored procedure

- DROP PROCEDURE is granted automatically to all users and databases when a new user or database is created.
- EXECUTE PROCEDURE is granted automatically only to the creator of a stored procedure when the object is created.

  Teradata does not grant this privilege automatically to the owner of the stored procedure when the owner and creator are not the same.

See the chapter "SQL Data Control Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4* for further information about the SQL forms of the GRANT and REVOKE statements and the ALTER PROCEDURE, CREATE PROCEDURE, DROP PROCEDURE, and EXECUTE PROCEDURE privileges.

Performing a Stored Procedure

The CALL Statement

A stored procedure is performed using the SQL CALL statement. But executing a CALL statement does not initiate a transaction.

Initiating a Transaction

Execution of the first SQL statement other than a control statement inside the stored procedure initiates a transaction. A control statement cannot initiate a transaction.

- In Teradata transaction mode, each statement within the stored procedure is a separate transaction. You can explicitly initiate a transaction by specifying BT and ET inside the stored procedure body.
- In ANSI transaction mode, unless the body of the stored procedure ends with a COMMIT, the actions of the stored procedure are not committed until a COMMIT or ROLLBACK occurs in subsequent statements.

The request number is incremented for each SQL request inside the stored procedure. For details of stored procedure execution, see "CALL" on page 3-13.

Data Type Codes

Teradata returns a specific set of CLIv2 data type codes to the calling application when the CALL statement is submitted.

See "DataInfo Parcel" in *Teradata Call-Level Interface Version2 for Network-Attached Systems* or *Teradata Call-Level Interface Version2 for Channel-Attached Systems* for a complete listing of the data type codes possible with a CALL statement.

Stored Procedure Parameters

The data type codes returned when the CALL statement is submitted include the parameter type. Stored procedure parameters can be of three types:

- IN (input parameter)
- INOUT (either input or output, or both)
- OUT (output parameter)

Parameters of all data types are nullable in stored procedures.

Restrictions on Stored Procedures

The following are some important restrictions on the size and use of stored procedures:

- The stored procedure body size of a stored procedure is limited to 6.4 MB. But there is no limit on the stored procedure object code (compiled stored procedure) size.
- The parser limits apply if the SQL statements within a stored procedure are large or complex.
- CALL statements are not allowed in a multistatement request.
- The number of nested CALL statements, including recursion, cannot exceed 15.
- The number of parameters in a procedure cannot exceed 256.
- Stored procedures cannot be renamed across databases.
- A stored procedure created in ANSI transaction mode cannot be performed in Teradata transaction mode, and vice versa.

  You can, however, perform the stored procedure after recreating it in the new session mode using REPLACE PROCEDURE. See "CREATE/ REPLACE PROCEDURE" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*.
- A stored procedure created on one platform (UNIX MP-RAS, for example) cannot be performed on another platform (Windows, for example).

  But, this limitation can be overcome by recompiling a stored procedure using the ALTER PROCEDURE statement. See "ALTER PROCEDURE" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*
- Stored procedures cannot be performed from triggers because you cannot include a CALL statement in a trigger definition.
- You can perform a stored procedure from a macro if the procedure is the only statement in the macro.
- Stored procedures do not support the following:
  - EXPLAIN and USING modifiers inside a stored procedure
  - EXECUTE macro statement
- A stored procedure created in a particular date form always displays the same date-time format without respect to the date form set for the executing session.

Stored Procedure Lexicon

Names

The names of stored procedures, as well as stored procedure parameters, local variables, labels, for-loop correlation names and columns, cursors, and for-loop variables must be valid Teradata SQL names (or identifiers).

All rules applying to naming a database object also apply to naming a stored procedure. See "SQL Lexicon" in *Teradata RDBMS SQL Reference, Volume 1*.

The following rules apply to specific names in stored procedures:

This name ...	Must be unique in ...
correlation or column	a FOR iteration statement, or a DECLARE CURSOR statement. The same correlation or column name can be reused in nested or non-nested FOR statements within a stored procedure. A correlation or column name can be the same as the for- loop variable and cursor names in a FOR statement.
cursor	nested FOR statements. A cursor name can be the same as the for-loop variable and correlation or column name in a FOR statement. In cursors defined using DECLARE CURSOR, a cursor name must be unique in the compound statement in which it is declared.
for-loop variable	nested FOR iteration statements. A for-loop variable name can be the same as the cursor name and correlation or column name in a FOR statement.
label	nested iteration statements or a group of nested BEGIN - END statements. Label names of iteration statements can be reused with other non-nesting iteration constructs or non-nesting BEGIN - END statements in a stored procedure. Labels have their own name space, so they do not interfere with other identifiers used for local variables and parameters.
local variable	the BEGIN - END compound statement in which it is declared.
parameter	a stored procedure. For example, a parameter name cannot be repeated for a local variable in the same stored procedure.
stored procedure	a database since it falls in the name space of tables, macros, views and triggers.

Keywords

Keywords in stored procedures are not case-sensitive. Uppercase and lowercase can normally be used interchangeably.

You can use more than one blank space character between syntax elements to increase readability, but multiple sequential blank space characters are treated as a single space.

For information on keywords, see "SQL Lexicon" in *Teradata RDBMS SQL Reference, Volume 1*. For keywords themselves, see "Restricted Words for V2R5.0" in *Teradata RDBMS SQL Reference, Volume 1*.

Literals

All Teradata-supported literals for directly specifying values in the text of an SQL statement, including control statements, are valid in stored procedures.

Local Variables

You can specify local variables of any Teradata-supported data type in the variable declaration statements within a BEGIN - END compound statement of the stored procedure.

A compound statement can have multiple variable declarations, and each DECLARE statement can contain multiple local variables.

If a local variable is specified in an SQL statement other than a control statement, it must be prefixed with a COLON character (**:**). Otherwise, it is assumed to be a column name.

A local variable name cannot be any of the following names reserved for status variable names:

- SQLCODE
- SQLSTATE
- ACTIVITY_COUNT

A DEFAULT clause, if specified for a local variable, can contain a literal. Expressions are not allowed.

local variable with the label of the corresponding compound statement in which the variable is declared. This helps to avoid conflicts that might be caused by reused local variables in nested compound statements.

See "DECLARE" on page 9-55 for details on the use of local variables.

Parameters

A stored procedure can have up to 256 parameters of any Teradata-supported data type and character. Stored procedure parameters and their attributes are stored in the DBC.TVFields table of the data dictionary.

If a parameter is specified in an SQL statement other than a control statement, it must be prefixed with a COLON character (**:**). Otherwise, Teradata assumes it to be a column name. See "Host Variables" on page 4-7 for a description of the same concept under a different name.

The following three names are reserved for status variables and cannot be used for parameters:

- SQLCODE
- SQLSTATE
- ACTIVITY_COUNT

The following clauses cannot be specified for parameters:

- DEFAULT
- FORMAT
- NOT NULL

Parameter Rules

The following rules apply to IN, INOUT and OUT parameters:

-

THIS parameter ...	CAN be a part of the ...	BUT cannot be a part of the ...
IN	source specification of an SQL statement	target specification of an SQL statement.
OUT	target specification of an SQL statement	source specification of an SQL statement.
INOUT	source and target specifications of an SQL statement.	

- INOUT parameters can be used for both input and output values.
  - You can specify an input value as an argument for the INOUT parameter while initiating the stored procedure execution.
  - You can read the output value from the same parameter after execution of the procedure.
- The total data size of all input parameters in a CREATE/REPLACE PROCEDURE cannot exceed 64 000 bytes.
- The total data size of all output parameters in CREATE/REPLACE PROCEDURE cannot exceed 64 000 bytes.
- When you invoke a stored procedure, the IN constants assume the data type of the value specified unless overridden in the CALL statement.

  The values of INOUT constants assume the data type defined for the parameter unless overridden in the CALL statement.

For other rules, details and examples of the usage of parameters in stored procedures, see "CALL" on page 3-13 and "CREATE PROCEDURE" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*

Labels

You can use a label with iteration statements (FOR, LOOP, REPEAT and WHILE) and BEGIN - END compound statements in a stored procedure. The following rules apply:

- A beginning label must be terminated by a COLON character (:)
- An ending label is not mandatory for an iteration statement or compound statement.

  If an ending label is specified, it must have a corresponding beginning label associated with that iteration or BEGIN - END statement. For example, an ending label following an END WHILE must have an equivalent beginning label and COLON character preceding the corresponding WHILE.
- The scope of a label is the iteration statement or BEGIN - END compound statement with which it is associated.

  This implies that if another iteration statement or compound statement is nested, the label name associated with the outer iteration or compound statement must not be used with any inner iteration statement(s) or compound statement(s).

For-Loop Variables

A for-loop variable is normally used as the name for a FOR iteration statement.

A for-loop variable must be used to qualify references to columns or correlation names. If not qualified with the for-loop variable name, the column or correlation name references in SQL statements, including control statements, are assumed to be parameters or local variables. The following rules apply:

- When used in an SQL statement other than a control statement, a for-loop variable must be prefixed with a COLON character (:).
- The scope of the for-loop variable is confined to the FOR statement with which it is associated.

  In the case of nested FOR statements, the for-loop variable associated with an outer FOR statement can be referenced in other statements inside the inner FOR statement(s).

Cursors

Cursors can be declared in stored procedures in two ways:

- Using the DECLARE CURSOR (Selection Form) statement
- Using the FOR iteration statement

The rules and guidelines governing the use of cursors in stored procedures are described in Chapter 7: "Cursors and Cursor Control Statements."

The following bullets summarize that information:

- The following statements with open cursors are not allowed in a stored procedure:
  - POSITION
  - REWIND
  - All SQL transaction statements
- The SELECT statement specified in the FOR iteration statement or DECLARE CURSOR statement is called the cursor specification. You can use both read-only and updatable cursors in a stored procedure.
- Cursors declared in a FOR statement and in a DECLARE CURSOR statement have the following differences:

FOR Loop Cursor	DECLARE CURSOR Cursor
The scope of the cursor is confined to the FOR statement in which it is defined. In the case of nested FOR statements, the cursor name specified in an outer FOR statement can be referenced in statements inside the inner FOR statement(s).	The scope of the cursor is the BEGIN - END compound statement in which it is declared. In nested compound statements, the scope of a cursor specified in an outer compound statement includes all the inner compound statements.
A positioned DELETE or UPDATE statement referencing the cursor makes it updatable.	The FOR UPDATE option makes the cursor updatable.
OPEN, FETCH and CLOSE operations take place implicitly as part of the for-loop execution.	OPEN, FETCH or CLOSE needs to be submitted as an explicit request.

- You can label FOR statements in which cursors are declared.
- You cannot label DECLARE CURSOR statements.

See "DECLARE CURSOR (Selection Form)" on page 7-26 and "FOR" on page 9-111 for more details and examples of the usage of cursors within stored procedures.

Correlation and Column Names

The columns in the cursor specification of a FOR statement or DECLARE CURSOR statement can be aliased using an optional keyword AS.

The ANSI SQL standard refers to aliases as correlation names. They are also referred to as range variables. The following rules apply:

- An expression used in the cursor specification must be aliased.
- The data type (including the character data type, CHARACTER SET in case of character type) of a correlation name or column is the data type of the corresponding correlation name or column in the cursor specification.
- An correlation name or column must be referenced in the body of the FOR iteration statement by qualifying it with the associated for-loop variable name. An unqualified name is assumed to be a local variable or a parameter name.
- The scope of a correlation name or column in a FOR iteration statement is the body of the FOR statement.

  In the case of nested FOR statements, a correlation name or column associated with an outer FOR statement can be referenced in statements inside the inner FOR statement(s).
- Correlation names or column names used in an SQL statement other than a control statement must be prefixed with a COLON character (:) when used in a stored procedure.

Data Types Supported in Stored Procedures

All data types supported by Teradata can be used for stored procedure parameters and local variables.

See “SQL Lexicon” in *Teradata RDBMS SQL Reference, Volume 1* and *Teradata RDBMS SQL Reference, Volume 3* for details and usage considerations.

Delimiters

All ANSI and Teradata supported delimiters can be used in stored procedures. Some examples are:

Use this delimiter …	Named …	To …
;	SEMICOLON	end each statement in a stored procedure body, including DML, DDL, DCL statement, control statements, and control declarations. The SEMICOLON character is the mandatory statement separator.
:	COLON	prefix status variables, local variables, parameters, and for-loop correlation names used in SQL statements other than control statements within stored procedures. A COLON character must suffix the beginning label if used with a compound statement or iteration statement.
(	LEFT PARENTHESIS	enclose lists of parameters or call arguments.
)	RIGHT PARENTHESIS	

The use of other delimiters like the COMMA character (,), the FULLSTOP character (.), and SQL operators in stored procedures is identical to their use elsewhere in Teradata SQL.

Lexical Separators

All lexical separators (comments, pad characters, and newline characters) supported by Teradata SQL can be used in stored procedures.

Newline characters need to be used wherever possible in the stored procedure body to increase its readability.

The newline character is implementation-specific and is typed by pressing the Enter key on non-3270 terminals or the Return key on 3270 terminals.

Locking Modifiers

Locking modifiers are supported with all DML, DDL, and DCL statements used in stored procedures except CALL.

Status Variables

The following status variables are supported for stored procedures:

- SQLCODE

  SQLCODE contains the error or warning code and reflects the condition of an SQL statement, including control statements.
- SQLSTATE

  SQLSTATE contains the ANSI-compliant code corresponding to SQLCODE.

  You should use SQLSTATE in preference to SQLCODE because SQLCODE has been dropped from the ANSI SQL-99 standard.
- ACTIVITY_COUNT

  ACTIVITY_COUNT returns the number of rows affected by a DML statement in a stored procedure.

  ACTIVITY_COUNT is a Teradata extension to the ANSI SQL-99 standard.

Status variables in an SQL statement other than a control statement must be prefixed with a COLON character (:) when used in a stored procedure.

For the definition and details of these status variables see Chapter 8: “Result Code Variables,” Appendix C: “SQL Communications Area (SQLCA),” and Appendix D: “SQLSTATE Mappings.”

For a complete listing of Teradata return codes mapped to their corresponding SQLSTATE codes, see Appendix D: “SQLSTATE Mappings.”

For information on mapping SQLCODEs to Teradata error codes, see Appendix C: “SQL Communications Area (SQLCA).”

Initial Values of Status Variables

Status variables are mapped to Teradata error codes and reflect the status of execution of stored procedure SQL statements, including control statements.

The initial value indicated in the last column is the value set at the beginning of stored procedure or embedded SQL application execution.

Status variable	Data Type	Initial Value
SQLCODE	SMALLINT	0
SQLSTATE	CHARACTER(5) CHARACTER SET is numeric or uppercase LATIN characters or a mix of both.	’00000’
ACTIVITY_COUNT	DECIMAL(18,0)	0

The values set at the end of the statement performance reflect the exception condition or completion condition, if one occurs. These conditions, other than successful completion, can be handled if a condition handler is specified for the particular SQLSTATE value.

After successful completion, the status variables are set to appropriate values for SQL statements other than control statements within the stored procedure. The status variables do not change for control statements.

See "DECLARE HANDLER" on page 9-58 for definitions and details of the successful, completion, and exception conditions.

Restrictions on Use of Status Variables in Stored Procedures

The following constraints apply to the usage of status variables in a stored procedure:

- The status variables are local to a stored procedure.

  They are not exported to the calling procedure in the case of nested stored procedures.
- You cannot explicitly declare status variables.
- Status variables cannot be specified in the following circumstances:
  - as the assignment target (LHS) of a SET statement.
  - in the INTO clause of an SQL SELECT... INTO statement.
  - in place of INOUT and OUT arguments in an SQL CALL statement.

Comments in a Stored Procedure

Teradata style comments, as well as ANSI SQL-99 style comments, can be used in a stored procedure. The ANSI SQL-99 definition of comments includes Teradata style comments.

Nested Teradata-style comments are not allowed.

Supported DDL Statements in Stored Procedures

Supported DDL Statements

You can use the following SQL DDL (Data Definition Language) statements in a stored procedure when the creator is also the immediate owner of the procedure. That is, a stored procedure can contain DDL statements only if it is created in the database of the user.

DDL Statements Supported in Stored Procedures

Teradata supports the following SQL DDL statements for stored procedures:

- ALTER TABLE
- ALTER TRIGGER
- BEGIN LOGGING
- COLLECT STATISTICS
- COMMENT
- CREATE DATABASE
- CREATE HASH INDEX
- CREATE INDEX
- CREATE JOIN INDEX
- CREATE MACRO
- CREATE PROFILE
- CREATE ROLE
- CREATE TABLE
- CREATE TRIGGER
- CREATE USER
- CREATE VIEW
- DELETE DATABASE
- DELETE USER
- DROP DATABASE
- DROP HASH INDEX
- DROP INDEX
- DROP JOIN INDEX
- DROP MACRO
- DROP PROCEDURE
- DROP PROFILE
- DROP ROLE
- DROP STATISTICS

- DROP TABLE
- DROP TRIGGER
- DROP USER
- DROP VIEW
- END LOGGING
- MODIFY DATABASE
- MODIFY PROFILE
- MODIFY USER
- RENAME MACRO
- RENAME PROCEDURE
- RENAME TABLE
- RENAME TRIGGER
- RENAME VIEW
- REPLACE MACRO
- REPLACE TRIGGER
- REPLACE VIEW

For information on the above DDL statements, see "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*.

Usage Notes

- You can use only DDL COMMENT statement in a stored procedure. You cannot specify DML COMMENT statements, which are restricted to embedded SQL applications, to fetch the comments for database objects, columns of a table, and parameters.
- All variations of CREATE TABLE statement are valid.
- If a CREATE VOLATILE TABLE statement is included in a stored procedure, the volatile table is created in the login user's database. If an object with the same name already exists in that database, the result is a runtime exception.

  DML statements within a stored procedure referencing the volatile table must either have the user's login database as the qualifier, or not have any qualifying database name.

- A CREATE DATABASE or CREATE USER statement in a stored procedure must contain the FROM clause. The result depends on this clause:

WHEN a stored procedure …	THEN …
contains a FROM clause	the specified database is the immediate owner of the USER or DATABASE created.
does not contain a FROM clause	an SPL compilation error is reported during procedure creation: 5568 – "SQL statement is not supported within a stored procedure." If CREATE USER/ DATABASE without a FROM clause is specified as a dynamic SQL statement within a stored procedure, the same error is reported as runtime exception during stored procedure execution.

Unsupported DDL Statements in Stored Procedures

Unsupported DDL Statements

You cannot use the following DDL statements in a stored procedure:

- ALTER PROCEDURE
- CREATE PROCEDURE
- DATABASE
- EXPLAIN (in any form)
- HELP (in any form)
- REPLACE PROCEDURE
- SET ROLE
- SET SESSION ACCOUNT
- SET SESSION COLLATION
- SET SESSION DATEFORM
- SET TIME ZONE
- SHOW (in any form)

Transaction Mode Impact on DDL Statements in Stored Procedures

The behavior of DDL statements specified in stored procedures at runtime depends on the transaction mode of the Teradata session in which the procedure is created.

- A DDL statement specified within an explicit (user-defined) transaction in a stored procedure in Teradata transaction mode must be the last statement in that transaction. Otherwise, a runtime exception (SQLCODE: 3932, SQLSTATE: ‘T3932’) is raised.
- When you perform a stored procedure in ANSI transaction mode, each DDL statement specified in the procedure body must be followed by a COMMIT WORK statement. Otherwise, a runtime exception (SQLCODE: 3722, SQLSTATE: ‘T3722’) is raised.

Supported DCL Statements in Stored Procedures

Supported DCL Statements

You can use the following SQL DCL (Data Control Language) statements in a stored procedure when the creator is also the immediate owner of the procedure. That is, a stored procedure can contain DCL statements only if it is created in the creating in the database of the user.

- GIVE
- GRANT LOGON
- REVOKE
- REVOKE MONITOR
- GRANT
- GRANT MONITOR
- REVOKE LOGON

For information on the above DCL statements, see "SQL Data Control Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*.

Transaction Mode Impact on DCL Statements in Stored Procedures

The behavior of DCL statements specified in stored procedures at runtime depends on the transaction mode of the Teradata session in which the procedure is created.

- A DCL statement specified within an explicit (user-defined) transaction in a stored procedure in Teradata transaction mode must be the last statement in that transaction. Otherwise, a runtime exception (SQLCODE: 3932, SQLSTATE: ‘T3932’) is raised.
- When performing a stored procedure in ANSI transaction mode, each DCL statement specified in the procedure body must be followed by a COMMIT WORK statement. Otherwise, a runtime exception (SQLCODE: 3722, SQLSTATE: ‘T3722’) is raised.

Supported DML Statements in Stored Procedures

Supported DML Statements

You can use the following SQL DML statements in a stored procedure:

- ABORT
- BEGIN TRANSACTION
- END TRANSACTION
- CALL
- CLOSE
- COMMIT
- DECLARE CURSOR (Selection form)
- DELETE, including basic/searched form and positioned form
- FETCH
- INSERT
- MERGE-INTO
- OPEN
- ROLLBACK
- SELECT (only in cursors)
- SELECT INTO
- UPDATE, including searched, positioned, and upsert form

For information on the above DML statements, see Chapter 3: “SQL Data Manipulation Language Statement Syntax”.

SQL Operations on Stored Procedures

The following SQL statements perform DML, DDL, Help, and Show operations on stored procedures. You can submit most of these statements from any application on Teradata client utilities or interfaces.

- ALTER PROCEDURE
- CALL
- CREATE PROCEDURE
- DROP PROCEDURE
- RENAME PROCEDURE
- REPLACE PROCEDURE
- HELP PROCEDURE
- HELP ‘SPL …’
- SHOW PROCEDURE

Note: CREATE PROCEDURE and REPLACE PROCEDURE are supported only from ODBC, JDBC, CLIv2 applications and from the QueryMan utility.

From the BTEQ and TeqTalk utilities, you must submit CREATE/REPLACE PROCEDURE statements in the file referenced by the COMPILE command.

Control Statements in Stored Procedures

You can use control statements and control declarations to write a stored procedure.

Control statements give computational completeness to SQL by providing assignment, conditional execution, loop, and branch capabilities to that language.

Control declarations contain the condition handlers and local variables in a stored procedure.

For the list of control statements and control declarations used to write a stored procedure, see "Control Statements" on page 9-39.

Completion Condition and Exception Condition Handlers

Stored procedures support completion condition and exception condition handlers of the CONTINUE and EXIT types, including:

- User-defined SQLSTATE-based condition handlers
- Generic exception condition handler SQLEXCEPTION
- Generic completion condition handlers SQLWARNING and NOT FOUND

For details on different condition handlers, and all terms associated with condition handling, see "DECLARE HANDLER" on page 9-58.

Cursor Declarations

Stored procedures support cursor declarations made in DECLARE CURSOR statements, and in FOR iteration statements inside stored procedures.

All DECLARE CURSOR statements must be specified *after* local variable declarations and *before* any condition handler declarations.

Every DECLARE CURSOR statement in a stored procedure must be terminated with a SEMICOLON character.

See "DECLARE CURSOR (Selection Form)" on page 7-26 and "FOR" on page 9-111 for details.

Rules for Using SQL Statements in Stored Procedures

Introduction

The rules governing the usage of any statement within a stored procedure, including static and dynamic SQL statements, control statements, condition handler, cursor declaration, and local declaration statements, depend on who is the immediate owner of the stored procedure being created or executed.

The immediate owner of a stored procedure is defined as the user or database space where the stored procedure is created. The creator is the user who creates the stored procedure in any database.

WHEN the Creator is ...	THEN ...
also the immediate owner	DDL, DCL statements and dynamic SQL statements are supported within the stored procedure during the procedure creation.
not the immediate owner	DDL, DCL and dynamic SQL statements are not supported within a stored procedure. Specifying such statements results in compilation errors and the stored procedure is not created.

When The Creator Is Also The Owner

The following rules apply to the usage of statements within a stored procedure when the creator is the immediate owner of a stored procedure:

- If any SQL statement specified in the stored procedure references a missing database object, an SPL compilation warning is reported during the procedure creation.

  If the referenced object does not exist when the stored procedure is performed, a runtime exception is reported.

  Note: If the cursor SELECT statement references a missing database object, an SPL compilation error is reported whether or not the creator is the immediate owner of a stored procedure.

- When the object created by an SQL statement inside the stored procedure body already exists, or exists with a different schema, an SPL compilation warning is reported.
- If the creator does not have the required privileges on the objects referenced in the stored procedure, appropriate warnings are reported during stored procedure creation.

If the required privilege does not exist when the stored procedure is performed, a runtime exception is reported.

Note: If the creator does not have the required privileges on the objects referenced in the cursor SELECT statement, an SPL compilation error is reported whether or not the creator is the immediate owner of a stored procedure.

When The Creator Is Not The Owner

The following rule applies to the usage of DML or transaction control statements within a stored procedure by users other than the immediate owner.

Compilation errors are reported and the procedure is not created in the following situations:

- If any SQL statement inside the procedure references a missing database object
- When the object created by an SQL statement inside the procedure body already exists, or exists with a different schema
- If the creator does not have the required privileges on the objects referenced in the stored procedure.

Ownership of Objects Created by Stored Procedures

- The immediate owner of the stored procedure is the creator of permanent objects created through the stored procedure. A volatile table is not a permanent object, and hence an exception.

  Other users executing the stored procedure do not get any automatic rights on the newly created objects. The immediate owner can explicitly grant access rights on the newly created objects to other users.
- If a database object in an SQL statement is not explicitly qualified by database name, the default database at the time of creation of the stored procedure is used to implicitly qualify the object name.

  If a DDL statement is creating the database object, the qualifying database (either implicit or explicit) is the immediate owner of the object created.

Privileges Required for Statement Execution

During stored procedure execution, Teradata checks the privileges of the immediate owner of the procedure for all statements specified and all objects referenced in the stored procedure body.

If a user other than the immediate owner is executing the stored procedure, then the immediate owner must have the required privileges WITH GRANT OPTION.

SQL Statement Errors

Errors and warnings resulting from any statement within the stored procedure body have the following impact:

WHEN This Occurs in Any Statement ...	THEN ...
syntax error	• a compilation error is reported. • the procedure is not created.
any error, when the creator of the stored procedure is not its immediate owner	• a compilation error is reported. • the procedure is not created. **Note**: An exception to this rule is the self-referencing stored procedure. See "Self-Referencing Stored Procedures" on page 9-34.
more than one error	only the first error is reported for that statement.
more than one warning	only the first warning is reported for that statement.
errors and warnings	only the first error is reported for that statement.
compilation warning(s), but no errors	the stored procedure is created with warnings

Unqualified Objects in SQL Statements

During stored procedure execution, the following rules apply to database objects referenced in any DML statement, and not explicitly qualified by a database name.

- An unqualified table reference defaults to the stored procedure's default database. The default database is the default database at the time when the stored procedure is created.

IF ...	THEN ...
no such table exists in the compile-time default database	the system looks for a volatile table with the same name in the login database for the user. • If the volatile table exists, that is accessed. • If the volatile table does not exist, runtime exception 3807 (Table/view/trigger/procedure does not exist) is reported.
the referenced table exists in the default database	the table is accessed, if a volatile table with the same name does not exist in the login database for the user.
	runtime exception 3806 (Table/view/trigger name is ambiguous) is reported, if a volatile table with the same name *also exists* in the login database for the user.

- All unqualified database objects referenced in the statements specified in the stored procedure body, including references to potential volatile tables, are verified from the current default database.

Using Dynamic SQL in Stored Procedures

Introduction

Dynamic SQL is a method of invoking an SQL statement by compiling and performing it at runtime from within a stored procedure. You can invoke DDL, DML or DCL statements, with some exceptions, as dynamic SQL in a stored procedure.

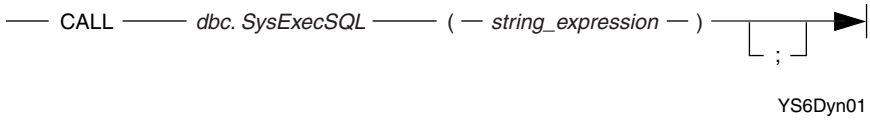
Dynamic SQL

Dynamic SQL statements are those statements whose request text can vary from execution to execution. They provide more usability and conciseness to the stored procedure definition.

Invoking Dynamic SQL

You set up and invoke dynamic SQL in a stored procedure using the following CALL statement within the stored procedure:

Syntax



YS6Dyn01

where:

Syntax element …	Specifies …
dbc.SysExecSQL	the string used for invoking dynamic SQL and validating the user rights. The qualifying database name DBC must be specified, unless the current default database is DBC.
string_expression	any valid string expression to build an SQL statement. The *string_expression* can contain: • string literals • status variables • local variables • input (IN and INOUT) parameters • for-loop aliases

Dynamic SQL Statements That Cannot Be Used In Stored Procedures

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

- ALTER PROCEDURE
- CALL
- CREATE PROCEDURE
- DATABASE
- EXPLAIN
- HELP (all forms)
- REPLACE PROCEDURE
- SELECT
- SELECT - INTO
- SET ROLE
- SET SESSION ACCOUNT
- SET SESSION COLLATION
- SET SESSION DATEFORM
- SET TIME ZONE
- SHOW
- Cursor statements (OPEN, FETCH, CLOSE)

Usage Rules for Dynamic SQL Statements

The following rules apply to the dynamic SQL statements specified in stored procedures:

- Dynamic SQL statements are not validated at compile time, that is, during stored procedure creation. The validation is done only during execution of the stored procedure.
- Multistatement requests cannot be specified as the value of the argument (string expression). If multistatement requests are specified, the error 5568 (SQL statement is not supported within a stored procedure) is reported during stored procedure execution.
- The ending SEMICOLON character is optional in the dynamically built SQL statement.
- The dynamically built SQL statement can
  - be a null statement.
  - contain comments (both Teradata and ANSI style).
  - contain newline and other pad characters.
- You can use only DDL COMMENT statement as dynamic SQL in a stored procedure. You cannot specify DML COMMENT statement to fetch the comments for database objects, columns of a table, and parameters.
- A CREATE DATABASE or CREATE USER statement used as dynamic SQL in a stored procedure must contain the FROM clause.

- The CALL DBC.SysExecSQL statement can be used any number of times in a stored procedure. With each call, only a single SQL statement can be specified in the string expression for dynamic SQL.
- The size of each dynamic SQL request (the *string_expression*) cannot exceed 32000 characters.
- The purpose of the CALL DBC.SysExecSQL statement is to check whether or not the creating user is also the immediate owner of the stored procedure and thus has the right to use dynamic SQL.
- No specific privilege is required to use the CALL DBC.SysExecSQL statement.

Ownership of Objects Created or Referenced Within Dynamic SQL Statements

The rules for objects referenced in, or created through the dynamic SQL statements in a stored procedure are identical to the rules for objects referenced in other statements. See "Ownership of Objects Created by Stored Procedures" on page 9-28.

Example

The following example illustrates the usage of dynamic SQL statements within a stored procedure:

```
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;
```

Self-Referencing Stored Procedures

Introduction

A stored procedure can be recursive, referencing itself directly or indirectly. That is, the stored procedure body can contain a CALL statement invoking the procedure being defined. Such CALL statements can also be nested.

When the stored procedure being created directly references or invokes itself, the procedure is created with an SPL compilation warning (not an error) because the referenced object (the procedure) does not exist. The warning occurs whether or not the creator of the stored procedure is also the immediate owner of the procedure.

This behavior of a self-referencing stored procedure is different from the general norm, where any reference to a missing object results in a compilation error and the stored procedure is not created, if the creator is *not* the immediate owner of the procedure. See "Example 2: Recursion" on page 9-36.

No specific limit exists on the level of recursion, but the stored procedure nesting limit of 15 applies. The limit is further reduced, if there are any open cursors.

Mutual Recursion

You can also create mutually recursive stored procedures, that is, procedures invoking each other in a stored procedure body. This is an indirect recursion. If the creator and the immediate owner of such mutually recursive stored procedures are not the same, an SPL compilation error is reported when an attempt is made to create either of the procedures, because the referenced procedure does not exist.

This error can be avoided by first creating one stored procedure without the CALL statement to the other procedure, then creating the second stored procedure, and then replacing the first procedure with a definition that includes the CALL to the second procedure. See "Example 3: Mutual Recursion" on page 9-37.

Chain Recursion

You can extend the mutual recursion process to have a recursion chain through multiple procedures (A calls B, B calls C and C calls A).

Examples

The first two examples illustrate the creation of a stored procedure that directly references itself. In both these cases, the stored procedure is created with a compilation warning, because the procedure being invoked by the CALL statement does not exist at compilation time. No compilation error is reported in either case.

The third example illustrates the creation of a stored procedure that entails mutual recursion, useful for avoiding any compilation warnings when the creator is the immediate owner of the stored procedure. This is useful for creating a new stored procedure or for changing the parameters of an existing procedure.

Example 1: Recursion

Assume that the table Employee exists, and that the user creating the stored procedure is also its immediate owner.

```
CREATE PROCEDURE spCall(INOUT empcode INTEGER,
                        INOUT basic DECIMAL (6, 2))
BEGIN
   IF (empcode < 1005) THEN
      SELECT empbasic INTO :basic FROM Employee
          WHERE empcode = :empcode ;
      INSERT Temptab(:empcode, :basic);
      SET empcode = empcode + 1;
      CALL spCall(:empcode, :basic);
   END IF;

   IF (empcode = 1005) THEN
      SET empcode = empcode - 1;
      SELECT max(empbasic) INTO :basic from Temptab;
   END IF;
END;
```

When the stored procedure is compiled, the following compilation warning appears, and the procedure spCall is created successfully.

```
SPL5000:W(L8), E(3807):Table/view/trigger/procedure 'spCall'
does not exist.
```

Assume that the stored procedure spCall is invoked the first time as spCall (1001, basic (title 'maximum'));. As the condition in the first IF statement evaluates to true for the values 1001, 1002, 1003 and 1004 passed as the argument for the parameter empcode, the stored procedure invokes itself four times.

Example 2: Recursion

Assume that the table Employee exists. Also assume that the user creating the stored procedure is *not* its immediate owner, but has the requisite privileges. This example illustrates that the reference to a missing object does not result in a compilation error in the case of a self-referencing stored procedure.

```
CREATE PROCEDURE db1.Recur(INOUT empcode INTEGER,
                           INOUT basic DECIMAL (6, 2))
BEGIN
   IF (empcode < 1004) THEN
      SELECT empbasic INTO :basic FROM Employee
          WHERE empcode = :empcode ;
      INSERT Temptab(:empcode, :basic);
      SET empcode = empcode + 1;
      CALL db1.Recur(:empcode, :basic);
   END IF;
END;
```

When the stored procedure is compiled, the following compilation warning appears, and the procedure Recur is successfully created in the database db1.

```
SPL5000:W(L8), E(3807):Table/view/trigger/procedure
'db1.Recur' does not exist.
```

Example 3: Mutual Recursion

Assume that the user U1 is creating the stored procedures. The creator is not the immediate owner of the stored procedures, because both Sp1 and Sp2 are created in the db1 database.

Step	Action

1	Create the first stored procedure Sp1 without recursion.
	`CREATE PROCEDURE db1.Sp1(INOUT p1 INTEGER)`
	`BEGIN`
	`END;`

2	Create the second procedure Sp2 that references the existing procedure db1.Sp1.
	`CREATE PROCEDURE db1.Sp2(INOUT p1 INTEGER)`
	`BEGIN`
	`  IF (p1 > 0) THEN`
	`  CALL db1.Sp1(:p1- 1);`
	`  END IF;`
	`END;`

3	Replace the stored procedure Sp1 with one that references Sp2.
	`REPLACE PROCEDURE db1.Sp1(INOUT p1 INTEGER)`
	`BEGIN`
	`  IF (p1 > 0 ) THEN`
	`  CALL db1.Sp2(:p1 - 1);`
	`  END IF;`
	`END;`

Archiving and Restoring Stored Procedures

You can archive and restore stored procedures only as part of a full database archival and restoration.

Individual stored procedures cannot be archived or restored using the ARCHIVE (DUMP) or RESTORE statements.

Control Statements

This section describes the flow of control statements used in a stored procedure. Control statements include:

- BEGIN - END
- CASE
- DECLARE
- DECLARE HANDLER, including
  - DECLARE HANDLER (CONTINUE Type)
  - DECLARE HANDLER (EXIT Type)
  - DECLARE HANDLER (SQLEXCEPTION Form)
  - DECLARE HANDLER (SQLWARNING Form)
  - DELCARE HANDLER (NOT FOUND Form)
  - DECLARE HANDLER (Control statement Handling)
- FOR
- IF
- ITERATE
- LEAVE
- LOOP
- REPEAT
- SET
- WHILE

All these statements are ANSI SQL-99 compliant.

Important: None of the control statements listed above and described in the sections that follow is valid outside a stored procedure. You *cannot* use any of the above statements interactively or within embedded SQL applications.

BEGIN - END

Purpose

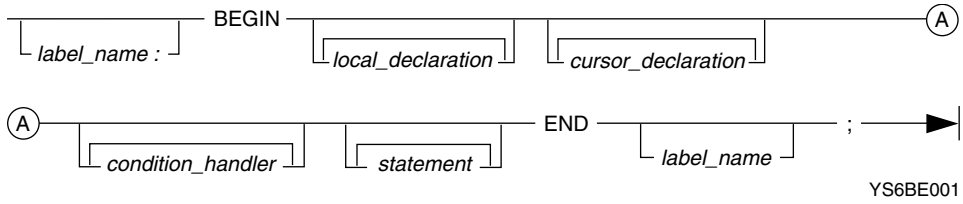
Delimits a compound statement in a stored procedure.

Invocation

Executable.

Stored Procedures only.

Syntax



where:

Syntax element …	Specifies …
label_name	an optional label for the BEGIN - END compound statement. The beginning-label must be terminated by a COLON character (:). An ending-label is not mandatory. But if an ending-label is specified, you must specify an equivalent beginning-label. The label of a BEGIN - END statement cannot be reused for any statement within it. Using label names for each BEGIN - END statement is recommended if you specify nested compound statements in a stored procedure.
local_declaration	a local variable declared using the DECLARE statement. In the case of nested compound statements, variables declared in an outer compound statement can be reused in any inner compound statement. Local variables can be qualified with the label of the compound statement in which the variable is declared. This helps in avoiding conflicts that can be caused by the reuse of local variables in nested compound statements.
cursor_declaration	a cursor declared using the DECLARE CURSOR statement. In the case of nested compound statements, a cursor declared in an outer compound statement can be reused in any inner compound statement.

Syntax element …	Specifies …
condition_handler	a condition handler declared using the DECLARE HANDLER statement. You can use BEGIN - END compound statements inside a *condition_handler* to enclose condition handler action statements.
statement	any one of the following: • DML, DDL or DCL statements supported by stored procedures. These include dynamic SQL statements. • control statements, including BEGIN - END.

ANSI Compliance

BEGIN - END is ANSI SQL-99-compliant.

Authorization

None.

Order of Declarations Within a BEGIN - END Block

In a BEGIN-END compound statement you can specify any number of declarations, and statements to perform the main tasks. All these are optional, but if specified, they must be in the following order within a BEGIN-END block:

1. Local variable declarations. See "DECLARE" on page 9-55.
2. Cursor declarations. See "Cursor Declarations" on page 9-26
3. Condition handler declarations. See "DECLARE HANDLER" on page 9-58 and the subsequent sections.
4. One of the following:
   - a single static or dynamic SQL statement or control statement
   - a compound statement enclosing a list of statements.

   See "Supported DDL Statements in Stored Procedures" on page 9-17.

Declarations of each type should be specified together. They cannot be interspersed with other types of declarations or other statements in the same block.

If compound statements are nested, you can specify the declarations in some, or all, or none of the BEGIN-END blocks. For details on the behavior of condition handlers in nested compound statements, see "DECLARE HANDLER" on page 9-58.

BEGIN - END Rules

The following rules apply to the use of BEGIN - END:

- Stored procedure definitions normally contain one BEGIN - END statement, although this is not mandatory. All other statements of the stored procedure must be specified within this compound statement.
- You can also use a BEGIN - END statement in *condition_handler* declarations to enclose a list of handler action statements.
- BEGIN - END compound statements can be nested. There is no limit on the nesting level.
- Every BEGIN statement must end with the keyword END.
- You can label the BEGIN - END statement. The scope of the label associated with the BEGIN - END statement is the entire statement.

  This includes all nested compound statements and excludes any handlers declared in the compound statement or nested compound statements.
- You can perform stored procedures from within a BEGIN - END statement.
- The scope of the local variables, parameters, and cursors declared in a compound statement is the entire compound statement, including all nested compound statements.
- If a local variable, parameter or cursor name in an inner compound statement is the same as one in an outer compound statement, the local variable, parameter, or cursor name in the inner compound statement takes precedence during execution over the name in the outer compound statement. See "Example 2" on page 9-44.
- Exceptions and completion conditions raised in a compound statement by any statement other than handler action statements are handled within the compound statement.

  If no appropriate handler is available for a condition in an inner compound statement, then that condition is propagated to the outer compound statement in search of a suitable handler. See "DECLARE HANDLER" on page 9-58.
- Exception or completion conditions raised by any statement in the action clause can be handled by a handler defined within the action clause.

  If a condition raised by a handler action is not handled within the action clause, then the condition is not propagated outwards to search for suitable handlers. It remains unhandled:

IF the unhandled condition is ...	THEN ...
an exception	the handler exits and the original condition with which the handler was invoked is propagated outwards to find appropriate handlers.
a completion condition	the condition is ignored and the handler action continues with the next statement.

See "DECLARE HANDLER" on page 9-58 for details.

Labels Referenced in LEAVE and ITERATE

If a label associated with a LEAVE or ITERATE statement inside a labeled BEGIN - END statement refers to the BEGIN - END block label, the following applies:

FOR this statement ...	Performance ...
LEAVE	terminates the BEGIN - END statement with which that label is associated at runtime. Control moves to the next statement following the terminated block. Such termination is treated as successful completion of the stored procedure if the procedure has only one BEGIN - END statement, or if the terminated block is the last statement in the stored procedure body. See "LEAVE" on page 9-129 for details on LEAVE statement.
ITERATE	returns a syntax error when the stored procedure body is parsed during stored procedure creation. See "ITERATE" on page 9-125 for details on ITERATE statement.

Example 1

The following example illustrates a valid stored procedure with nested compound statements.

```
CREATE PROCEDURE spAccount(OUT p1 char(30))
L1: BEGIN
       DECLARE i INTEGER;
     DECLARE DeptCursor CURSOR FOR
      SELECT DeptName from Department;
     DECLARE CONTINUE HANDLER FOR SQLSTATE value '23505'
          L2: BEGIN
      SET p1='Failed To Insert Row';
           END L2;
   L3: BEGIN
   INSERT INTO table_1 VALUES(1,10);
   IF SQLCODE <> 0 THEN LEAVE L1;
   END L3;
    INSERT INTO table_2 VALUES(2,20);
END L1;
```

The procedure body in this example contains a labeled block L1 enclosing a local variable declaration, cursor declaration, condition handler declaration, the nested block labeled L3, and other statements.

The first INSERT statement and the IF statement are part of the inner compound statement labeled L3 and the second is part of the outer block labeled L1.

The BEGIN - END block labeled L2 is a part of the handler declaration.

Example 2

The following example shows the use of an outer compound statement's variable in the inner compound statement by qualifying the variable with the compound statement label.

```
CREATE PROCEDURE spSample1(INOUT IOParam1 INTEGER,
                           OUT OParam2 INTEGER)
L1: BEGIN
   DECLARE K INTEGER DEFAULT 10;
   L2: BEGIN
      DECLARE K INTEGER DEFAULT 20;
      SET OParam2 = K;
      SET IOParam1 = L1.K;
   END L2;
   …
END L1;
```

K is the local variable declared in the outer compound statement L1 and reused in the inner compound statement L2.

After stored procedure execution, the parameter OParam2 takes the default value of K defined in L2, that is 20, because the local declaration of the variable in the inner block takes precedence over the declaration of the same variable in an outer block.

On the other hand, IOParam1 takes the default value of K defined in L1, that is 10, because K is qualified in the second SET statement with the label L1 of the outer compound statement.

Example 3

The following example creates a valid stored procedure with local variable and condition handler declarations. Assume that table1 is dropped before executing this stored procedure.

The INSERT statement in the stored procedure body raises ‘42000’ exception condition, invoking the EXIT handler. The DROP TABLE statement inside the the handler action clause raises another ‘42000’ exception, which is handled by the CONTINUE handler.

```
CREATE PROCEDURE spSample3(OUT p1 CHAR(80))
BEGIN
   DECLARE i INTEGER DEFAULT 20;

   DECLARE EXIT HANDLER
      FOR SQLSTATE '42000'
      BEGIN
         DECLARE i INTEGER DEFAULT 10;
         DECLARE CONTINUE HANDLER
            FOR SQLEXCEPTION
               SET p1 = 'Table does not exist';
         DROP TABLE table1;
         CREATE TABLE table1 (c1 INTEGER);
         INSERT INTO table1 (:i);
      END;

   INSERT INTO table1 VALUES(1000,'aaa');
   /* table1 does not exist */

END;
```

Example 4

The following example shows the valid reuse of local variables and condition handlers for the same SQLSTATE code in non-nested compound statements.

```
CREATE PROCEDURE spSample (OUT po1 VARCHAR(50),
                           OUT po2 VARCHAR(50))
BEGIN
     DECLARE i INTEGER DEFAULT 0;
     L1: BEGIN
         DECLARE var1 VARCHAR(25) DEFAULT 'ABCD';
         DECLARE CONTINUE HANDLER FOR SQLSTATE '42000'
            SET po1 = "Table does not exist in L1';
         INSERT INTO tDummy (10, :var1);
         -- Table Does not exist
     END L1;
     L2: BEGIN
         DECLARE var1 VARCHAR(25) DEFAULT 'XYZ';
         DECLARE CONTINUE HANDLER FOR SQLSTATE '42000'
            SET po2 = "Table does not exist in L2';
         INSERT INTO tDummy (:i, :var1);
         -- Table Does not exist
     END L2;
END;
```

For more details and examples of condition handler behavior in compound statements, see "DECLARE HANDLER" on page 9-58 and subsequent sections.

CASE Statement

Purpose

Provides conditional execution of statements based on the evaluation of the specified conditional expression or equality of two operands.

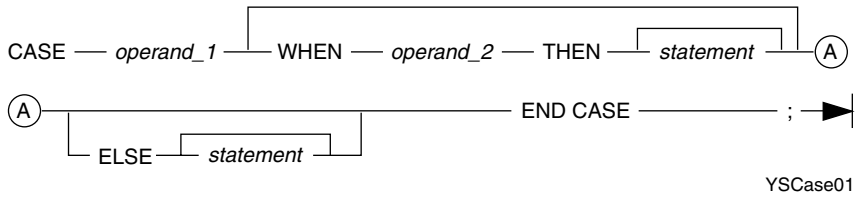
The CASE statement is different from the SQL CASE expression, which returns the result of an expression.

Invocation

Executable

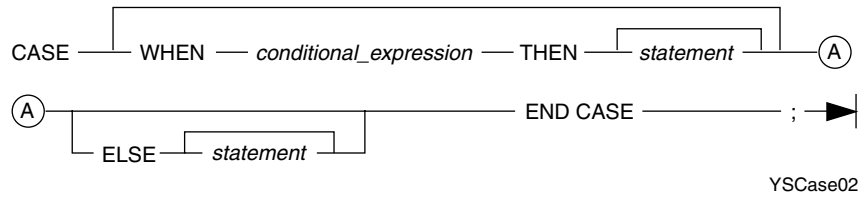
Stored procedures only.

Syntax 1



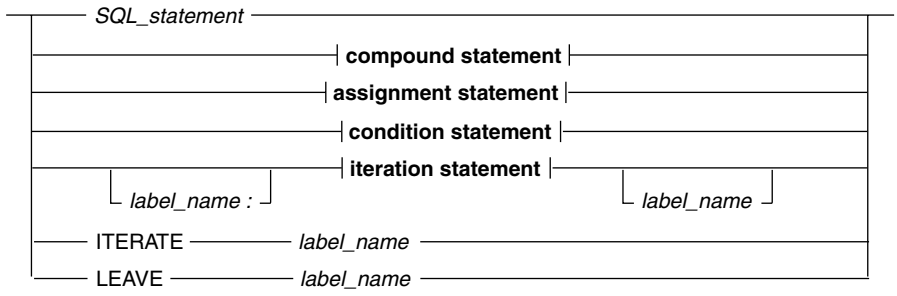
YSCase01

Syntax 2



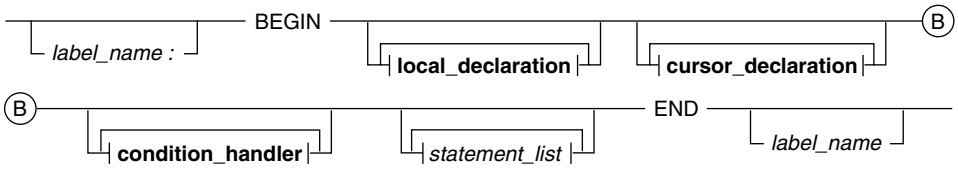
YSCase02

statement



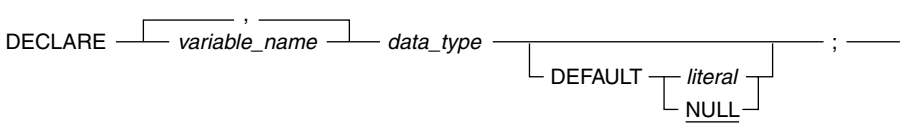
YS6CP01B

compound statement

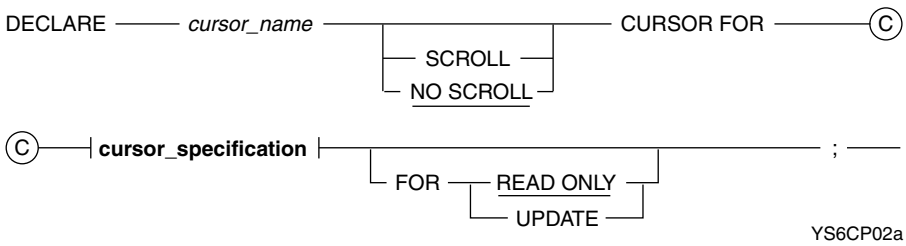


YS6CP01C

local_declaration

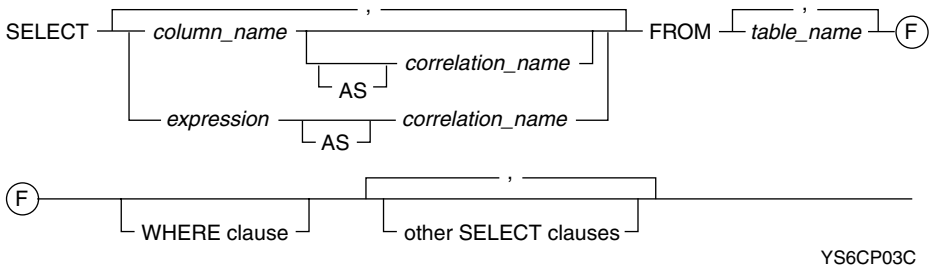


cursor_declaration



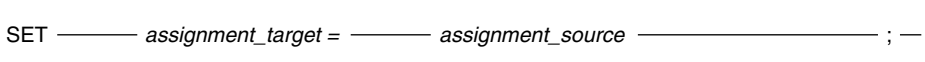
YS6CP02a

cursor-specification



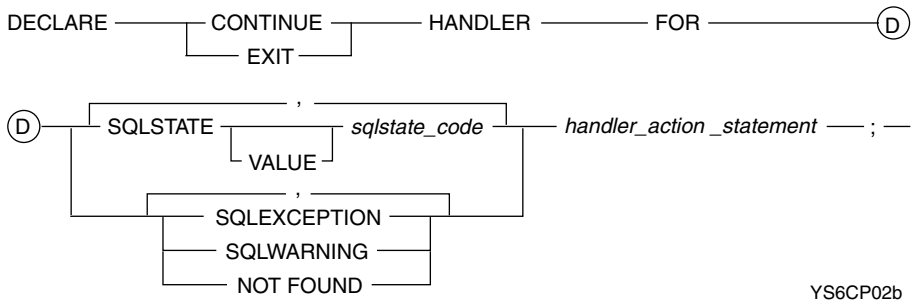
YS6CP03C

assignment statement

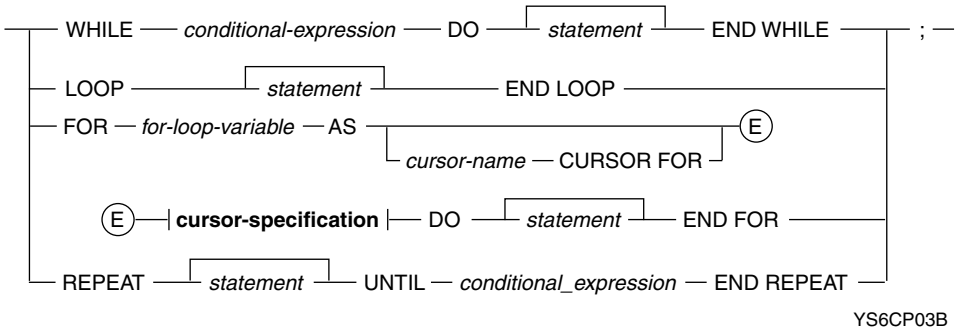


YS6CP02C

condition-handler



iteration statement



where:

Syntax element …	Specifies …
operand_1 *operand_2*	value expressions or arithmetic and string expressions. You can specify stored procedure local variables, status variables, IN or INOUT parameters, literals, and FOR loop column and correlation names in the value expression. OUT parameters and subqueries are not allowed. The data type of *operand_1* and *operand_2* must be compatible with each other.
statement	any of the following: • DML, DDL or DCL statement that can be used in a stored procedure. These include dynamic SQL statements. • control statements, including BEGIN - END.

Syntax element …	Specifies …
conditional_expression	a boolean condition used to determine whether a statement or statements in the THEN clause should be performed. You can specify stored procedure local variables, status variables, IN or INOUT parameters, literals, and FOR loop column and correlation names in the *conditional_expression*. OUT parameters and subqueries are not allowed. You cannot use IN and NOT IN operators if the conditional list contains any local variables, parameters, or cursor aliases.

ANSI Compliance

CASE is ANSI SQL-99-compliant.

Authorization

None.

Two Forms of the CASE Statement

The CASE statement has two forms:

This form …	Provides …
Simple CASE statement	conditional execution of statements based on the equality of the operands. See "Syntax 1" on page 9-46. It tests whether an expression matches one of a number of values, and then branches accordingly.
Searched CASE statement	conditional execution of statements based on the evaluation of a conditional_expression. See "Syntax 2" on page 9-46.

The alternative to using CASE statements is using an IF-THEN-ELSEIF-ELSE statement. See "IF" on page 9-118.

CASE statements are generally preferred when there are more than two conditions or values to be checked.

Simple CASE Statement

In this form of the conditional statement, you can perform a list of SQL statements, including control statements, associated with at most one WHEN clause or ELSE clause, depending on whether *operand_1* (value-expression) equals *operand_2* (value-expression).

The WHEN clauses are evaluated in the order in which they are specified in the CASE statement. The process of evaluation is as follows:

Stage	Process
1	The first WHEN clause is evaluated. • If the value-expression (*operand_1)* specified in the CASE clause is equal to the value-expression (*operand_2)* in the WHEN clause, the statements of that WHEN clause are performed. • Control goes to the next statement in the stored procedure. If the value expressions are *not* equal, then the next WHEN clause, if it exists, is evaluated.
2	All subsequent WHEN clauses are evaluated as described in stage 1.
3	When there are no more WHEN clauses to evaluate, the ELSE clause, if it exists, is taken up and the statements of the ELSE clause are performed. Control goes to the next statement in the stored procedure.
4	If there is no ELSE clause and the value-expression in the CASE clause does not find a match in any of the WHEN clauses, • a runtime exception (“Case not found for CASE statement”, SQLSTATE=’20000’, SQLCODE = 7601) occurs. • the execution of the CASE statement is terminated.

Searched CASE Statement

This form of the CASE statement performs a list of statements when the conditional expression in the WHEN clause evaluates to true. You can perform the statements associated with at most one WHEN clause or ELSE clause.

The WHEN clauses are evaluated in the order in which they are specified in the CASE statement. The process of evaluation is as follows:

Stage	Process
1	The first WHEN clause is evaluated. • If the conditional expression specified in the WHEN clause is true, the statements of that WHEN clause are performed. • Control moves to the next statement in the stored procedure. If the conditional expression is *not* true, then the next WHEN clause, if it exists, is evaluated.
2	All subsequent WHEN clauses are evaluated as described in stage 1.
3	When there are no more WHEN clauses to evaluate, the ELSE clause, if exists, is taken up and the statements of the ELSE clause are performed. Control moves to the next statement in the stored procedure.
4	If there is no ELSE clause and the conditional expression in none of the WHEN clauses evaluates to true, • a runtime exception (“Case not found for CASE statement”, SQLSTATE=’20000’, SQLCODE = 7601) occurs. • the execution of the CASE statement is terminated.

Exception Handling in CASE Statements

If a statement following a WHEN or ELSE clause raises an exception and the stored procedure contains a handler to handle the exception condition, the behavior is identical to exceptions occurring within an IF or WHILE statement.

See “DECLARE HANDLER (Control Statement Handling)” on page 9-105 for examples and rules governing exception conditions.

If the value expression or conditional expression of a CASE statement raises an exception and the stored procedure contains a CONTINUE handler to handle the exception condition, the control moves to the statement following END CASE, after the condition handler action completes successfully.

Example 1: Simple CASE

The following stored procedure includes a simple CASE statement.

```
CREATE PROCEDURE spSample(IN pANo INTEGER,
                          IN pName CHAR(30), OUT pStatus CHAR(50))
BEGIN
  DECLARE vNoOfAccts INTEGER DEFAULT 0;
    SELECT COUNT(*) INTO :vNoOfAccts FROM Accounts;
      CASE vNoOfAccts
          WHEN 0 THEN
             INSERT INTO Accounts (:pANo, :pName);
          WHEN 1 THEN
             UPDATE Accounts
                SET aName = :pName WHERE aNo = :pANo;
          ELSE
             SET pStatus = 'Total ' || vNoOfAccts || 'customer
                                                  accounts';
     END CASE;
END;
```

In the above example, the appropriate SET statement of a WHEN clause is performed depending on the value of the local `vNoAccts`.

IF the value of vNoAccts is ...	THEN it matches ...	AND this statement is performed ...
0	the first WHEN clause	`INSERT INTO Accounts (:pANo, :pName);`
1	the second WHEN clause	`UPDATE Accounts SET aName = :pName WHERE aNo = :pANo;`
any other number	the ELSE clause	`SET pStatus = 'Total ' || vNoAccts || ' customer accounts';`

Example 2: Searched CASE

The following stored procedure includes a searched CASE statement.

```
CREATE PROCEDURE spSample ( IN pANo INTEGER,
                 IN pName CHAR(30), OUT pStatus CHAR(50))
BEGIN
  DECLARE vNoAccts INTEGER DEFAULT 0;
    SELECT COUNT(*) INTO :vNoAccts FROM Accounts;
    CASE
        WHEN vNoAccts = 0 THEN
            INSERT INTO Accounts (:pANo, :pName);
        WHEN vNoAccts = 1 THEN
            UPDATE Accounts
               SET aName = :pName WHERE aNo = :pANo;
        WHEN vNoAccts > 1 THEN
            SET pStatus = 'Total ' || vNoAccts || ' customer
                                                 accounts';
    END CASE;
END;
```

In the above example, the appropriate SET statement of a WHEN clause is performed depending on the value of the local variable `vNoAccts`.

IF the value of vNoAccts is …	THEN the conditional expression in this clause is true…	AND this statement is performed …
0	the first WHEN clause	`INSERT INTO Accounts (:pANo, :pName);`
1	the second WHEN clause	`UPDATE Accounts SET aName = :pName WHERE aNo = :pANo;`
>1	the third WHEN clause	`SET pStatus = 'Total' || vNoAccts || 'customer accounts';`

If the value of vNoAccts is NULL, the stored procedure raises a runtime exception (“Case not found for CASE statement”, SQLSTATE=’20000’, SQLCODE = 7601) in the absence of the ELSE clause. However, vNoAccts cannot be set to NULL by this example.

Example 3: Searched CASE

The following example illustrates the use of FOR loop aliases in the conditional expressions of a searched CASE statement:

```
CREATE PROCEDURE spSample()
Label1:BEGIN
    FOR RowPointer AS
        c_employee CURSOR FOR
        SELECT DeptNo AS c_DeptNo,
            employeeid AS c_empid FROM Employee
    DO
        CASE
     WHEN RowPointer.c_DeptNo > 10 THEN
        INSERT INTO Dept VALUES (:RowPointer.c_DeptNo,
                                 :RowPointer.c_empid) ;
     WHEN RowPointer.c_DeptNo <= 10 THEN
        UPDATE Employee
          SET DeptNo = :RowPointer.c_DeptNo + 10 ;
        INSERT INTO Dept VALUES (:RowPointer.c_DeptNo,
                                  :RowPointer.c_empid) ;
   END CASE;
    END FOR;
END Label1;
```

DECLARE

Purpose

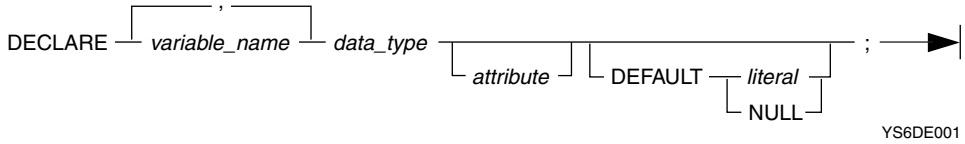
Declares one or more local variables.

Invocation

Nonexecutable control declaration.

Stored procedures only.

Syntax



where:

Syntax element …	Specifies …
variable_name	the name of an SQL local variable to be declared. This must be a valid Teradata SQL name. Reserved words and words reserved as status variable names are not permitted. At run time, you can qualify the variable with the label of the BEGIN - END statement in which the variable is being declared, provided that the statement has a label, as follows: *label.variable_name*
data_type	the data type of the declared local variable. You can specify CHARACTER SET with parameters of character data type after the *data_type* for each parameter. If CHARACTER SET is not specified, the character set will default to the character set of the user creating/compiling the stored procedure. You can also specify CASESPECIFIC or NOT CASESPECIFIC with *data_type*.
literal/NULL	the default value for the local variables. This must be a literal, and compatible with the data type specified. A DEFAULT clause cannot contain an expression. If a default value is specified for a local variable declaration, then that default is assigned to all variables in the list. The variables are initialized to NULL if no default value is specified.

ANSI Compliance

DECLARE is ANSI SQL-99-compliant.

Authorization

None.

Variable Declaration Rules

The following rules apply to the use of local variable declarations:

- You can only declare local variables within a BEGIN - END compound statement.
- You can specify any number of local variable declarations in each BEGIN - END compound statement. Each declaration must end with a SEMICOLON character.
- Within each declaration, you can specify any number of local variables, separated by commas in a list.
- All local variable declarations in a compound statement must be specified *before* any cursor declarations, condition handlers and other statements.
- The scope of a local variable is the BEGIN - END compound statement in which it is declared and all nested compound statements.
- No two variables declared in a compound statement can have the same name.

  A variable name can, however, be reused in any nested compound statement.
- Each local variable declaration consists of the following elements:
  - Local variable name (mandatory)
  - Variable data type (mandatory)
  - Default value for the local variable (optional).

    The default value must be compatible with the data type declared.

Example 1

The declaration is completely specified:

```
DECLARE hErrorMsg CHAR(30) DEFAULT 'NO ERROR';
```

Example 2

Multiple local variables of the same data type can be specified in one declaration statement.

The following declaration declares both `hAccountNo` and `tempAccountNo` to be INTEGER. No default is specified for either variable; therefore NULL is assigned as the default for both.

```
DECLARE hAccountNo, tempAccountNo INTEGER;
```

The following statement declares the data types of `hLastName` and `hFirstName` to be CHAR(30).

```
DECLARE hFirstName, hLastName CHAR(30);
```

Example 3

A default value can be assigned for each local variable specified in a declaration.

In the following example, a default value of 'NO ERROR' is explicitly assigned to `hNoErrorMsg` and `hErrorMsg`:

```
DECLARE hNoErrorMsg, hErrorMsg CHAR(30) DEFAULT 'NO ERROR';
```

DECLARE HANDLER

Purpose

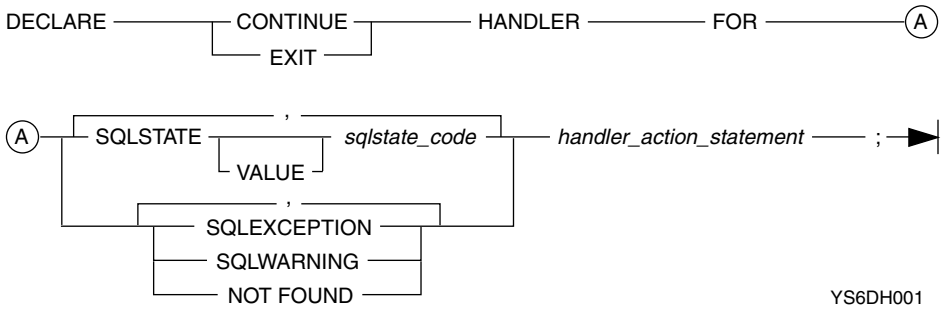
Associates a condition handler with one or more exception or completion conditions to be handled in a stored procedure.

Invocation

Executable control declaration.

Stored procedures only.

Syntax



where:

Syntax element …	Specifies …
CONTINUE EXIT	the type of handler action to be performed.
sqlstate_code	the 5-character literal SQLSTATE code to be handled. See Appendix D: "SQLSTATE Mappings" for a list of SQLSTATE codes and their meanings. You can specify any number of valid SQLSTATE values in a comma-separated list, but '00000' which represents successful completion of statements, is not allowed. You can specify either a list of SQLSTATE values or a list of generic conditions, but not both.

Syntax element …	Specifies …
SQLEXCEPTION SQLWARNING NOT FOUND	generic condition to be handled. You can specify one of these or any combination of the generic conditions in a comma-separated list in a handler declaration. These can be specified in any order, but you cannot repeat any generic condition in the same compound statement.
handler_action_ statement	either a single statement or multiple statements enclosed in a compound statement that define the handler action. The handler action is performed when a particular exception or completion condition is returned to the application. The statement(s) can be any of the following: • SQL DML, DDL, or DCL statements supported by stored procedures. These include dynamic SQL. • Control statements, including nested compound statements. Declaration (local variable, cursor, or handler) statements are not allowed as a single statement for handler action. These can be submitted from within a compound statement.

ANSI Compliance

DECLARE HANDLER is ANSI SQL-99-compliant.

Authorization

None.

Condition Handling: Overview

Benefits of Condition Handling

The principal benefits of stored procedure condition handlers include:

- Reducing error-handling code in the applications by using CALL (to invoke a debugging stored procedure) and by modularizing the error code.
- Trapping exceptions in an application and resolving them in the same execution, without affecting the application.
- Handling most exceptions (which would otherwise cause the stored procedure to terminate) and, in this way, continuing with stored procedure execution.
- Providing a different handling mechanism for different conditions.

Condition Handler Types

Each handler declaration must specify one of two handler types:

- CONTINUE
- EXIT

Both handler types perform the handler action specified by one or more statements.

The difference between the two types is that CONTINUE passes control to the next statement within the compound statement and EXIT passes control to the next statement in the stored procedure outside the compound statement that contains the handler.

The behavior of CONTINUE and EXIT handlers is described in "DECLARE HANDLER (CONTINUE Type)" on page 9-80 and "DECLARE HANDLER (EXIT Type)" on page 9-85.

Condition Handler Forms

Handlers can be either SQLSTATE-based or generic. Generic condition handlers are of three forms:

- SQLEXCEPTION (generic exception condition) handler
- SQLWARNING (generic completion condition) handler
- NOT FOUND (generic "no data found" completion condition) handler

The behavior of these generic condition handlers is described in "DECLARE HANDLER (SQLEXCEPTION Form)" on page 9-92, "DECLARE HANDLER (SQLWARNING Form)" on page 9-97, and"DECLARE HANDLER (NOT FOUND Form)" on page 9-101.

Control Statement Handling

The handling of conditions raised by SQL statements, including control statements, within a stored procedure is described in "DECLARE HANDLER (Control Statement Handling)" on page 9-105.

Terms Associated with Condition Handling

The following terms are associated with condition handling in stored procedures.

Term	Definition
Completion condition	When the performance of an SQL statement, including a control statement, is completed without any fatal event and the Teradata response indicates success or OK with warnings. After completion (other than successful completion) of a request, the SQLCODE contains the return code (warning code), the SQLSTATE is set to a value other than '00000' representing the completion condition and the ACTIVITY_COUNT is set to either "0" or a non-zero value depending on the SQL statement. Examples of completion condition: • an SQL statement, including a control statement, is performed with warnings. • zero rows affected by an UPDATE or DELETE statement. • zero rows returned by a SELECT INTO statement. • no data found on cursor fetch
Condition	Represents an error or informational state caused by execution of an SQL statement, including a control statement. Exception conditions or completion conditions are raised to provide information in the status variables SQLSTATE, SQLCODE and ACTIVITY_COUNT about execution of the SQL statement including a control statement.
Condition handler	A construct defined to perform one or more actions depending on the SQLSTATE value returned to an application. The handler first defines one or more conditions to be handled and then the associated actions. The actions are performed when the corresponding condition occurs during stored procedure execution. If you do not care what particular SQLSTATE code is returned to the stored procedure when an exception condition occurs, you can specify the keyword SQLEXCEPTION instead of one or more specific SQLSTATE codes. SQLEXCEPTION is treated as a generic exception condition handler.
Condition value	An SQLSTATE value which is a 5-character quoted string literal. See definition of SQLSTATE in "Stored Procedure Lexicon" on page 9-8.

Term	Definition
Exception condition	When the execution of an SQL statement, including a control statement, is unsuccessful. The Teradata response indicates an ERROR or FAILURE. After an exception condition is handled, the SQLCODE reflects the return code, the SQLSTATE is set to a value other than '00000' representing the exception condition, and the ACTIVITY_COUNT is set to "0". Examples of exception condition include: • non-valid cursor state • divide by zero violation • string truncation (only in ANSI transaction mode) • cardinality violation
Generic condition handler	Handler declared to handle generic conditions, represented by the keywords SQLEXCEPTION, SQLWARNING, or NOT FOUND. These keywords are declared instead of one or more specific SQLSTATE codes. SQLEXCEPTION represents all exception conditions. SQLWARNING represents all completion conditions, excepting successful completion and "no data found" completion conditions. NOT FOUND represents all "no data found" completion conditions.
Successful completion	When the Teradata response to the execution of an SQL statement indicates success or "ok" without any warning or other non-fatal event. After successful completion of a request, the SQLSTATE is set to '00000', SQLCODE is set to "0" and the ACTIVITY_COUNT is set to either "0" or a non-zero value depending on the SQL statement. The status variable values are unchanged for a control statement.

SQLSTATE

SQLSTATE is a status variable to which SQL exception and completion conditions are posted. SQLSTATE is used both by embedded SQL applications and by stored procedures to reflect the execution status of a statement.

SQLSTATE codes represent successful completion and exception (error or failure) conditions. The keyword SQLEXCEPTION is used to represent all exception SQLSTATE values.

For more information about SQLSTATE, see "SQLSTATE" on page 8-2.

SQLSTATE codes and their mappings to Teradata error codes are described in Appendix D: "SQLSTATE Mappings."

Condition Handler Rules

The following rules apply to condition handlers:

- You can declare condition handlers for exception conditions and completion conditions other than successful completion.
- You can declare condition handlers only within a compound statement.

  No handlers can be declared in stored procedures that do not contain a compound statement.
- No condition handler can be defined for successful completion (SQLSTATE = '00000')
- You cannot repeat the same SQLSTATE code within the FOR clause of a DECLARE HANDLER statement.
- You cannot declare the same SQLSTATE code for multiple condition handlers in a compound statement.

  However, the same SQLSTATE code can be reused for condition handlers in other nested or non-nested compound statements within a stored procedure. See "Example 3" on page 9-68.
- You can specify either SQLEXCEPTION, or SQLWARNING, or NOT FOUND, or any combination of these generic conditions in a handler declaration.
- You can declare each generic condition handler at most once in a compound statement.

  The same generic condition can be reused in other compound statements within a stored procedure.
- You cannot declare a specific SQLSTATE value and one or more generic conditions within the same DECLARE HANDLER statement.
- When you specify multiple statements for handler action, all the statements must be contained within a BEGIN - END compound statement.

  You can submit nested compound statements for handler action.
- The scope of a condition handler is the compound statement in which it is declared, including all nested compound statements.

Condition Handlers in Nested Compound Statements

The following bullets list the rules for using condition handlers within nested compound statements:

- Exceptions and completion conditions raised in a compound statement by any statement other than handler action statements are handled within that compound statement if an appropriate handler exists.

  In nested compound statements, conditions that find no suitable handler in an inner compound statement are propagated to the outer statement in search of a handler.

The different scenarios possible when no handler is available to handle a particular condition in an inner compound statement, are described in the following table:

<table>
<tr><th>IF a condition is raised ...</th><th>AND an appropriate handler...</th><th>THEN ...</th></tr>
<tr><td rowspan="2">in a non-nested compound statement</td><td>exists in that statement</td><td>the condition is handled and the stored procedure execution either continues or terminates based on the type of handler.</td></tr>
<tr><td colspan="2">does not exist:
<table><tr><th>AND if the condition is...</th><th>THEN ...</th></tr><tr><td>an exception</td><td>the stored procedure terminates.</td></tr><tr><td>a completion condition</td><td>the stored procedure execution continues.</td></tr></table></td></tr>
<tr><td rowspan="2">• in a nested compound statement, or
• by a statement other than a handler action statement</td><td>exists within that statement</td><td>the condition is handled.</td></tr>
<tr><td colspan="2">does not exist within that statement, then the immediate outer statement is searched for a suitable handler.
• If a handler exists within that statement, the condition is handled.
• If no handler exists, the next outer compound statement is searched for a suitable handler.
If no appropriate handler exists in the outermost compound statement:
<table><tr><th>AND if the condition is...</th><th>THEN ...</th></tr><tr><td>an exception</td><td>the stored procedure terminates.</td></tr><tr><td>a completion condition</td><td>the stored procedure execution continues with the statement following the statement that caused the condition.</td></tr></table></td></tr>
</table>

- The rules for propagation and handling of exception or completion conditions raised by any handler action statement are different from the above. See details and examples under "Conditions Raised by a Handler Action" on page 9-69.

Status Variable Values

On completion of a handler action, status variables are set to the values indicated by the following table:

Status Variable	Status Code
SQLSTATE	'00000'
SQLCODE	0
ACTIVITY_COUNT	0

These values are returned only if the handler is of CONTINUE type. In the case of EXIT handler, control exits the compound statement in which the condition is raised.

If the evaluation of an expression within a stored procedure raises an exception or completion condition then the values for the status variables SQLSTATE, SQLCODE, and ACTIVITY_COUNT are all set to values corresponding to the particular warning, error or failure code returned by Teradata. An example is a divide-by-zero condition.

Successful evaluation of an expression does not raise a completion condition.

Precedence of Specific Condition Handlers

In a stored procedure containing specific SQLSTATE condition handlers and a generic condition handler to handle similar conditions, the specific condition handlers take precedence. The following rules apply to such situations.

- When both SQLEXCEPTION and specific SQLSTATE handlers for exception conditions are specified in a stored procedure.

IF a raised exception ...	THEN this handler action performs ...
matches the SQLSTATE value specified for a handler	the action defined for the condition handler.
does not match a specific SQLSTATE code specified for any handler	the action defined for the generic condition handler.

- When both SQLWARNING and specific SQLSTATE handlers for completion conditions are specified.

IF a raised condition ...	THEN this handler action performs ...
matches any of the SQLSTATE values specified for a handler	the action defined for the specific condition handler.
does not match a specific SQLSTATE code specified for any handler	the action defined for the generic completion condition handler.

- When both NOT FOUND and specific SQLSTATE handlers for "no data found" completion condition are specified.

IF a "no data found" condition occurs and if the SQLSTATE value…	THEN this handler action performs …
matches any of the specific completion condition handlers	the action defined for the specific completion condition handler.
does not match any specific handler	the action defined for the generic NOT FOUND condition handler.

- The completion condition "no data found" has precedence over the other completion conditions. Only a generic NOT FOUND handler or a specific handler can handle the "no data found" completion condition.

Exception Condition Transaction Semantics

CONTINUE and EXIT exception conditions are governed by the same set of rules. Stored procedure behavior is consistent with the transaction semantics of ANSI and Teradata transaction modes in general.

The following table describes the transaction semantics for error and failure conditions, respectively.

FOR this condition type …	AND this transaction mode …	This action is taken by the Transaction Manager …
ERROR	ANSI	a request-level rollback. Only the statement that raised the exception condition is rolled back.
FAILURE	ANSI Teradata	a transaction-level rollback. All updates performed within the transaction are rolled back.

Example 1

This example illustrates how a CONTINUE handler associated with an outer compound statement handles an exception condition raised in an inner compound statement.

```
CREATE PROCEDURE spSample1(IN pName CHAR(30), IN pAmt INTEGER)
BEGIN
   DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
      INSERT INTO Proc_Error_Tbl VALUES (:SQLSTATE,
        CURRENT_TIMESTAMP, 'spSample1', 'Duplicate Row Error');
      ...
      L1: BEGIN
      DECLARE counter INTEGER DEFAULT 5;
      DECLARE CONTINUE HANDLER FOR SQLSTATE '42000'
        L2: BEGIN
           INSERT INTO Proc_Error_Tbl VALUES (:SQLSTATE,
                       CURRENT_TIMESTAMP, 'spSample1',
                                   'Table does not exist');
        ...
        END L2;
      WHILE (counter < 1022012) DO
        INSERT INTO tab1 VALUES (:pName, :pAmt) ;
         -- Duplicate row error
              SET counter = counter + 1;
      END WHILE;
      ...
      END L1;
      ...
END;
```

Assume that table tab1 is created as follows:

```
CREATE SET TABLE tab1(c1 CHAR(30), c2 INTEGER);
```

Now perform the stored procedure spSample1.

```
CALL spSample1('Richard', 100);
```

The INSERT within the WHILE statement in L1 raises a duplicate row exception. Since there is no handler within the compound statement L1 to handle this exception, it is propagated to the outer compound statement, which has no label.

The CONTINUE handler declared for SQLSTATE ’23505’ is invoked. This handler handles the exception condition and the stored procedure execution continues from the statement following the INSERT that raised the condition.

Example 2

This example illustrates how an exception raised in an inner compound statement remains unhandled in the absence of a suitable handler in the entire stored procedure.

```
CREATE PROCEDURE spSample1(IN pName CHAR(30), IN pAmt INTEGER)
L1: BEGIN
...
   L2: BEGIN
      DECLARE vName CHAR(30);
      INSERT INTO tab1 VALUES (:pName, :pAmt);
      -- The table does not exist
      -- exception is raised, and not handled
      SET vName = pName;
   END L2;
   INSERT ...
END L1;
```

Assume that table tab1 is dropped:

```
DROP TABLE tab1;
```

Now perform the stored procedure spSample1.

```
CALL spSample1('Richard', 10000);
```

During stored procedure execution, the first INSERT statement raises an exception with SQLSTATE '42000' but there is no handler defined to handle this exception in the compound statement labeled L2.

No handler is defined for the raised exception even in the outer compound statement labeled L1, so the stored procedure terminates with the error code corresponding to SQLSTATE '42000'.

Example 3

The following example shows the valid reuse of the same SQLSTATE code in nested compound statements.

Assume that the table tDummy is dropped before executing the stored procedure. The same kind of exception condition occurs in the compound statements labeled L1 and L2 and the condition is handled on both occasions.

The stored procedure is created with two compilation warnings if the creator is also the immediate owner of the procedure. If the creator is not the owner, the procedure is not created and two errors are reported.

```
CREATE PROCEDURE spSample (OUT po1 VARCHAR(50),
                           OUT po2 VARCHAR(50))
BEGIN
     DECLARE i INTEGER DEFAULT 0;
     L1: BEGIN
         DECLARE var1 VARCHAR(25) DEFAULT 'ABCD';
         DECLARE CONTINUE HANDLER FOR SQLSTATE '42000'
             SET po1 = "Table does not exist in L1';
         INSERT INTO tDummy (10, :var1);
        -- Table does not exist.
         L2: BEGIN
             DECLARE var1 VARCHAR(25) DEFAULT 'XYZ';
             DECLARE CONTINUE HANDLER FOR SQLSTATE '42000'
                 SET po2 = "Table does not exist in L2';
             INSERT INTO tDummy (:i, :var1);
        -- Table does not exist.
         END L2;
    END L1;
END;
```

Conditions Raised by a Handler Action

The action clause of a condition handler can be a nested or non-nested compound statement. Exception or completion conditions raised by any statement in the action clause can be handled by a handler defined within the action clause.

If a condition raised by a handler action is not handled within the action clause, then that condition is not propagated outwards to search for suitable handlers. Other handlers associated with the compound statement cannot handle the condition raised by any handlers. Such conditions remain unhandled. The following table compares unhandled exception and completion conditions:

IF the unhandled condition is ...	THEN ...
an exception	the handler exits and the original condition with which the handler was invoked is propagated outwards to find appropriate handlers. If no suitable handler exists for the original condition, the stored procedure terminates.
a completion condition	the condition is ignored and the execution continues with the next statement in the handler action.

These situations are illustrated in the following cases and examples.

Case 1

Consider the following case of a handler action for an exception condition raising a new exception, which is handled.

Stage	Process
1	The stored procedure raises an exception with the SQLSTATE code '42000', which means the referenced database object does not exist.
2	The exception condition is handled by a condition handler.
3	Assume that the handler action raises the divide by zero exception '22012'.
4	A handler exists within the handler action group of statements to handle this exception, and it is handled.

Case 2

Consider the following case of a handler action for an exception condition raising a new exception, which is not handled.

Stage	Process
1	The stored procedure raises an exception with the SQLSTATE code '42000', which means the referenced database object does not exist.
2	Assume that the handler action clause raises the divide by zero exception '22012'.
3	If a suitable handler for this newly raised exception does not exist within the handler action group of statements, then the newly raised condition is not propagated outside to search for handlers.
4	The handler action exits, and the original exception condition '42000' is propagated outwards in search of a suitable condition handler.
5	If no suitable handler is found for the original condition, the stored procedure terminates and returns a Teradata error code corresponding to the original exception condition '42000'.

Case 3

Consider the following case of a handler action for a completion condition raising an exception.

Stage	Process
1	The stored procedure raises a completion condition (a warning) with the SQLSTATE code 'T7473', which means the requested sample size is larger than table rows.
2	Assume that the handler action raises an exception condition '23505' for attempting to insert a duplicate row in table.

Stage	Process
3	If a suitable handler for ’23505’ exists within the handler action, then the condition is handled.
4	If a suitable handler for '23505' does not exist within the handler action, the original condition 'T7473' is propagated outward to look for a subtable handler to handle the condition.
5	If the original completion condition is handled and: • if the handler is of CONTINUE type the stored procedure execution continues with the statement that raised the completion condition. • if the handler is of EXIT type control exits the compound statement that contains the EXIT handler. If the completion condition is not handled, the stored procedure execution continues with the next statement.

Case 4

Consider the following case of a handler action for a completion condition raising another completion condition.

Stage	Process
1	The stored procedure raises a completion condition ’T7473’, which means the requested sample size is larger than table rows.
2	The completion condition is handled by a generic condition handler.
3	Assume that the handler action raises a "no data found" completion condition.
4	This new completion condition is ignored and the stored procedure execution continues with the remaining statements.

Rules for Reporting Handler Action-Raised Conditions

An important aspect of case 3 and case 4 is how conditions raised by a handler action associated with the completion of an SQL statement are reported to the calling stored procedure or application.

- If a raised condition is handled without exceptions, then the status variables are set to reflect the successful completion condition. No information about the raised condition is sent to the caller. Thus, if a failure occurs in a stored procedure and it is handled, the caller is not aware of the occurrence of failure or the transaction rollback.

  An appropriate mechanism like application rules must be defined to get information about the occurrence of the condition.

- If a statement within the handler action associated with a completion condition handler raises a completion condition other than successful completion, and if there is no suitable handler for that condition, the execution continues with the next statement inside the handler action clause. The status variables contain the values reflecting the original completion condition.

  On completion of the handler action, the status variables are set to reflect the successful completion of the handler action.
- If the possibility of a handler action clause raising an exception condition is known, a suitable handler can be placed inside the handler action clause, while creating the stored procedure, to handle such exceptions. The handlers can be nested as deep as necessary.

Example 1

In this example, an exception raised by a handler remains unhandled, and the original condition that invoked the handler is propagated outwards:

```
CREATE PROCEDURE spSample2(IN pName CHAR(30), IN pAmt INTEGER)
BEGIN
DECLARE EXIT HANDLER FOR SQLSTATE '23505'
   INSERT INTO Error_Tbl VALUES (:SQLSTATE,CURRENT_TIMESTAMP,
          'spSample2', 'Failed to insert record');
   ...
   L1:BEGIN
      DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
      BEGIN
         INSERT INTO Proc_Error_Tbl VALUES (:SQLSTATE,
            CURRENT_TIMESTAMP, spSample2',
            'Failed to Insert record');
      END;
      INSERT INTO tab1 VALUES (:pName, :pAmt);
      INSERT INTO tab1 VALUES (:pName, :pAmt);
      -- Duplicate row error
      ...
   END L1;
    ...
END;
```

Assume that the table tab1 is created as follows:

```
CREATE SET TABLE tab1(c1 CHAR(30), c2 INTEGER);
```

Drop the table Proc_Error_Tbl:

```
DROP TABLE Proc_Error_Tbl;
```

Now perform the procedure spSample2:

```
CALL spSample2('Richard', 100);
```

During stored procedure execution, the last INSERT statement of the compound statement L1 raises a duplicate row exception. The CONTINUE handler declared for SQLSTATE ’23505’ is invoked. The handler action statement (INSERT) results in another exception '42000'.

Since there is no handler within this handler to handle SQLSTATE '42000', the original condition with which the handler was invoked, SQLSTATE '23505', is propagated outwards. The outer compound statement has an EXIT handler defined for SQLSTATE '23505'. This handler handles the exception and control exists the compound statement. Because the procedure contains no other statement, the procedure terminates.

Example 2

In this example, a completion condition raised within a handler is ignored.

```
CREATE PROCEDURE spSample1(IN pName CHAR(30), IN pAmt INTEGER)
BEGIN
   DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
   BEGIN
      DELETE FROM temp_table;
      INSERT INTO Proc_Error_Tbl VALUES (:SQLSTATE,
         CURRENT_TIMESTAMP, ’spSample1',
         'Failed to Insert record');
   END;
   INSERT INTO tab1 VALUES (:pName, :pAmt);
   INSERT INTO tab1 VALUES (:pName, :pAmt);
     -- duplicate row error
    ...
END;
```

Assume that the tables temp_table, and tab1 are defined as follows:

```
CREATE TABLE temp_table(c1 INTEGER, c2 CHAR(30));
CREATE SET TABLE tab1(c1 CHAR(30), c2 DECIMAL(18,2));
```

Now perform the procedure:

```
CALL spSample1('Richard', 10000);
```

The last INSERT statement raises a duplicate row exception and the CONTINUE handler declared for this error is invoked. The DELETE statement in the handler action clause results in a "no data found" completion condition.

Since there is no handler defined within the handler to handle this condition, the condition is ignored and the stored procedure execution continues from the next statement (INSERT) in the handler action clause.

Example 3

This example combines conditions raised by statements within and outside a handler action clause, and shows how an exception raised by a handler action remains unhandled.

```
REPLACE PROCEDURE han1(INOUT IOParam1 INTEGER, INOUT IOParam2
                                                    CHAR(20))
Loutermost: BEGIN
   DECLARE Var1 INTEGER DEFAULT 10;
   L1: BEGIN
      DECLARE EXIT HANDLER FOR SQLSTATE '42000'

         -- Statement 3_1a
         SET IOParam2 = 'Table does not exist in the outer
                                                 block';
      DECLARE EXIT HANDLER FOR SQLSTATE '23505'
         L2: BEGIN
         DECLARE EXIT HANDLER FOR SQLSTATE '23505'

              -- Statement 3_2a
            SET IOParam2 = ' Duplicate row error ';
         DECLARE EXIT HANDLER
            FOR SQLSTATE '42000'
            BEGIN

           -- Statement 3_3a
            SET IOParam2 = 'Non-existent table in inner block ';
           -- Statement 3_3b
            INSERT INTO tab1 VALUES (:IOParam1);
                     -- duplicate row error
            END;
           -- Statement 3_3c
            INSERT INTO notable VALUES (:IOParam1, :IOParam2);
                                                 -- 42000
         END L2; /* End Label L2 */
      -- Statement 3_4a
      DELETE tab1 ALL;
      -- Statement 3_4b
      SET IOParam1 = Var1;
      -- Statement 3_4c
      INSERT INTO tab1 VALUES (:IOParam1);
      -- Statement 3_4d
      INSERT INTO tab1 VALUES (:IOParam1);
                  -- duplicate row error
   END L1; /* End Label L1 */
END Loutermost;
```

During stored procedure execution, the INSERT statement (Statement 3_4d) raises a duplicate row exception. The first EXIT handler declared for SQLSTATE ’23505’ is invoked because the handler is in the same compound statement labeled L1.

Then the Statement 3_3c in L2 raises an exception with SQLSTATE '42000'. The EXIT handler defined for ’42000’ is invoked to handle this exception. The INSERT statement (Statement 3_3b within the handler) raises a duplicate row exception. Since there is no handler within the handler to handle this new condition, the handler exits.

The original condition corresponding to SQLSTATE '23505', which invoked the outermost handler, is propagated outwards. Since there is no suitable handler for that in the outermost compound statement Loutermost, the stored procedure terminates with the error corresponding to '23505'.

Example 4: Condition Handlers In Nested Stored Procedures

The example in this section is based on the following stored procedure:

```
CREATE PROCEDURE spSample7a()
BEGIN
    DECLARE hNumber INTEGER;

     -- Statement_7a_1
    UPDATE Employee SET Salary_Amount = 10000
    WHERE Employee_Number = 1001;

    -- Statement_7a_2
    INSERT INTO EmpNames VALUES (1002, 'Thomas');

     -- Statement_7a_3
     UPDATE Employee
     SET Salary_Amount = 10000
     WHERE Employee_Number = 1003;
END;
```

If the EmpNames table had been dropped, Statement_7a_2 in the above stored procedure returns an error with SQLSTATE code ’42000’ that is not handled because no condition handler is defined for it.

Note that the second procedure calls the first procedure at Statement_7b_2.

Consider a second stored procedure definition:

```
CREATE PROCEDURE spSample7b()
BEGIN
    DECLARE hNumber INTEGER;
    DECLARE EXIT HANDLER FOR SQLSTATE '42000'
     INSERT INTO Proc_Error_Table
            (:SQLSTATE, CURRENT_TIMESTAMP, 'spSample7b',
                        'Failed to Insert Row');

  -- Statement_7b_1
  SELECT nextEmpNum INTO :hNumber
      FROM EmpNext;
  UPDATE Employee
      SET nextEmpNum = :hNumber+1;

  -- Statement_7b_2
  CALL spSample7a();

  -- Statement_7b_3
  UPDATE Employee SET Salary_Amount = 10000
  WHERE Employee_Number = 1003;
END;
```

Example 5: ANSI Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI transaction mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
     WHERE Employee_Number = 1000;

CALL spSample7b();
```

When the above three SQL statements are invoked in ANSI transaction mode, the following occurs:

Stage	Process
1	The stored procedure statement marked as Statement_7b_2 calls stored procedure spSample7a.
2	Statement_7a_2 in stored procedure spSample7a raises an exception with SQLSTATE code '42000'.
3	Control returns to the calling procedure, spSample7b, along with the exception because there is no condition handler defined in spSample7a.
4	The exception is handled in spSample7b and the handler action is performed.

Stage	Process
5	Control exits the calling compound statement because the handler type is EXIT.
6	The following items are left uncommitted: • The first two interactive SQL statements • Statement_7a_1 from spSample7a • Statement_7b_1 from spSample7b • The INSERT statement from the condition handler in spSample7b. The following items are not performed: • Statement_7a_3 from spSample7a • Statement_7b_3 from spSample7b

Multiple Condition Handlers In a Stored Procedure

The example in this section is based on the following stored procedure:

```
CREATE PROCEDURE spSample10()
BEGIN
  DECLARE EmpCount INTEGER;
  DECLARE CONTINUE HANDLER
   FOR SQLSTATE '42000'
   H1:BEGIN

   -- Statement_10_1
   UPDATE Employee
   SET Ename = 'John';

   -- Suppose column Ename has been dropped.
   -- Statement_10_1 returns SQLSTATE code '52003' that is
   -- defined for the handler within the
   -- block that activates this handler.

   -- Statement_10_2
   INSERT INTO Proc_Error_Table (:SQLSTATE,
      CURRENT_TIMESTAMP, 'spSample10', 'Failed to
         Insert Row');
   END H1;

DECLARE EXIT HANDLER
   FOR SQLSTATE '52003'
INSERT INTO Proc_Error_Table (:SQLSTATE,
  CURRENT_TIMESTAMP, 'spSample10', 'Column does not exist');
```

```
DECLARE CONTINUE HANDLER
  FOR SQLWARNING
INSERT INTO Proc_Error_Table (:SQLSTATE,
 CURRENT_TIMESTAMP, 'spSample10', 'Warning has occurred');

DECLARE CONTINUE HANDLER
  FOR NOT FOUND
INSERT INTO Proc_Error_Table (:SQLSTATE,
 CURRENT_TIMESTAMP, 'spSample10', 'No data found');

-- Statement_10_3
UPDATE Employee
  SET Salary_Amount = 10000
  WHERE Employee_Number = 1001;

-- Statement_10_4
INSERT INTO EmpNames VALUES (1002, 'Thomas');

-- Suppose table EmpNames has been dropped.
-- Statement_10_4 returns SQLSTATE '42000' that is
-- handled.

-- Statement_10_5
UPDATE Employee
 SET Salary_Amount = 10000
 WHERE Employee_Number = 1003;

-- Statement_10_6
SELECT COUNT(*) INTO :EmpCount FROM Employee SAMPLE 5;

-- Suppose table Employee has only three rows.
-- Statement_10_6 returns SQLSTATE 'T7473' that is
-- handled by SQLWARNING handler.

-- Statement_10_7
DELETE Employee WHERE Employee_Number = 1;

-- Suppose table Employee does not have a row for
-- Employee_Number = 1. Statement_10_7 returns SQLSTATE
-- '02000' that is handled by NOT FOUND handler.

END;
```

Example 6: ANSI Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI transaction mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee
   SET Salary_Amount = 10000
   WHERE Employee_Number = 1000;

CALL spSample10();
```

When the above three SQL statements are invoked in ANSI transaction mode, the following occurs:

Stage	Process
1	Statement_10_4 in the called stored procedure raises an exception with SQLSTATE code '42000' that is handled using a CONTINUE handler.
2	While performing the handler action for SQLSTATE '42000', Statement_10_1 raises an exception with SQLSTATE code '52003'. Because an exception raised by a handler cannot be handled outside the handler action clause, control does not pass to the handler for SQLSTATE code '52003'.
3	The procedure terminates and returns the original SQLSTATE code '42000' to the caller.
4	The following statements are not performed: • Statement_10_2 • Statement_10_5 • Statement_10_6 • Statement_10_7
5	The following statements remain active in a transaction that is not yet committed: • The first two interactive SQL statements • Statement_10_3

DECLARE HANDLER (CONTINUE Type)

Introduction

CONTINUE handlers are useful for handling completion conditions and exception conditions not severe enough to affect the flow of control.

Actions Taken by a CONTINUE Handler

When a condition is raised, a CONTINUE handler does the following:

1. Performs the handler action.
2. Passes control to the next statement following the statement that invoked it.
3. Performs all remaining SQL statements following the statement that raised the condition.

The following table describes the detailed flow of control for a CONTINUE handler when it is activated by a raised exception:

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>A stored procedure statement raises an exception or completion condition.</td></tr>
<tr><td>2</td><td>The CONTINUE handler written to handle that exception or completion condition performs its designated action.
<table>
<tr><th>IF ...</th><th>THEN in the next stage, control ...</th></tr>
<tr><td>the handler action completes successfully</td><td rowspan="2">returns to the statement following the statement that raised the condition.</td></tr>
<tr><td>the exception was raised by a statement embedded within a control statement such as FOR, IF, LOOP, or WHILE</td></tr>
<tr><td>a control statement raises an exception (for example, while evaluating a conditional expression)</td><td>passes to the statement following the control statement that raised the condition.</td></tr>
</table></td></tr>
<tr><td>3</td><td>If a handler action raises an exception or completion condition, and if a suitable handler exists within that handler action, the newly raised condition is handled.
Control returns to the handler action clause.</td></tr>
</table>

Examples of a CONTINUE Handler

The following examples illustrate the behavior of a CONTINUE handler. The examples are based on the following stored procedure:

```
CREATE PROCEDURE spSample4()
BEGIN
  DECLARE hNumber INTEGER;
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
   INSERT INTO Proc_Error_Table
    (:SQLSTATE, CURRENT_TIMESTAMP, 'spSample4',
     'Failed to Insert Row');

-- Statement_4_1
UPDATE Employee
  SET Salary_Amount = 10000
  WHERE Employee_Number = 1001;

-- Statement_4_2
INSERT INTO EmpNames VALUES (1002, 'Thomas');

-- If the EmpNames table had been dropped, Statement_4_2
-- returns SQLEXCEPTION that is handled.

-- Statement_4_3
UPDATE Employee
  SET Salary_Amount = 10000
  WHERE Employee_Number = 1003;
END;
```

Example 1: ANSI Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI transaction mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
  WHERE Employee_Number = 1000;

CALL spSample4();
```

If an SQL statement reports either an error condition or a failure condition such as deadlock in ANSI transaction mode, the condition is handled using a condition handler.

When the above three SQL statements are invoked in ANSI transaction mode, the following process table describes the impact of an error condition with respect to them:

Stage	Process
1	The stored procedure statement marked as Statement_4_2 raises an exception with the SQLSTATE code '42000'. The request is rolled back.
2	"The handler is invoked for the '42000' condition."
3	Because this handler type is CONTINUE, control passes to Statement_4_3 after the handler action completes.
4	The following items are left uncommitted: • The first two interactive SQL statements • Statement_4_1 • Statement_4_3 • The INSERT statement from the condition handler in spSample4.

When the above three SQL statements are invoked in ANSI transaction mode, the following process table describes the impact of a failure condition with respect to them:

Stage	Process
1	The stored procedure statement marked as Statement_4_2, which is invoked by the CALL spSample4() statement, returns an SQLSTATE code that indicates a failure condition.
2	The effects of Statement_4_1 and of the first two interactively entered SQL statements are rolled back and the transaction is rolled back.
3	The returned SQLSTATE code invoked the CONTINUE handler defined for the block, which is written to handle that specific condition (failure in ANSI transaction mode).
4	Because the handler type is CONTINUE, the stored procedure submits the handler action statements and Statement_4_3 in a new transaction and the stored procedure performance continues with the next statement after the handler action completes.

Example 2: Teradata Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in Teradata transaction mode. Because the statements are invoked in Teradata transaction mode, each is an implicit transaction.

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
      WHERE Employee_Number = 1000;

CALL spSample4();
```

When the above three SQL statements are invoked in Teradata transaction mode, the following occurs:

Stage	Process
1	The stored procedure statement marked as Statement_4_2 raises an exception with the SQLSTATE code '42000'. The implicit statement is rolled back.
2	SQLSTATE code '42000' invokes the CONTINUE handler defined to handle that specific condition.
3	Since this handler type is CONTINUE, the changes made by Statement_4_1 is not affected. Because the first two BTEQ requests are implicit transactions, their updates are not rolled back. Control passes to Statement_4_3 after the handler action completes.

Example 3: Teradata Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in Teradata transaction mode. Note that the BT statement at the beginning of the sequence makes the SQL statements into a single explicit transaction.

```
BT;

INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
   WHERE Employee_Number = 1000;

CALL spSample4();
```

When the above three SQL statements are invoked in Teradata transaction mode, the following occurs:

Stage	Process
1	The stored procedure statement marked as Statement_4_2 raises an exception with the SQLSTATE indicating a failure condition.
2	The updates made by Statement_4_1, Statement_4_2, and the first three BTEQ requests are all rolled back.
3	The failure condition invokes the CONTINUE handler defined to handle that specific condition.
4	Because the handler type is CONTINUE, Statement_4_3 is performed after the handler action completes. Note that both the handler action and Statement_4_3 are performed as implicit transactions because the effect of the initial BT was revoked when it was rolled back in Stage 2.

DECLARE HANDLER (EXIT Type)

Introduction

EXIT handlers deal with conditions that are serious enough to terminate the procedure.

Actions Taken By An EXIT Handler

When a condition is raised, an EXIT handler does the following:

1. Performs the handler action.
2. Implicitly exits the BEGIN - END compound statement in which the condition is raised.

The stored procedure execution continues with the remaining statements outside the compound statement. If the procedure contains no other statement, the procedure terminates and control passes to the caller.

The following table describes the detailed flow of control for an EXIT handler when it is activated by a raised exception or completion condition:

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>A stored procedure statement raises an exception or completion condition.</td></tr>
<tr><td>2</td><td>The EXIT handler written to handle that exception performs its designated action.
<table>
<tr><th>IF ...</th><th>THEN the next stage in the process is this ...</th></tr>
<tr><td>the handler action completes successfully</td><td>control transfers to the end of the compound statement or, if at the top level, exits the stored procedure.
All open cursors declared in the compound statement are implicitly closed.</td></tr>
<tr><td>the CREATE PROCEDURE statement for this procedure defines INOUT or OUT parameters</td><td>the value for ACTIVITY_COUNT in the SUCCESS response is set to 1.</td></tr>
<tr><td>no INOUT or OUT parameters are defined for the procedure</td><td>then the value for ACTIVITY_COUNT in the SUCCESS response is set to 0.</td></tr>
<tr><td>the caller is a stored procedure</td><td>the status variable in the calling stored procedure is set to a value appropriate for the returned condition code.
For SQLSTATE, the value is set to ’00000’.
For SQLCODE, the value is set to 0.</td></tr>
<tr><td>a control statement raises an exception</td><td>control exits the compound statement that contains the invoked EXIT handler.</td></tr>
</table></td></tr>
</table>

Stage	Process
3	If the handler action raises a condition, it is handled in case an appropriate handler is defined within the handler action clause. See "Conditions Raised by a Handler Action" on page 9-69.

Examples of an EXIT Handler

The following examples illustrate the behavior of an EXIT handler. The examples are based on the following stored procedure:

```
CREATE PROCEDURE spSample5()
BEGIN
  DECLARE hNumber INTEGER;
  DECLARE EXIT HANDLER
  FOR SQLSTATE '42000'
   INSERT INTO Proc_Error_Table
    (:SQLSTATE, CURRENT_TIMESTAMP, 'spSample5',
     'Failed to Insert Row');

-- Statement_5_1
UPDATE Employee
  SET Salary_Amount = 10000
  WHERE Employee_Number = 1001;

-- Statement_5_2
INSERT INTO EmpNames VALUES (1002, 'Thomas');

-- If the EmpNames table had been dropped, Statement_5_2
-- returns an SQLSTATE code of '42000' that is handled.

-- Statement_5_3
UPDATE Employee
  SET Salary_Amount = 10000
  WHERE Employee_Number = 1003;
END;
```

In the cases that follow, control exits the stored procedure and passes to the caller after the handler action is complete because the example stored procedure contains only one compound statement.

In the case of a stored procedure with nested compound statements, the scope of the EXIT handler and its behavior described in these examples apply only to the compound statement in which the handler is defined.

If the handler defined within the compound statement cannot handle the raised condition, then the condition is propagated outwards in search of a suitable handler.

Example 1: ANSI Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI transaction mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
   WHERE Employee_Number = 1000;

CALL spSample5();
```

If an exception condition that is not a failure condition is reported, the following occurs:

Stage	Process
1	The stored procedure statement marked as Statement_5_2 raises an exception with the SQLSTATE code '42000'. The request is rolled back.
2	SQLSTATE code '42000' invokes the EXIT handler defined to handle that specific condition.
3	Because this handler type is EXIT, the update made by Statement_5_1 is not affected. Control passes to the caller after the handler action completes. If the stored procedure has any other statements outside the calling compound statement, control passes to the next statement outside the calling compound statement. The implicit transaction initiated by the first interactively invoked SQL statement remains outstanding.

If an exception condition is reported (that is a failure condition), the following occurs:

Stage	Process
1	The stored procedure statement marked as Statement_5_2 raises an exception with the SQLSTATE code that indicates a failure condition.
2	The effects of Statement_5_1, Statement_5_2, and the first two interactively entered SQL statements are rolled back.
3	The returned SQLSTATE code invokes the EXIT handler defined for that specific condition.
4	Control exists the calling compound statement and passes to the next statement, if any, after the handler action completes.
5	A new transaction remains outstanding if there are SQL statements performed in the EXIT handler that have not been committed.

Example 2: Teradata Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in Teradata transaction mode. Because the statements are invoked in Teradata transaction mode, each is an implicit transaction.

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
   WHERE Employee_Number = 1000;

CALL spSample5();
```

When the above three SQL statements are invoked in Teradata transaction mode, the following occurs:

Stage	Process
1	The stored procedure statement marked as Statement_5_2 raises an exception with the SQLSTATE code '42000'. The implicit statement is rolled back.
2	SQLSTATE code '42000' invokes the EXIT handler defined for that specific condition.
3	Because this handler type is EXIT, and Statement_5_1 was performed in an implicit transaction, the update made by that statement is not affected. Because the first two BTEQ requests are implicit transactions, their updates are not rolled back. Control exits the calling compound statement and passes to the next statement, if any, after the handler action completes.

Example 3: Teradata Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in Teradata transaction mode. Note that the BT statement at the beginning of the sequence makes the SQL statements into a single explicit transaction.

```
BT;
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
   WHERE Employee_Number = 1000;

CALL spSample5();
```

When the above three SQL statements are invoked in Teradata transaction mode, the following occurs:

Stage	Process
1	The stored procedure statement marked as Statement_5_2 raises an exception with the SQLSTATE code '42000', indicating a failure condition.
2	The updates made by Statement_5_1, Statement_5_2, and the first three BTEQ requests are all rolled back.
3	The failure condition invokes the EXIT handler defined to handle that specific condition.
4	Because the handler type is EXIT, control exits the compound statement and passes to the next statement, if any, after the handler action completes. Note that the handler action is performed as an implicit transaction because the effect of the initial BT was revoked when it was rolled back in Stage 2.

Example of an EXIT Handler That Contains a COMMIT Statement

The following example illustrates the behavior of an EXIT handler that contains a COMMIT transaction control statement. The example assumes the following stored procedure:

```
CREATE PROCEDURE spSample6()
BEGIN
  DECLARE hNumber INTEGER;
  DECLARE EXIT HANDLER
  FOR SQLSTATE '42000'
   INSERT INTO Proc_Error_Table
    (:SQLSTATE, CURRENT_TIMESTAMP, 'spSample6',
     'Failed to Insert Row');

-- Statement_6_1
UPDATE Employee
  SET Salary_Amount = 10000
  WHERE Employee_Number = 1001;

-- Statement_6_2
COMMIT;

-- Statement_6_3
UPDATE Employee
  SET Salary_Amount = 10000
  WHERE Employee_Number = 1003;
```

```
-- Statement_6_4
INSERT INTO EmpNames VALUES (1002, 'Thomas');

-- If the EmpNames table had been dropped, Statement_6_2
-- returns an SQLSTATE code of '42000' that is handled.
END;
```

Example 1: ANSI Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI transaction mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
   WHERE Employee_Number = 1000;

CALL spSample6();
```

If an exception condition that is not a failure condition is reported, then the following occurs:

Stage	Process
1	The first two BTEQ requests and Statement_6_1 and Statement_6_2 from the stored procedure perform and commit normally.
2	Statement_6_3 initiates a new transaction.
3	The stored procedure statement marked as Statement_6_4 raises an exception with the SQLSTATE code '42000'.
4	Statement_6_4 is rolled back.
5	SQLSTATE code '42000' invokes the EXIT handler defined to handle that specific condition.
6	Because this handler type is EXIT, control exits the compound statement and passes to the next statement outside that compound statement, if any, after the handler action completes. If the stored procedure contains no other statement, the procedure terminates and control passes to the caller. The handler action performs within the transaction begun by Statement_6_3.

If an exception is report (that is a failure condition), the following occurs:

Stage	Process
1	The stored procedure statement marked as Statement_6_4 raises an exception with the SQLSTATE code that indicates a failure condition.
2	The effects of Statement_6_1, Statement_6_2, and the first two interactively entered SQL statements are committed and are not rolled back. The failure of Statement_6_4 rolls back its transaction *and* that of Statement_6_3 (because Statement_6_3 was not committed).
3	The failure condition invokes the EXIT handler defined to handle that specific condition.
4	The handler action statements initiate a new transaction. Control exits the calling compound statement and passes to the next statement, if any, after the handler action completes. If the stored procedure contains no other statement, the procedure terminates and control passes to the caller.

DECLARE HANDLER (SQLEXCEPTION Form)

Introduction

SQLEXCEPTION is a generic condition that represents the SQLSTATE codes for all exception conditions. The handler associated with SQLEXCEPTION is invoked when an exception condition is raised during statement execution, and a handler to handle the specific exception condition does not exist.

An SQLEXCEPTION handler can be written as an EXIT handler or as a CONTINUE handler.

Actions Taken By An SQLEXCEPTION Handler

The following table describes the flow of control for an SQLEXCEPTION handler when it is activated by a raised exception:

Stage	Process
1	A statement in the stored procedure raises an exception.
2	The generic condition handler is invoked if no handler exists to handle the specific exception condition.
3	An SQLEXCEPTION handler performs its designated action.
4	The next stage in the process depends on the handler type. **IF the handler is this type …** / **THEN control passes to the …** CONTINUE / next statement in the current block. EXIT / end of the current block.
5	Interaction with specific handlers varies depending on the situation. See "Example 3: Specific and Generic Condition Handlers in the Same Procedure" on page 9-94 for an example.

Example 1: Generic Condition Handler

The following example illustrates the behavior of an SQLEXCEPTION handler. The example assumes the following stored procedure:

```
CREATE PROCEDURE spSample11()
BEGIN
  DECLARE hNumber INTEGER;
  DECLARE EXIT HANDLER
  FOR SQLEXCEPTION
    INSERT INTO Proc_Error_Table (:SQLSTATE,
      CURRENT_TIMESTAMP, 'spSample11', 'Generic
                              handler performed');

  -- Statement_11_1
  UPDATE Employee
    SET Salary_Amount = 10000
     WHERE Employee_Number = 1001;

  -- Statement_11_2
  INSERT INTO EmpNames VALUES (1002, 'Thomas');

  -- If the EmpNames table had been dropped,
  -- Statement_11_2 returns SQLSTATE '42000' that is handled.

  -- Statement_11_3
   UPDATE Employee
    SET Salary_Amount = 10000
     WHERE Employee_Number = 1003;
END;
```

Example 2: ANSI Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI transaction mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
   WHERE Employee_Number = 1000;

CALL spSample11();
```

When the above three SQL statements are invoked in ANSI transaction mode, the following occurs:

Stage	Process
1	Statement_11_2 in the called stored procedure raises an error condition with SQLSTATE '42000' that is handled by the SQLEXCEPTION handler because no specific handler exists for SQLSTATE code '42000'.
2	Statement_11_2 is rolled back.
3	Because the condition handler is an EXIT handler, control passes to the caller after the handler action finishes. If the stored procedure has nested blocks, control passes to the next statement following the calling compound statement.
4	The following items remain active and uncommitted: • The first two interactive SQL statements • Statement_11_1 • The INSERT statement inside the handler

Example 3: Specific and Generic Condition Handlers in the Same Procedure

The following example illustrates the behavior of an SQLEXCEPTION handler and a specific condition handler when both DECLARE HANDLER forms are combined in a stored procedure. The example assumes the following stored procedure:

```
CREATE PROCEDURE spSample12()
BEGIN
  DECLARE hNumber INTEGER;
  DECLARE CONTINUE HANDLER
    FOR SQLEXCEPTION
  -- Handler_1
    BEGIN
      UPDATE exception_table
        SET exception_count = exception_count + 1;
      INSERT INTO Proc_Error_Table (:SQLSTATE,
        CURRENT_TIMESTAMP, 'spSample12', 'Failed to insert
                                          row');
    END;
  DECLARE EXIT HANDLER FOR SQLSTATE '53000'
  -- Handler_2
    INSERT INTO Proc_Error_Table (:SQLSTATE,
    CURRENT_TIMESTAMP, 'spSample12', 'Column does not exist');

  -- Statement_12_1
  UPDATE Employee
    SET Salary_Amount = 10000
    WHERE Employee_Number = 1001;
```

```
  -- Statement_12_2
  INSERT INTO EmpNames VALUES (1002, 'Thomas');

  -- If the EmpNames table has been dropped, Statement_12_2
 -- returns the SQLSTATE code '42000' that is handled

  -- Statement_12_3

  UPDATE Employee
    SET Salary_Amount = 10000
    WHERE Employee_number = 1003;

 -- If Salary_Amount has been dropped,  -- Statement_12_3
returns the SQLSTATE code '53000' that is handled

END;
```

Example 4: ANSI Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI transaction mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
  WHERE Employee_Number = 1000;

CALL spSample12();
```

When the above three SQL statements are invoked in ANSI transaction mode, the following occurs:

Stage	Process
1	Statement_12_2 in the stored procedure raises an exception with SQLSTATE code '42000' that is not handled because no condition handler exists.
2	Statement_12_2 is rolled back.
3	The generic SQLEXCEPTION handler, named Handler_1 by a comment, is activated. On successful completion of the handler action Statement_12_3 performs because Handler_1 is a CONTINUE handler.
4	Statement_12_3 raises an exception condition with SQLSTATE code '53000'.
5	Control passes to Handler_2, which is explicitly defined to handle that SQLSTATE condition.

Stage	Process
6	Handler_2 performs its handler action. Because Handler_2 is an EXIT handler, control passes to the end of the block after the handler action completes. The procedure terminates, if it does not contain any other statements.
7	The following items remain active, but are not committed: • The first two interactive SQL statements • Statement_12_1 • Action statement for Handler_1 • Action statement for Handler_2

DECLARE HANDLER (SQLWARNING Form)

Introduction

SQLWARNING is a generic condition that represents the SQLSTATE codes for all completion conditions (other than successful completion and "no data found" conditions).

The handler associated with SQLWARNING is invoked when a completion condition is raised during statement execution, and a handler to handle the specific completion condition does not exist.

An SQLWARNING handler can be of EXIT type or CONTINUE type.

Actions Taken By An SQLWARNING Handler

The flow of control for an SQLWARNING generic condition handler is similar to the flow of control for the SQLEXCEPTION handler. The difference is that an SQLWARNING handler is activated by a raised completion condition.

SQLWARNING cannot handle a "no data found" condition. An SQLWARNING handler, if declared in a stored procedure either along with a NOT FOUND handler or separately, is not activated by a "no data found" completion condition.

Example 1: Generic Condition Handler

The following example illustrates the behavior of an SQLWARNING handler. The example assumes the following stored procedure:

```
CREATE PROCEDURE spSample11()
BEGIN
  DECLARE EmpCount INTEGER;
  DECLARE EXIT HANDLER
  FOR SQLWARNING
    INSERT INTO Proc_Error_Table (:SQLSTATE,
      CURRENT_TIMESTAMP, 'spSample11', 'Generic handler
           performed');

  -- Statement_11_1
  UPDATE Employee
    SET Salary_Amount = 10000
     WHERE Employee_Number = 1001;
```

```
  -- Statement_11_2
  SELECT COUNT(*) INTO :EmpCount FROM Employee SAMPLE 5;

  -- Suppose table Employee has only three rows.
  -- Statement_11_2  returns SQLSTATE 'T7473' that is
  -- handled by the SQLWARNING handler.
END;
```

Example 2: ANSI Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI transaction mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
   WHERE Employee_Number = 1000;

CALL spSample11();
```

When the above three SQL statements are invoked in ANSI transaction mode, the following occurs:

Stage	Process
1	Statement_11_2 in the called stored procedure raises a completion condition with SQLSTATE 'T7473' that is handled by the SQLWARNING handler because no specific handler exists for SQLSTATE code 'T7473' .
2	Because the condition handler is of EXIT type, control passes to the caller after the handler action finishes.
3	The following items remain active and uncommitted: • The first two interactive SQL statements • Statement_11_1 • The INSERT statement inside the handler

Example 3: Specific and Generic Condition Handlers in the Same Procedure

The following example illustrates the behavior of an SQLWARNING handler and a specific condition handler when both DECLARE HANDLER forms are combined in a stored procedure. The example assumes the following stored procedure:

```
CREATE PROCEDURE spSample12()
BEGIN
  DECLARE EmpCount INTEGER DEFAULT 0;
  -- Handler_1
 DECLARE CONTINUE HANDLER
    FOR SQLWARNING
    BEGIN
      UPDATE warning_table
        SET warning_count = warning_count + 1;
      INSERT INTO Proc_Error_Table (:SQLSTATE,
        CURRENT_TIMESTAMP, 'spSample12', 'Generic warning
             handler');
    END;

  -- Handler_2
 DECLARE EXIT HANDLER FOR SQLSTATE 'T7473'
    INSERT INTO Proc_Error_Table (:SQLSTATE,
    CURRENT_TIMESTAMP, 'spSample12', 'Requested sample is
larger              than table rows');

  -- Statement_12_1
  UPDATE Employee
    SET Salary_Amount = 10000
    WHERE Employee_Number = 1001;

  -- Statement_12_2
  SELECT COUNT(*) INTO :EmpCount FROM Employee SAMPLE 5;

  -- Suppose the table Employee has only three rows.
  -- Statement_12_2 returns SQLSTATE 'T7473' that is
  -- handled by specific handler.
END;
```

Example 4: ANSI Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI transaction mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
   WHERE Employee_Number = 1000;

CALL spSample12();
```

When the above three SQL statements are invoked in ANSI transaction mode, the following occurs:

Stage	Process
1	Statement_12_2 in the called stored procedure raises a completion condition with SQLSTATE code 'T7473' that is handled by the specific handler Handler_2.
2	Handler_2 performs its handler action. Because Handler_2 is an EXIT handler, and the procedure has only one compound statement, the procedure terminates after the handler action completes.
3	The following items remain active, but are not committed: • The first two interactive SQL statements • Statement_12_1 • Action statement for Handler_2

DECLARE HANDLER (NOT FOUND Form)

Introduction

NOT FOUND is a generic condition that represents the SQLSTATE codes for all "no data found" completion conditions.

The handler associated with NOT FOUND is invoked when a "no data found" completion condition is raised during statement execution and a handler to handle the specific condition does not exist.

A NOT FOUND handler can be of EXIT type or CONTINUE type.

Actions Taken By a NOT FOUND Handler

The flow of control for a NOT FOUND generic condition handler is similar to the flow of control for an SQLEXCEPTION or SQLWARNING handler. The difference is that a NOT FOUND handler is activated when a "no data found" completion condition is raised.

Example 1: Generic Condition Handler

The following example illustrates the behavior of a NOT FOUND handler. The example assumes the following stored procedure:

```
CREATE PROCEDURE spSample11()
BEGIN
  DECLARE EmpCount INTEGER;
  DECLARE EXIT HANDLER
  FOR NOT FOUND
    INSERT INTO Proc_Error_Table (:SQLSTATE,
      CURRENT_TIMESTAMP, 'spSample11', 'Generic
      no data found handler performed');

  -- Statement_11_1
  UPDATE Employee
    SET Salary_Amount = 10000
    WHERE Employee_Number = 1001;
```

```
  -- Statement_11_2
  DELETE Employee WHERE Employee_Number = 1;

  -- Suppose table Employee does not have a row for
  -- Employee_Number 1. Statement_11_2 returns SQLSTATE
  -- '02000' that is handled by NOT FOUND handler.

END;
```

Example 2: ANSI Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI transaction mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
   WHERE Employee_Number = 1000;

CALL spSample11();
```

When the above three SQL statements are invoked in ANSI transaction mode, the following occurs:

Stage	Process
1	Statement_11_2 in the called stored procedure raises a completion condition with SQLSTATE '02000' that is handled by the NOT FOUND handler because no specific handler exists for SQLSTATE code '02000' .
2	Because the condition handler is an EXIT handler, control passes to the caller after the handler action finishes.
3	The following items remain active and uncommitted: • The first two interactive SQL statements • Statement_11_1 • The INSERT statement inside the handler

Example 3: Specific and Generic Condition Handlers in the Same Procedure

The following example illustrates the behavior of a NOT FOUND handler and a specific condition handler when both DECLARE HANDLER forms are combined in a stored procedure. The example assumes the following stored procedure:

```
CREATE PROCEDURE spSample12()
BEGIN
  DECLARE CONTINUE HANDLER
  FOR NOT FOUND
  -- Handler_1
    BEGIN
      UPDATE warning_table
        SET warning_count = warning_count + 1;
      INSERT INTO Proc_Error_Table (:SQLSTATE,
        CURRENT_TIMESTAMP, 'spSample12', 'Generic no data
found             handler');
    END;
  DECLARE EXIT HANDLER FOR SQLSTATE '02000'
  -- Handler_2
    INSERT INTO Proc_Error_Table (:SQLSTATE,
    CURRENT_TIMESTAMP, 'spSample12', 'No data found');

  -- Statement_12_1
  UPDATE Employee
    SET Salary_Amount = 10000
    WHERE Employee_Number = 1001;

  -- Statement_12_2
  DELETE Employee WHERE Employee_Number = 1;

  -- Suppose table Employee does not have a row for
  -- Employee_Number 1. Statement_12_2 returns SQLSTATE
  -- '02000' that is handled by NOT FOUND handler.

END;
```

Example 4: ANSI Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI transaction mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
WHERE Employee_Number = 1000;

CALL spSample12();
```

When the above three SQL statements are invoked in ANSI transaction mode, the following occurs:

Stage	Process
1	Statement_12_2 in the called stored procedure raises a completion condition with SQLSTATE code '02000' that is handled by the specific handler Handler_2.
2	Handler_2 performs its handler action. Because Handler_2 is an EXIT handler, control passes to the caller after the handler action completes.
3	The following items remain active, but are not committed: • The first two interactive SQL statements • Statement_12_1 • Action statement for Handler_2

DECLARE HANDLER (Control Statement Handling)

Introduction

This section describes the behavior of SQL statements when they raise conditions in a stored procedure.

Cursor Handling for Exceptions in FOR Loops

The respective handling of open cursors for Failure and Error conditions are described in the following table:

IF this exception occurs while a FOR cursor loop is executing ...	THEN all open cursors are ...
FAILURE	closed as part of a transaction rollback.
ERROR	not closed.

The handler action specified by the condition handler is performed only after all the open cursors have been closed.

Example 1: WHILE Loop Exceptions

The example below assumes the following stored procedure:

```
CREATE PROCEDURE spSample8()
BEGIN
  DECLARE hNumber INTEGER;
  DECLARE CONTINUE HANDLER
  FOR SQLSTATE '42000'
   INSERT INTO Proc_Error_Table
    (:SQLSTATE, CURRENT_TIMESTAMP, 'spSample8',
     'Failed to Insert Row');

   SET hNumber = 1;

   -- Statement_8_1
   UPDATE Employee SET Salary_Amount = 10000
   WHERE Employee_Number = 1001;

   WHILE hNumber < 10
   DO
   -- Statement_8_2
   INSERT INTO EmpNames VALUES (1002, 'Thomas');
   SET hNumber = hNumber + 1;
   END WHILE;
```

```
   -- If the EmpNames table had been dropped,
   -- Statement_8_2 returns an SQLSTATE code of
   -- '42000' that is handled.

   -- Statement_8_3
   UPDATE Employee
     SET Salary_Amount = 10000
     WHERE Employee_Number = 1003;
END;
```

Example 2: ANSI Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI transaction mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee
SET Salary_Amount = 10000
WHERE Employee_Number = 1000;

CALL spSample8();
```

When the above three SQL statements are invoked in ANSI transaction mode, the following occurs:

Stage	Process
1	Statement_8_2 within the called stored procedure raises an error condition with SQLSTATE '42000' that is handled. The statement is rolled back.
2	The condition handler is activated.
3	Because the handler type is CONTINUE, performance resumes from the SET statement within the WHILE loop after the handler action completes and the WHILE loop is *not* exited because of the exception.
4	During each iteration Statement_8_2 raises an exception which is handled. Statement_8_3 performs on termination of the WHILE loop.
5	The following items are not committed: • The first two interactive SQL statements • Statement_8_1 • Action statement for the condition handler • Statement_8_3

When stored procedure spSample8 is created in a session in Teradata transaction mode, the process described in the above table applies with one difference: because every request is an implicit transaction in Teradata transaction mode, the following statements are committed:

- The first two interactive SQL statements
- Statement_8_1
- Action statement for the condition handler
- Statement_8_3

Example 3: Exceptions Raised By an IF Statement

The example below assumes the following stored procedure:

```
CREATE PROCEDURE spSample9()
BEGIN
  DECLARE hNumber, NumberAffected INTEGER;
  DECLARE CONTINUE HANDLER
   FOR SQLSTATE '22012'
   INSERT INTO Proc_Error_Table
    (:SQLSTATE, CURRENT_TIMESTAMP, 'spSample9',
     'Failed Data Handling');

   SET hNumber = 100;

     -- Statement_9_1
     UPDATE Employee SET Salary_Amount = 10000
       WHERE Employee_Number BETWEEN 1001 AND 1010;

   SET NumberAffected = ACTIVITY_COUNT;

   IF hNumber/NumberAffected < 10 THEN

   -- If the UPDATE in Statement_9_1 results in 0 rows
   -- being affected, the IF condition raises an
   -- exception with SQLSTATE '22012' that is
   -- handled.

   -- Statement_9_2
     INSERT INTO data_table (NumberAffected, 'DATE');

     SET hNumber = hNumber + 1;

   END IF;
```

```
   -- Statement_9_3
     UPDATE Employee
       SET Salary_Amount = 10000
       WHERE Employee_Number = 1003;
END;
```

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI transaction mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee
SET Salary_Amount = 10000
WHERE Employee_Number = 1000;

CALL spSample9();
```

With respect to the above stored procedure, consider the following:

Stage	Process
1	The IF statement in the called stored procedure raises a divide-by-zero error condition with SQLSTATE '22012' that is handled.
2	Because the handler type is CONTINUE, performance resumes from Statement_9_3 after the handler action completes.
3	Statement_9_2 and the SET statement are inside the IF statement that raised the error condition; so they are not performed. The updates made by the following items remain intact in a transaction that is uncommitted: • Statement_9_1 • Statement_9_3

Example 4: Exception Raised by a Condition in WHILE Loop

The following example illustrates the behavior of WHILE statement when a condition in the loop raises an exception. This behavior also applies to IF and FOR statements. The example assumes the following stored procedure.

```
CREATE PROCEDURE spSample8()
BEGIN
   DECLARE hNumber INTEGER;
   DECLARE CONTINUE HANDLER
   FOR SQLSTATE '22012'
    INSERT INTO Proc_Error_Table
    (:SQLSTATE,    CURRENT_TIMESTAMP, 'spSample8',
    'Failed in WHILE condition');

   SET hNumber = 1;
   SET hNo = 0;
```

```
   -- Statement_8_1
   UPDATE Employee SET Salary_Amount = 10000
     WHERE Employee_Number = 1001;

     WHILE ((hNumber/hNo) < 10)
   DO
   -- Statement_8_2
   INSERT INTO EmpNames VALUES (1002, 'Thomas');
   SET hNumber = hNumber + 1;
   END WHILE;

   -- The condition in WHILE statement raises
   -- an exception and returns SQLSTATE code
   -- of 22012 that is handled.

   -- Statement_8_3
   UPDATE Employee
    SET Salary_Amount = 10000
    WHERE Employee_Number = 1003;
END;
```

Example 5: ANSI Transaction Mode

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI transaction mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee
SET Salary_Amount = 10000
WHERE Employee_Number = 1000;

CALL spSample8();
```

When the above three SQL statements are invoked in ANSI transaction mode, the following occurs:

Stage	Process
1	The condition in the WHILE statement within the called stored procedure raises an exception.
2	The condition handler is activated. Since the condition handler is of CONTINUE type control passes to the next statement after the WHILE loop; that is, statement_8_3 and execution of stored procedure continues from statement_8_3.
3	Statement_8_2 and SET statement in WHILE loop are not performed.

Stage	Process
4	On completion of the stored procedure execution, the following statements are not committed: • The first two interactive SQL statements • Statement_8_1 • Action statement for the condition handler • Statement_8_3

When stored procedure spSample8 is created in a session in Teradata transaction mode, the process described in the above table applies with one difference: because every request is an implicit transaction in Teradata transaction mode, the following statements are committed:

- The first two interactive SQL statements
- Statement_8_1
- Action statement for the condition handler
- Statement_8_3

FOR

Purpose

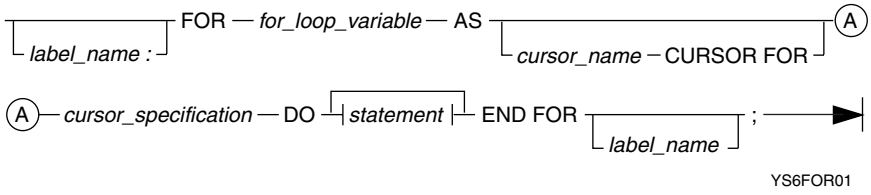
Performs a statement for each row fetched from a table.

Invocation

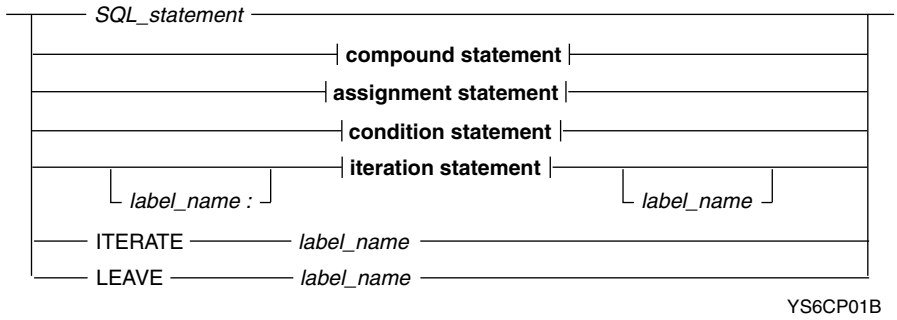
Executable.

Stored procedures only.

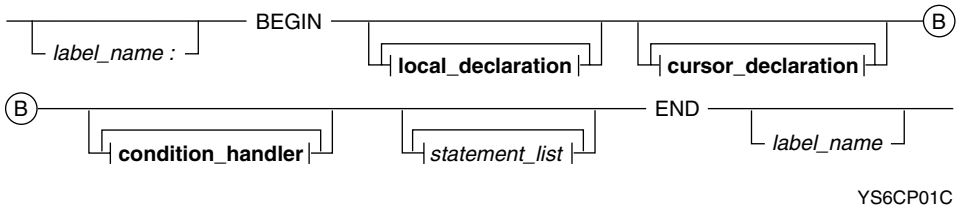
Syntax



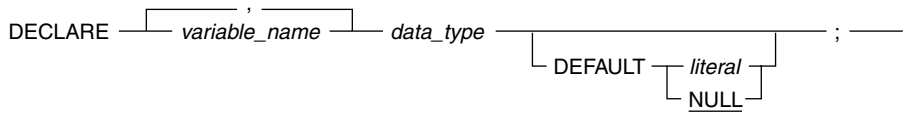
statement



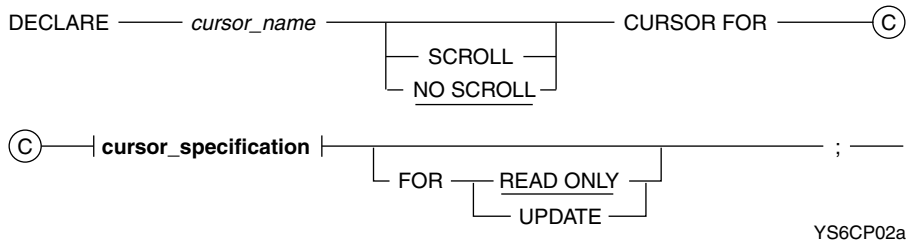
compound statement



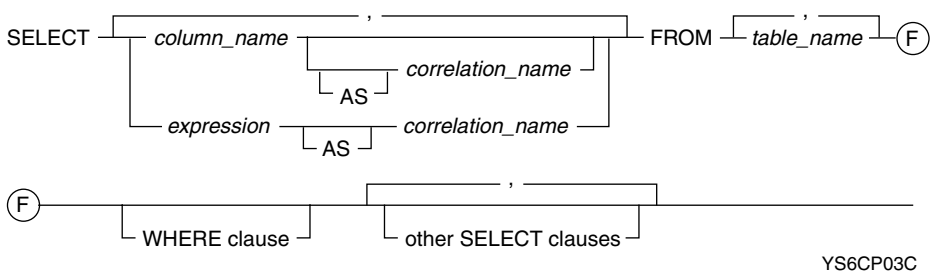
local_declaration



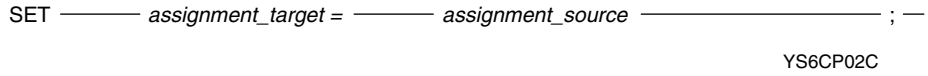
cursor_declaration



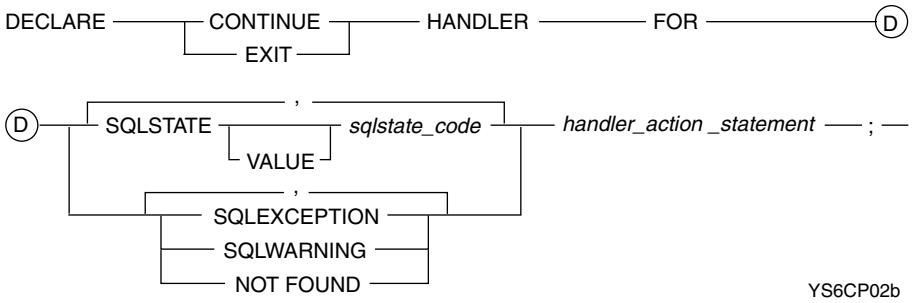
cursor-specification



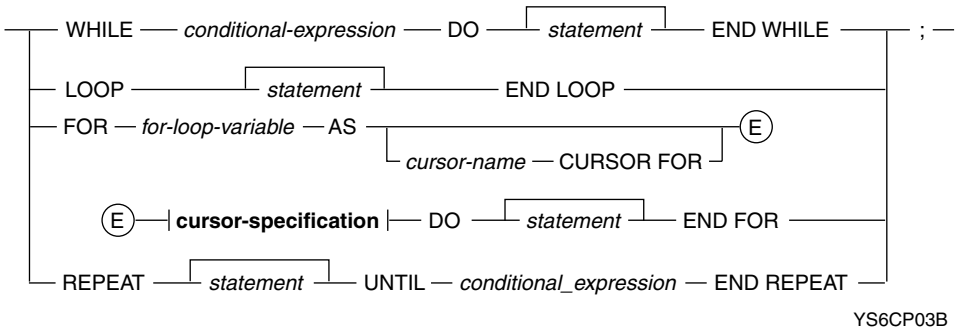
assignment statement



condition-handler



iteration statement



YS6CP03B

where:

Syntax element …	Specifies …
label_name	an optional label for the FOR statement. If an ending-label is specified, you must specify a beginning-label that is equivalent to the ending-label. The beginning-label must be terminated by a COLON character (:). The label name of the BEGIN - END compound statement cannot be reused in an iteration statement. One label name cannot be reused within one group of nested FOR statements, but can be reused for different non-nesting iteration statements.
for_loop_variable	the name of the loop.
cursor_name	the name of the cursor. Used for updatable cursors as the object of the WHERE CURRENT OF clause.
cursor_specification	a single SELECT statement used as the cursor. In read-only cursors, the single SELECT statement can contain one SELECT or multiple SELECTs that use set operators like UNION, INTERSECT or MINUS. Updatable cursors do not support the set operators.
statement	one or more DML, DDL, DCL statements, including dynamic SQL statements, or control statements, including BEGIN - END compound statements.

ANSI Compliance

FOR is ANSI SQL-99-compliant.

Authorization

FOR requires the following privileges:

- SELECT privilege on the database object referenced in the specified *cursor_specification*.
- UPDATE or DELETE privilege if the cursor is updatable.

LEAVE and ITERATE Within a FOR Statement

You can perform LEAVE and ITERATE statements within a FOR block.

See "ITERATE" on page 9-125 and "LEAVE" on page 9-129 for details.

Using a Correlation Name for a Cursor Specification

You can define aliases for the columns and expressions in a cursor using the standard *object* AS *correlation_name* syntax. You must qualify any aliased object with the *for_loop_variable* name if you reference it within the loop.

You cannot reference a non-aliased cursor expression within the loop.

SQL Statements Within a FOR Loop

The following rules apply to using SQL statements within a FOR loop:

- You can specify all DML statements, including CALL, positioned UPDATE and positioned DELETE.
- You can specify all control statements.
- Transaction statements are allowed only in Read-only cursors. They cannot be specified in updatable cursors.
- Each local variable, parameter, column, correlation name, or status variable referenced in the SQL statement must have been previously declared.

Updatable and Read-Only Cursors

An updatable, or positioned, cursor is a cursor defined by the application for a query that can also used to update the results rows.

A cursor is updatable if there is at least one positioned DELETE or positioned UPDATE that references it inside the FOR loop.

You can use updatable and read-only cursors in stored procedures with the following exceptions:

Updatable Cursors	Read-Only Cursors
Allowed only in ANSI transaction mode.	Allowed in ANSI and Teradata transaction modes.
Positioned DELETE or UPDATE statements can be used. The table name in these statements must be the same as that used in the cursor specification. • A positioned UPDATE can perform multiple updates of the current row of the cursor. • A positioned DELETE can delete the current row of the cursor after multiple updates.	Positioned DELETE or UPDATE statements cannot be used.

Rules For Cursors

The following rules apply to the use of FOR cursors:

- ABORT, COMMIT, and ROLLBACK statements are not permitted in an updatable cursor.

  An attempt to perform any of these statements returns a runtime error.
- The cursor specification must not return the warning code 3999.
- The cursor specification cannot contain a WITH…BY clause.
- If the cursor specification contains a UNION operator, the referenced correlation or column name must be the correlation or column names used in the first SQL SELECT statement.

Rules For FOR Loop Variables

The following rules apply to the use of FOR loop variables:

- FOR loop variable names must be unique if they are used in nested FOR iteration loops.
- FOR loop variable names can be the same as the cursor name and correlation names within a FOR iteration statement.
- If you use a FOR loop variable in an SQL statement other than a control statement within the iteration statement, you must prefix it with a COLON character (:).
- Unqualified symbols in a FOR loop are assumed to be variable or parameter names.

Rules for FOR Loop Correlation Names

The following rules apply to the use of FOR loop correlation names:

- A correlation name must be unique in a FOR iteration statement; however, the same correlation name can be used both for nested and non-nested FOR iteration statements.
- A correlation name can be the same as the FOR loop variable and the names of cursors within a FOR iteration statement.
- Columns and correlation names must be qualified with a FOR loop variable when referenced in SQL statements, including control statement, within the iteration statement.
- If a column or correlation name is not qualified, then column and correlation name references are treated as either parameters or local variables.
- The scope of a FOR iteration statement correlation name is the body of the statement.

Rules for FOR Loop Cursor Names

The following rules apply to the names of FOR loop cursors:

- A cursor name must be unique if used in the nested FOR iteration statements.
- A cursor name can be the same as the for-loop variable or the correlation or column names in a FOR statement.
- The scope of a cursor name is confined to the FOR statement in which it is defined. If FOR statements are nested, the cursor name associated with an outer FOR statement can be referenced in statements within inner FOR statement(s).

Example 1

```
L1:
   FOR CustCursor AS c_customer CURSOR FOR
     SELECT CustomerNumber AS Number
           ,CustomerName AS Name
           ,(Amount + 10000) a
       FROM customer
   DO
     SET hCustNbr = CustCursor.Number;
     SET hCustName = CustCursor.Name;
     SET hAmount = CustCursor.a + CustCursor.a * 0.20;
     INSERT INTO Cust_temp VALUES (hCustNbr, hCustName);
   END FOR L1;
```

Example 2

```
FOR CustCursor AS c_customer CURSOR FOR
     SELECT CustomerNumber
           ,CustomerName
       FROM Customer
   DO
     SET hCustNbr = CustCursor.CustomerNumber;
     SET hCustName = CustCursor.CustomerName;
     DELETE FROM Customer WHERE CURRENT OF c_customer;
   END FOR;
```

Example 3

```
L1:
   FOR CustCursor AS c_customer CURSOR FOR
      SELECT CustomerNumber AS Number
            ,CustomerName AS Name
            ,(Amount + 10000) a
        FROM Customer
   DO
      SET hCustNbr = CustCursor.Number;
      SET hCustName = CustCursor.Name;
      SET hAmount = CustCursor.a + CustCursor.a * 0.20;
      IF hAmount > 50000 THEN
        hAmount = 500000;
      END IF;
      UPDATE customer
        SET amount = :hAmount WHERE CURRENT OF c_customer;
      INSERT INTO Cust_temp VALUES (hCustNbr,
                   hCustName);
   END FOR;
```

IF

Purpose

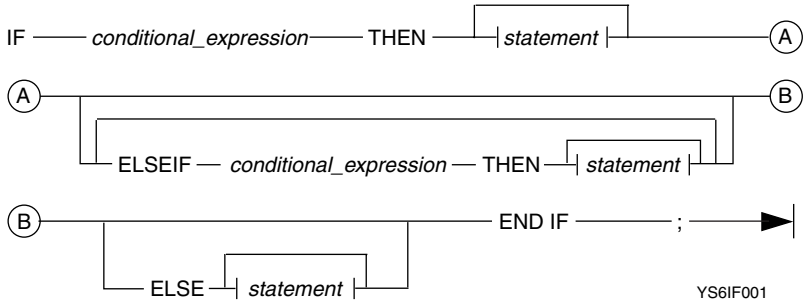
Provides conditional execution based on the truth value of a condition.

Invocation

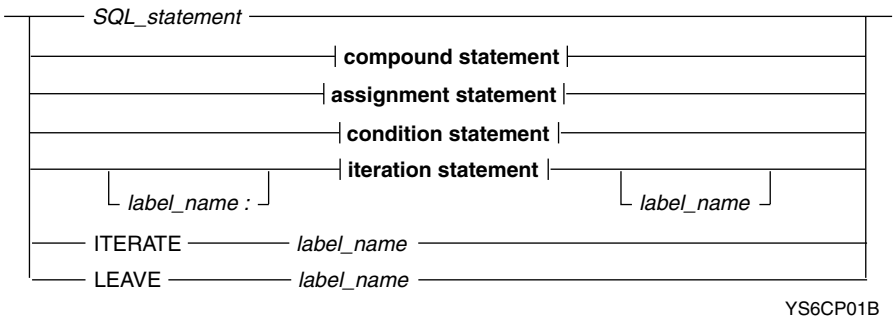
Executable

Stored procedures only.

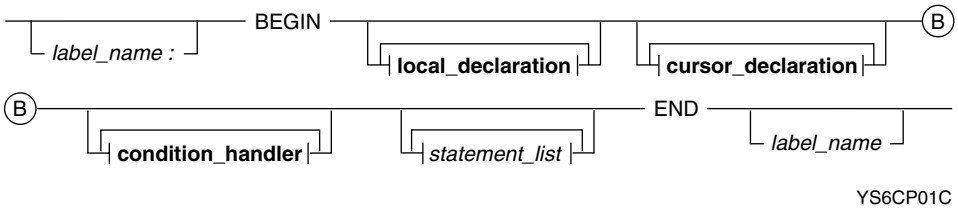
Syntax



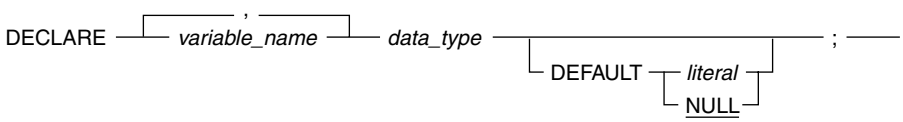
statement



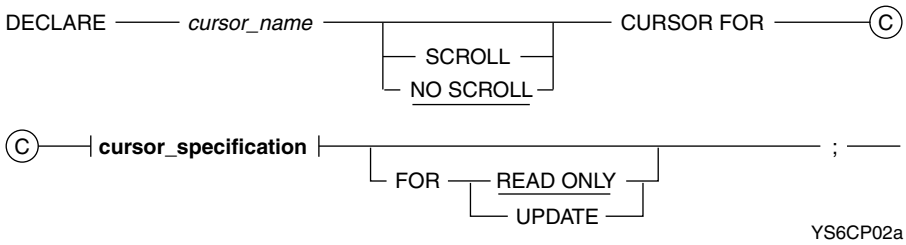
compound statement



local_declaration

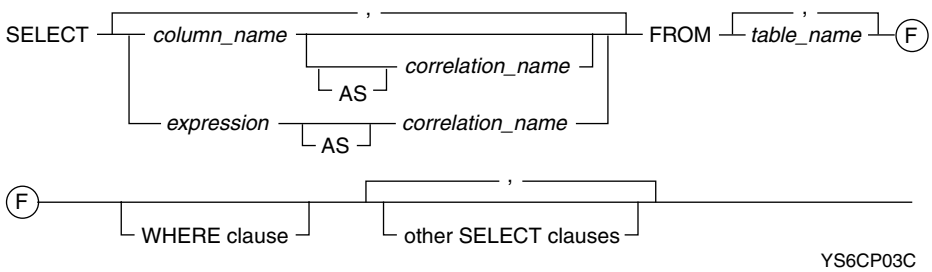


cursor_declaration



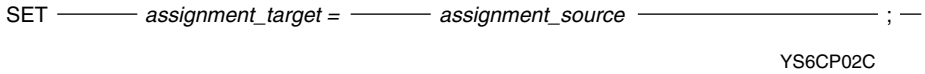
YS6CP02a

cursor-specification



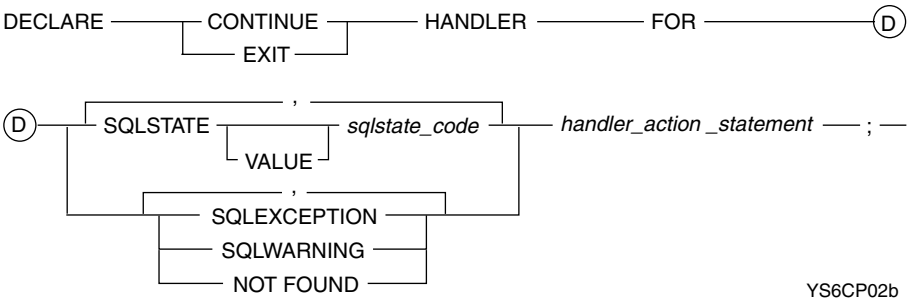
YS6CP03C

assignment statement



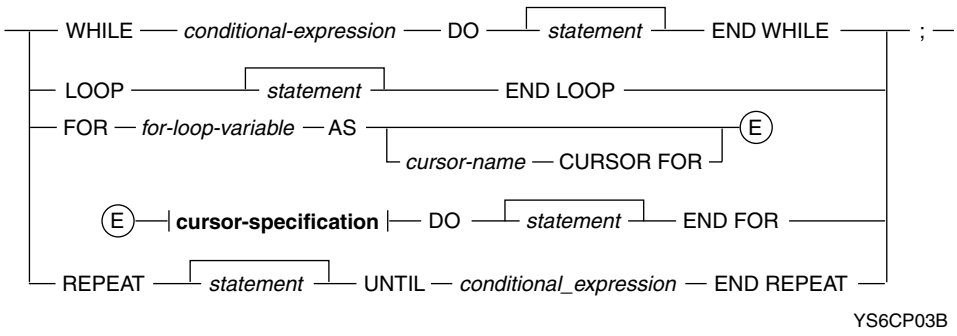
YS6CP02C

condition-handler



YS6CP02b

iteration statement



YS6CP03B

where:

Syntax element …	Specifies …
conditional_expression	a boolean condition used to evaluate whether a statement or statements embedded within the IF block should be performed. You cannot use IN and NOT IN operators if the conditional list contains any local variables, parameters, or cursor aliases. OUT parameters are not allowed in conditional_expression.
statement	any of the following: • DML, DDL or DCL statement that can be used in a stored procedure. These include dynamic SQL statements. • Control statements, including BEGIN - END compound statements.

ANSI Compliance

IF is ANSI SQL-99-compliant.

Authorization

None.

IF Syntax Combinations

There are four valid forms of the IF statement:

- IF-THEN-END IF
- IF-THEN-ELSE-END IF
- IF-THEN-ELSEIF-END IF
- IF-THEN-ELSEIF-THEN-ELSE-END IF

ELSEIF Rule

You can specify an unlimited number of ELSEIF clauses in an IF statement, but each must be associated with a condition as in the case of the initial IF clause.

IF-THEN-END IF

This form of IF performs the statements within the IF and END IF bounds when *conditional_expression* evaluates to TRUE.

The following statement is an example of IF-THEN-END IF:

```
IF hNoAccts = 1 THEN
  INSERT INTO temp_table VALUES (:hNoAccts,
  'One Customer');
END IF;
```

IF-THEN-ELSE-END IF

This form of IF performs the statements within the IF and ELSE bounds when *conditional_expression* evaluates to TRUE. Otherwise, the statements within the ELSE and END IF bounds perform.

In the following example, only one (and perhaps none) of the specified INSERT statements performs, depending on the value for `hNoAccts`.

```
IF hNoAccts = 1 THEN
  INSERT INTO temp_table VALUES (:hNoAccts,
  'One customer');
ELSE
  INSERT INTO temp_table VALUES (:hNoAccts,
  'More than one customer');
END IF;
```

IF-THEN-ELSEIF-END IF

This form of IF behaves as described in the table below:

Stage	Process
1	The statements between the IF and ELSEIF boundaries perform when IF evaluates to TRUE. Control then passes to the statement following END IF.
2	The statements associated with each ELSEIF are evaluated for their truth value.
3	When a statement associated with an ELSEIF evaluates to TRUE, then the statements within its block perform. Subsequent ELSEIF clauses do not perform.
4	When no statement in the IF/END IF block evaluates to TRUE, then none of the statements can perform.

Example 1

In the following example, either one (and only one) of the ELSEIF clauses performs its associated DML statement or none does, depending on the value for `hNoAccts`.

```
IF hNoAccts = 1 THEN
  INSERT INTO temp_table VALUES (:hNoAccts, 'One
  customer');
ELSEIF hNoAccts = 0 THEN
  INSERT INTO temp_table VALUES (:hNoAccts,
  'No customer');
END IF;
```

Example 2

In the following example, one (and only one) of the ELSEIF clauses performs its associated DML statement or none does, depending on the value for `hNoAccts`.

```
IF hNoAccts = 1 THEN
  INSERT INTO temp_table VALUES (:hNoAccts, 'One
  customer');
ELSEIF hNoAccts = 0 THEN
     INSERT INTO temp_table VALUES (:hNoAccts,
     'No customer');
ELSEIF hNoAccts < 0 THEN
     INSERT INTO temp_table VALUES (:hNoAccts,
  'Unknown customer');
END IF;
```

IF-THEN-ELSEIF-ELSE-END IF

This form of IF behaves as described in the table below:

Stage	Process
1	The statements between the IF and ELSEIF boundaries perform when IF evaluates to TRUE. Control then passes to the statement following END IF.
2	The statements associated with each ELSEIF are evaluated for their truth value.
3	When a statement associated with an ELSEIF evaluates to TRUE, then the statements within its block perform. Subsequent ELSEIF clauses do not perform even if they evaluate to TRUE.
4	When no statement in any of the IF/ELSEIF blocks evaluates to TRUE, then the statement associated with the ELSE clause performs.

Example 1

In the following example, one (and only one) of the DML statements associated with an ELSEIF or ELSE clause performs, depending on the value for `hNoAccts`.

```
IF hNoAccts = 1 THEN
  INSERT INTO temp_table VALUES (:hNoAccts, 'One
  customer');
ELSEIF hNoAccts = 0 THEN
     INSERT INTO temp_table VALUES (:hNoAccts,
     'No customer');
ELSE
     INSERT INTO temp_table VALUES (:hNoAccts,
  'More than one customer');
END IF;
```

Example 2

In the following example, one (and only one) of the DML statements associated with an ELSEIF or ELSE clause performs, depending on the value for `hNoAccts`.

```
IF hNoAccts = 1 THEN
  INSERT INTO temp_table VALUES (:hNoAccts, 'One
  customer');
ELSEIF hNoAccts = 0 THEN
     INSERT INTO temp_table VALUES (:hNoAccts,
     'No customer');
ELSEIF hNoAccts < 0 THEN
     INSERT INTO temp_table VALUES (:hNoAccts,
     'Non-valid customer');
ELSE
     INSERT INTO temp_table VALUES (:hNoAccts,
     'More than one customer');
END IF;
```

ITERATE

Purpose

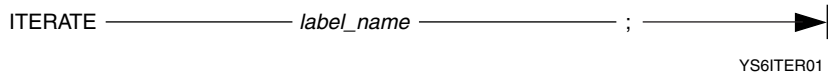
Terminates the execution of an iterated SQL statement and begins the next iteration of the iterative statement in the loop.

Invocation

Executable

Stored procedures only.

Syntax



where:

Syntax element …	Specifies …
label_name	the name of the label on which iteration is to be performed.

ANSI Compliance

ITERATE is ANSI SQL-99-compliant.

Authorization

None.

Rules for ITERATE

The following rules apply to ITERATE:

- ITERATE is not an independent statement. You can only use it with a FOR, LOOP, REPEAT, or WHILE iteration statement.
- A statement label *must* follow the ITERATE keyword immediately.
- ITERATE statement cannot reference the label associated with the BEGIN - END compound statement within which the ITERATE is embedded.
- The statement label *must* be associated with the iteration statement within which the ITERATE is embedded.
- If you specify an ITERATE inside nested FOR loops and it refers to a label associated with an outer iteration statement, then all cursors opened within the outer iteration statement are closed before performing the next iteration.

Actions Performed by ITERATE

ITERATE performs the following actions depending on the referenced iteration statement:

IF ITERATE specifies the label of ...	AND ...	THEN ...	ELSE ...
FOR statement	the existence of a next row in the cursor is verified	performance continues with the next statement in the FOR loop	the cursor is closed and performance continues with the next statement outside the corresponding END FOR terminator for the FOR loop.
LOOP statement		the first statement inside the LOOP block performs unconditionally.	
REPEAT statement		performance continues with the first statement inside the REPEAT loop without any evaluation of the UNTIL clause.	
WHILE statement	the conditional expression for the WHILE evaluates to TRUE	performance continues with the first statement inside the WHILE loop	performance continues with the next statement outside the corresponding END WHILE terminator for the loop.

Example 1

The following example illustrates a valid use of ITERATE to iterate a WHILE statement.

```
SELECT minNum INTO :hminNum FROM limits
   WHERE LIMIT_TYPE = 'HIGHNUM';
L1:
WHILE hCounter > 0 DO
   INSERT INTO transaction (trans_num, account_num)
     VALUES (:hCounter, :hAccountNum);
   SET hCounter = hCounter - 1;
   IF hCounter >= hminNum THEN
      ITERATE L1;
   END IF;
   -- The following two statements perform only when
   -- hCounter < hminNum
   UPDATE limit SET minNum = hCounter;
   SET hminNum = hCounter;
END WHILE L1;
```

Example 2

The following example illustrates the use of an ITERATE statement to iterate an outer loop.

```
LOOP1:
WHILE hCounter > 0 DO
   SELECT highNum INTO :maxNum FROM limits
WHERE LIMIT_TYPE = 'HIGHNUM';
L1:
LOOP
      INSERT INTO transaction (trans_num,
      account_num) VALUES (:hCounter, :hAccountNum);
    SET hCounter = hCounter - 1;
    IF (hCounter = 10) THEN
      IF (hOnceIterated = 0) THEN
      SET hOnceIterated = 1);
        ITERATE LOOP L1;
      END IF;
    END IF;
    -- The following statement performs only if
    -- hCounter <> 10 or hOnceIterated <> 0
    SET hNum = hNum + 10;
   END LOOP L1;
   IF hCounter >= MaxNum THEN
      ITERATE LOOP1;
   END IF;
   -- The following statement performs only if
   -- hCounter < MaxNum.
      INSERT INTO transaction (trans_num,
      account_num) VALUES (:hCounter, h:AccountNum);
END WHILE LOOP1;
UPDATE transaction
  SET account_num = :hAccountNum + 10;
```

Example 3

The following example demonstrates the use of ITERATE to iterate outside a FOR loop. When there are no more rows to fetch, the cursor closes and control iterates out of the FOR loop.

```
L1:
LOOP
   INSERT INTO transaction (trans_num, account_num)
     VALUES (:hCounter, :hAccountNum);
   SET hCounter = hCounter - 1;
   FOR RowPointer AS c_customer CURSOR FOR
      SELECT CustomerNumber AS Number
            ,CustomerName AS Name
            ,(Amount + 10000) a
        FROM customer
   DO
      SET hCustNum = RowPointer.Number;
      IF hCustNum >= 100 THEN
         ITERATE L1;
      END IF;
      -- The following statements perform only if
      -- hCustNum < 100; else the cursor closes before
      -- iterating outside the FOR loop block.
      SET hCustName = RowPointer.Name;
      SET hAmount = RowPointer.a +
                    RowPointer.a * 0.20;
      INSERT INTO Cust_temp VALUES (:hCustNum,
        :hCustName);
   END FOR;
   SET hNum = hNum + 10;
END LOOP L1;
```

LEAVE

Purpose

Breaks execution of a labeled iteration or compound statement and continues execution outside an iteration statement.

Invocation

Executable.

Stored procedures only.

Syntax



where:

Syntax element …	Specifies …
label_name	the name of the label for the iteration statement or BEGIN - END block to be terminated by the LEAVE.

ANSI Compliance

LEAVE is ANSI SQL-99-compliant.

Authorization

None.

Rules for LEAVE

The following rules apply to LEAVE:

- You can specify LEAVE anywhere within the scope of its referred label.
- The label must be associated with either of the following:
  - An iteration statement
  - The BEGIN - END compound statement in which you embed the LEAVE statement.

Actions Performed by LEAVE

- If the LEAVE statement references a label associated with a compound statement, it terminates the execution of that compound statement.

  This action is treated as successful completion of the stored procedure.
- If LEAVE references the label of an iteration statement (FOR, LOOP, REPEAT, or WHILE), it breaks the iteration and passes control to the next statement outside the label.
- LEAVE performs the following actions depending on the referenced iteration statement:

IF LEAVE specifies the label of . . .	THEN . . .
Any statement	LEAVE terminates the performance of its associated iteration statement and any iteration statements nested within.
FOR statement	the cursor closes before control is transferred to the next statement outside the referenced iteration statement.
An outer iteration statement containing FOR statement(s)	all open cursors specified in the iteration statement are closed before control transfers to the next statement following the iteration statement.
The BEGIN - END compound statement to which the LEAVE belongs	all open cursors declared in that compound statement are closed and then control is transferred to the statement following that compound statement caller, terminating the procedure. A success or “ok” response is returned to the procedure.

- Any error condition encountered while closing the cursor is reflected in the status variables.

  Example:

```
SQLCODE=7600, SQLSTATE='T7600', ACTIVITY_COUNT=0.
```

Example 1

The following example illustrate a valid use of LEAVE to terminate the execution of a stored procedure:

```
CREATE PROCEDURE spSample()
SPLABEL:
BEGIN
   DECLARE vCount INTEGER DEFAULT 0;
   WHILE vCount <= 10
   DO
      UPDATE table_1
       SET table_1.column_1 = :vCount
       WHERE table_1.column_2 > 10;
     IF ACTIVITY_COUNT = 0 THEN
          LEAVE SPLABEL;
     END IF;
   END WHILE;
END;
```

Example 2

The following example illustrates a valid use of LEAVE with an iteration statement:

```
LABEL1:
WHILE i < 10 DO
   UPDATE table_1
    SET table_1.column_1 = :i
    WHERE table_1.column_2 > 10;
  IF ACTIVITY_COUNT > 1 THEN
      LEAVE LABEL1;
  END IF;
   SET i = i+1;
END WHILE LABEL1;
```

LOOP

Purpose

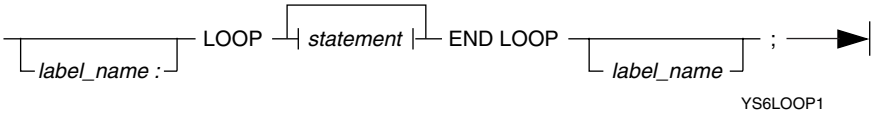
Repeats the performance of one or more statements embedded within the defined iteration statement.

Invocation

Executable.

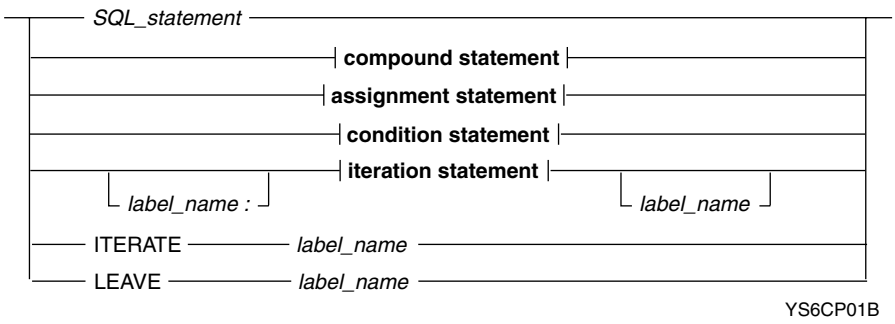
Stored procedures only.

Syntax



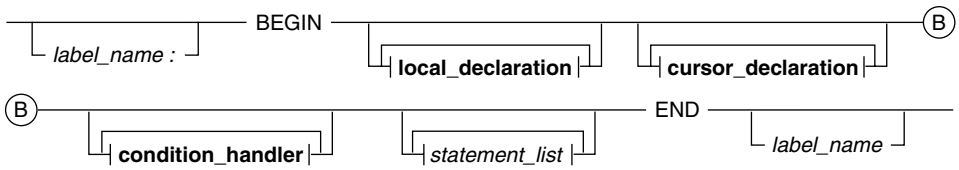
YS6LOOP1

statement



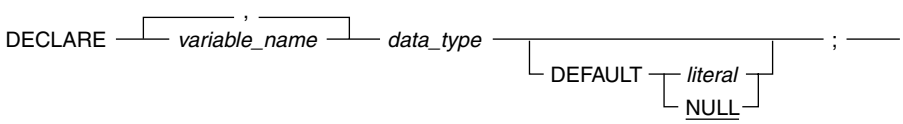
YS6CP01B

compound statement

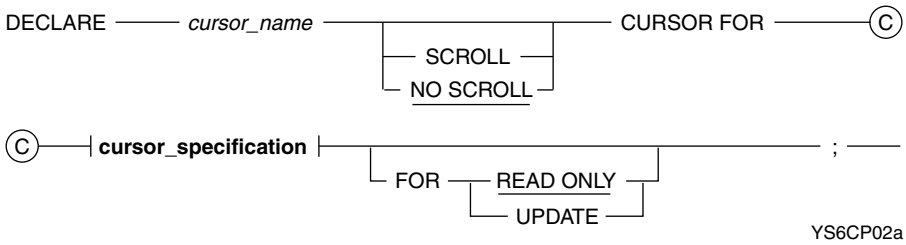


YS6CP01C

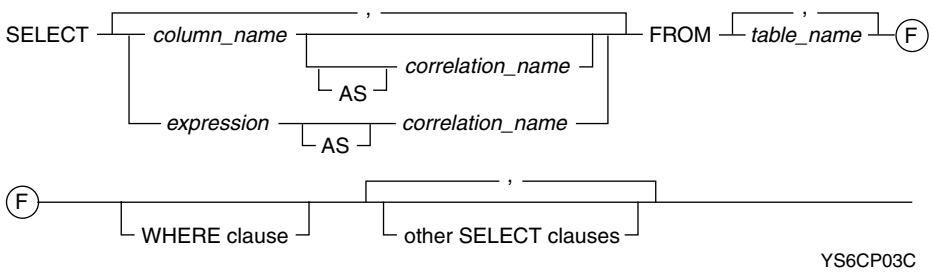
local_declaration



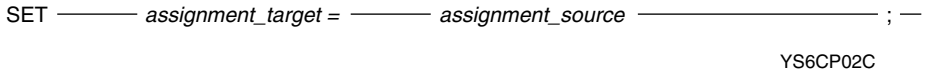
cursor_declaration



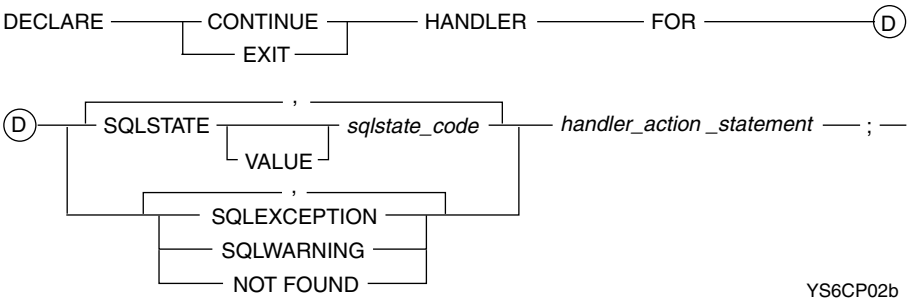
cursor-specification



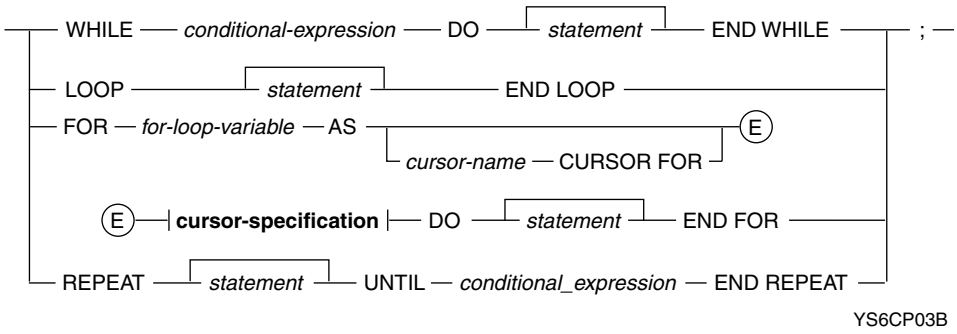
assignment statement



condition-handler



iteration statement



YS6CP03B

where:

Syntax element . . .	Specifies . . .
label_name	an optional label for the LOOP statement. If an ending-label is specified, you must specify a beginning-label that is equivalent to the ending-label. The beginning-label must be terminated by a COLON character (:). The label name of the BEGIN - END compound statement cannot be reused in an iteration statement. One label name cannot be reused within one group of nested LOOP statements, but can be reused for different non-nesting iteration statements.
statement	a statement list to be processed unconditionally. The list can contain any of the following: • SQL DML, DDL or DCL statements, including dynamic SQL. • Control statements, including BEGIN - END.

ANSI Compliance

LOOP is ANSI SQL-99-compliant.

Authorization

None.

Rules for LOOP

The following rules apply to LOOP:

- You can qualify LOOP with a statement label.

  A LEAVE statement specified within the LOOP breaks the iteration statement, passing control to the next statement following the statement with that label.
- You must specify a LEAVE statement inside the LOOP statement to ensure normal termination of the statement.

  If you do not, the loop iterates continuously and can only be stopped by an asynchronous abort.

LOOP-Terminating Errors

The following are conditions that cause LOOP-terminating errors:

- If a statement in the LOOP raises an exception condition and a CONTINUE handler has been declared for that condition, then the stored procedure execution continues.
- If an EXIT handler has been declared, then the statement terminates the stored procedure execution.
- If a statement within the loop raises an error condition and its associated SQLSTATE code is not defined for a handler, then both the loop and the stored procedure terminate.

Example

The following LOOP statement is valid:

```
L1:
LOOP
  INSERT INTO transaction (trans_num, account_num)     VALUES
(:hCounter, :hAccountNum);
  SET hCounter = hCounter - 1;
  IF hCounter = 0 THEN
     LEAVE L1;
  END IF;
END LOOP L1;
```

REPEAT

Purpose

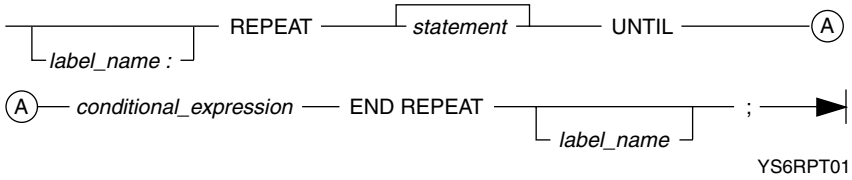
Repeats the execution of one or more statements until the specified condition evaluates to true.

Invocation

Executable.

Stored procedures only.

Syntax



where:

Syntax element …	Specifies …
label_name	an optional label for the REPEAT statement. If an ending-label is specified, you must specify a beginning-label that is equivalent to the ending-label. The beginning-label must be terminated by a COLON character (:). The label name of the BEGIN - END compound statement cannot be reused in an iteration statement. One label name cannot be reused within one group of nested REPEAT statements, but can be reused for different non-nesting iteration statements.
statement	a statement list to be performed. The list can contain any of the following: • DML, DDL or DCL statements, including dynamic SQL. • Control statements, including BEGIN - END.
conditional_expression	a boolean condition used to evaluate whether a statement or statements embedded within the REPEAT loop should be performed. You cannot use IN and NOT IN operators if the conditional list contains any local variables, parameters, or cursor aliases. OUT parameters are not allowed in conditional_expression.

ANSI Compliance

REPEAT is ANSI SQL-99-compliant.

Authorization

None.

Rules for REPEAT

You can qualify the REPEAT statement with a label name. If a label name is provided for REPEAT, then:

- A LEAVE statement inside the block can use that label name to leave the REPEAT statement.
- If an ITERATE statement is specified within the block, and it refers to the label associated with REPEAT, the execution is continued from the beginning of the REPEAT statement without evaluating the conditional expression specified with the UNTIL clause.

Exception Handling in REPEAT Statements

If a statement within the REPEAT statement, or the conditional expression of the UNTIL clause raises an exception and the stored procedure contains a handler to handle that exception condition, the behavior is identical to exceptions occurring within a WHILE statement.

See "DECLARE HANDLER (Control Statement Handling)" on page 9-105 for rules governing exceptions.

Difference Between REPEAT and WHILE

REPEAT – END REPEAT is similar to the WHILE – END WHILE statement, with some differences:

REPEAT...	WHILE ...
makes the first iteration unconditionally. REPEAT always performs the sequence of statements at least once.	makes the first iteration and subsequent iterations only if a specified condition is true.
performs statements until a specified condition is met.	performs statements as long as a specified condition is met.

Example

The following example shows the use of a REPEAT statement:

```
CREATE PROCEDURE ProcessTrans(IN pAcctNum INTEGER
                              IN pStartTrans INTEGER,
                              IN pEndTrans INTEGER )
BEGIN
    DECLARE vTransNum INTEGER;
    SET vTransNum = pStartTrans;
    …;
    REPEAT
        INSERT INTO trans (trans_num, acct_nbr)
               VALUES (:vTransNum, :pAcctNum);
        SET vTransNum = vTransNum + 1;
      UNTIL vTransNum > pEndTrans
    END REPEAT;
    …;
END;
```

SET

Purpose

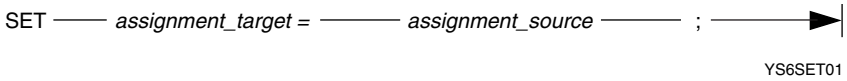
Assigns a value to a local variable or parameter in a stored procedure.

Invocation

Executable.

Stored procedures only.

Syntax



where:

Syntax element ...	Specifies ...
assignment_target	the name of the variable or parameter to be assigned a value.
assignment_source	the value to be assigned to *assignment_target*.

ANSI Compliance

SET is ANSI SQL-99-compliant.

Authorization

None.

Rules for SET

The following rules apply to SET:

- All valid expressions except those containing subqueries are permitted in a SET statement assignment source.
- Both assignment target and assignment source must be specified.
- Assignment target is always on the lefthand side (LHS) of the SET expression.
- Assignment source is always on the righthand side (RHS) of the SET expression.
- Assignment target must be either a local variable or OUT or INOUT parameter.
- Assignment source can be a local variable, IN or INOUT parameter, for-loop correlation name, status variable, or constant expression.

- The data type of the assignment source must be compatible with the data type specified for the assignment target.

FOR this component of a SET assignment . . .	The following constructs are . . .	
	Valid	**Not Valid**
assignment target	• local variable name • OUT or INOUT parameter name	• status variables • FOR loop column and correlation names • IN parameters
assignment source	a literal or an expression containing one of the following: • local variables • IN or INOUT parameters • FOR loop column and correlation names when the SET statement is within the scope of the FOR statement • status variables	• OUT parameter • FOR loop column and correlation names when the SET statement is not within the scope of the FOR statement.

Example

The following example illustrates a valid use of the SET statement to assign values to variables and parameters.

```
SET hNoAccts = hNoAccts + 1;
SET hErrorText = 'SQLSTATE: '||sqlstate||
', SQLCODE: '||sqlcode||', ACTIVITY_COUNT: '
||activity_count;
```

WHILE

Purpose

Repeats the execution of a statement or statement list while a specified condition evaluates to true.

Invocation

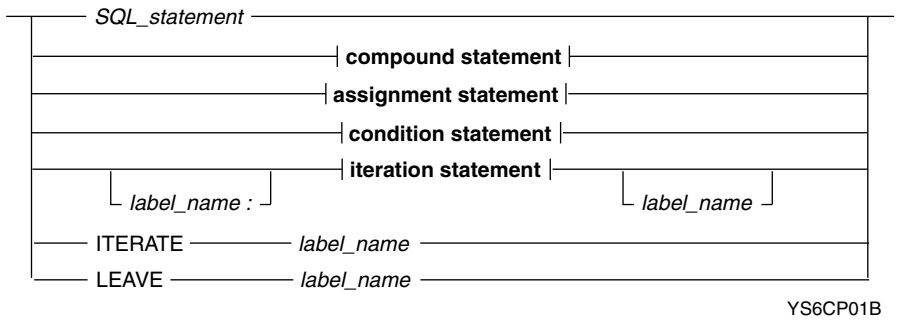
Executable.

Stored procedures only.

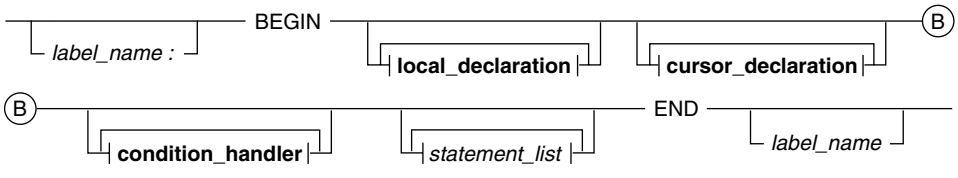
Syntax



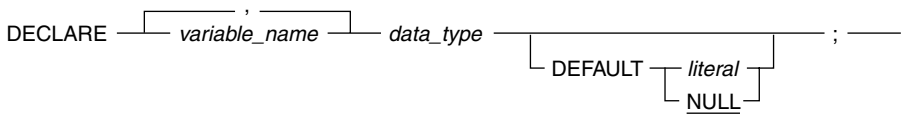
statement



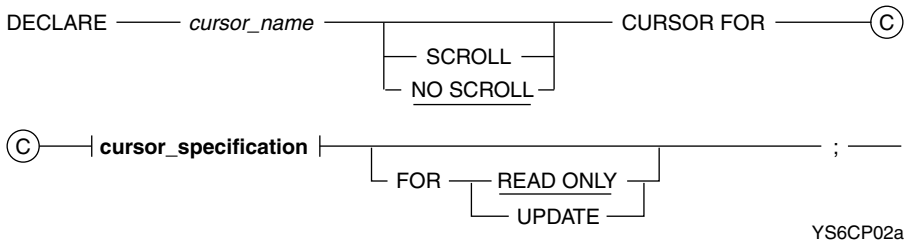
compound statement



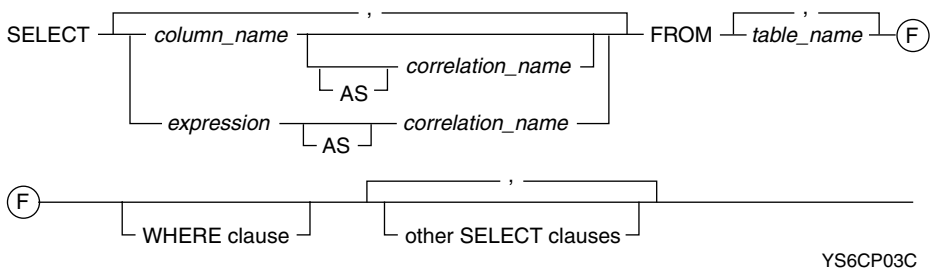
local_declaration



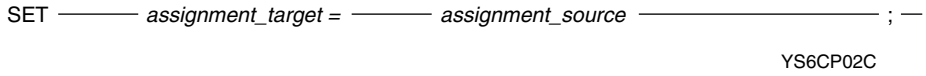
cursor_declaration



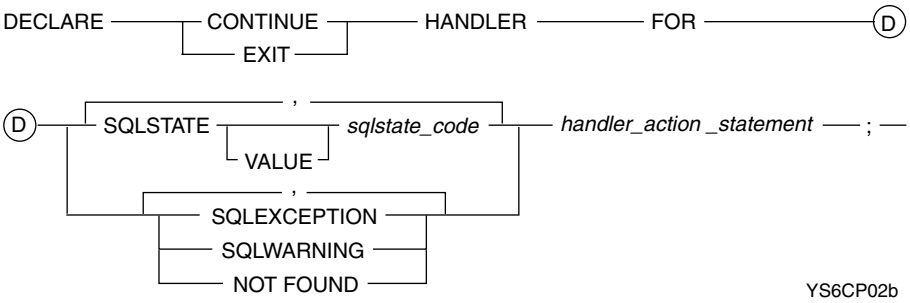
cursor-specification



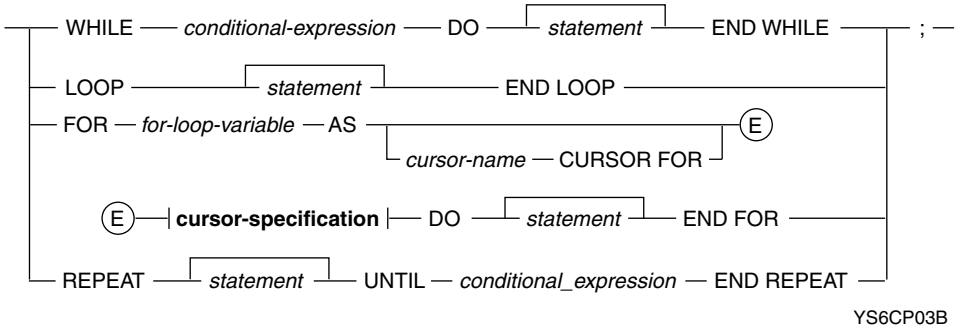
assignment statement



condition-handler



iteration statement



YS6CP03B

where:

Syntax element …	Specifies …
label_name	an optional label for the WHILE statement. If an ending-label is specified, you must specify a beginning-label that is equivalent to the ending-label. The beginning-label must be terminated by a COLON character (:). The label name of the BEGIN - END compound statement cannot be reused in an iteration statement. One label name cannot be reused within one group of nested WHILE statements, but can be reused for different non-nesting iteration statements.
conditional_expression	a boolean condition used to evaluate whether a statement or statements embedded within the WHILE loop should be performed. You cannot use IN and NOT IN operators if the conditional list contains any local variables, parameters, or cursor correlation names. OUT parameters are not allowed in conditional_expression.
statement	a list of statements to be performed. The list can contain any of the following: • DML, DDL or DCL statements, including dynamic SQL. • Control statements, including BEGIN - END.

ANSI Compliance

WHILE is ANSI SQL-99-compliant.

Authorization

None.

Rules for WHILE

The following rules apply to WHILE:

- You can qualify WHILE with a label.
- You can specify a LEAVE or ITERATE statement within a WHILE statement.

  See "ITERATE" on page 9-125 and "LEAVE" on page 9-129 for details.

Example 1

```
WHILE hCounter > 0
   DO
      INSERT INTO transaction (trans_num, account_num)
        VALUES (:hCounter, :hAccountNum);
      SET hCounter = hCounter - 1;
   END WHILE;
```

Example 2

LOOP1:

```
   WHILE hCounter > 0
   DO
       SELECT highNum INTO :maxNum
       FROM limits WHERE LIMIT_TYPE = 'HIGHNUM';
       IF hCounter >= MaxNum THEN
           LEAVE LOOP1;
       END IF;
       INSERT INTO transaction (trans_num, account_num)
         VALUES (:hCounter, :hAccountNum);
       SET hCounter = hCounter - 1;
   END WHILE;
```

Stored Procedures and Tactical Queries

Introduction

Stored procedures can be of great benefit for some tactical query applications. Some examples of using stored procedures to process complex updates, reduce workload overhead, and maintain security audits are described in this section. A comparison of the relative efficiency of stored procedures and macros for different tactical query applications is also provided.

Using Stored Procedures to Perform Complex Tactical Updates

Complex updates in a tactical query are often more easily processed if they are disassembled and then integrated within a stored procedure. The computational completeness offered by stored procedure control statements permits you to perform SQL statements iteratively and conditionally, which not only makes what they are doing more explicit than the nested subqueries required to write many complex updates, but can also make them easier to process.

For example, to perform the following complex update, a stored procedure would perform two single-AMP statements, each of which applies only single row hash locks.

```
UPDATE orders
SET o_orderpriority = 5
WHERE o_orderkey = 39256
AND EXISTS
  (SELECT * FROM lineitem
   WHERE l_orderkey = o_orderkey);
```

The following two EXPLAIN reports represent the disassembled SQL statements that could be written inside the stored procedure to replace the previously described complex update.

```
EXPLAIN
SELECT *
FROM lineitem
WHERE l_orderkey = 39256;
```

```
Explanation
------------------------------------------------------------------------
  1) First, we do a single-AMP RETRIEVE step from CAB.lineitem by
     way of the primary index "CAB.lineitem.L_ORDERKEY = 39256"
     with no residual conditions into Spool 1, which is built locally
     on that AMP.  The input table will not be cached in memory, but it
     is eligible for synchronized scanning.  The size of Spool 1 is
     estimated with high confidence to be 4 rows.  The estimated time
     for this step is 0.15 seconds.
  -> The contents of Spool 1 are sent back to the user as the result of
     statement 1.  The total estimated time is 0.15 seconds.
```

Using the appropriate conditional logic, the procedure would then perform the second statement only if at least one lineitem row is returned from the first request.

```
EXPLAIN
UPDATE orders
SET o_orderpriority = 5
WHERE o_orderkey = 39256

Explanation
-----------------------------------------------------------------------
  1) First, we do a single-AMP UPDATE from CAB.orders by way of
     the unique primary index "CAB.orders.O_ORDERKEY = 39256"
     with no residual conditions.
```

You would need to code these two statements within an explicit transaction, using BEGIN TRANSACTION and END TRANSACTION statements to specify the transaction boundaries.

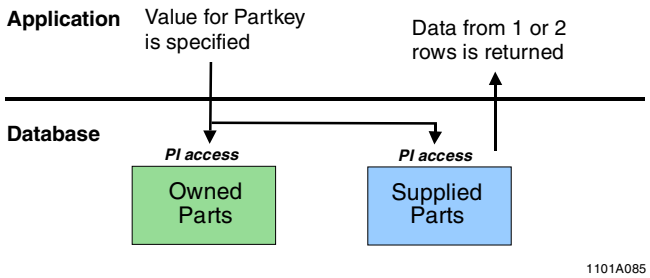
Because stored procedures cannot return multirow answer sets, macros are usually a better approach to use when you need to return more than one row from a table.

Using Stored Procedures to Eliminate Unnecessary Database Work

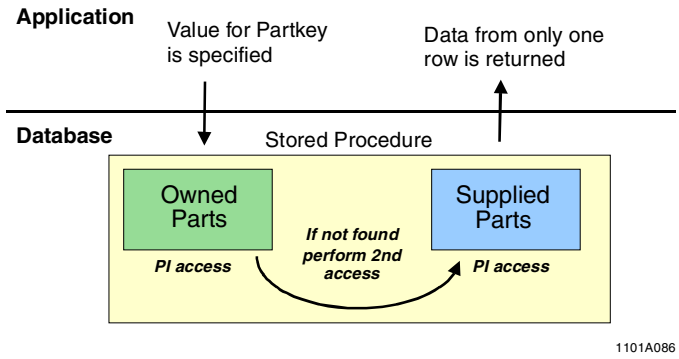
Suppose you have an application that uses data from only one of two possible tables, but you cannot know which of the two tables is required until you attempt to access the first. Using a stored procedure to solve this problem can eliminate processing that would otherwise be necessary to solve the problem.

To illustrate, consider a database with two tables that contain parts information: owned_parts and supplied_parts. Assume you have a business rule that says that you return owned_parts data if it exists, and only return supplied_parts data in the absence of owned_part data having the specified prime key value. Also assume that you have a tactical query that probes for parts data by specifying a part key.

Using a macro to perform this query, you would have to access both tables for each request, then pass the data to the application to determine which to use when two rows are returned, as illustrated in the following graphic.

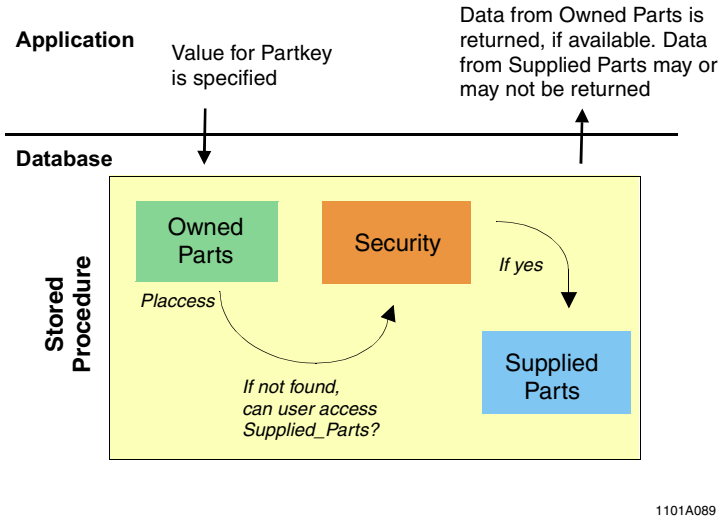


This extra work is unnecessary if you write a stored procedure to solve the problem. You can code logic in a stored procedure to determine if a row has been found in the owned_parts table, and, if so, to then return to the application without attempting to access the supplied_parts table. The following graphic provides a high level picture of the processing involved.



Security and Auditing

Considering the same parts example from "Using Stored Procedures to Eliminate Unnecessary Database Work" on page 9-146, suppose that only select users are permitted to access the supplied_parts table. With a modification to the previous stored procedure, you can code logic to check the privileges for the user who submitted the procedure against a security table. The procedure then checks permissions before access to supplied_parts is allowed without the user even being aware that his access had been monitored, as illustrated by the following graphic:

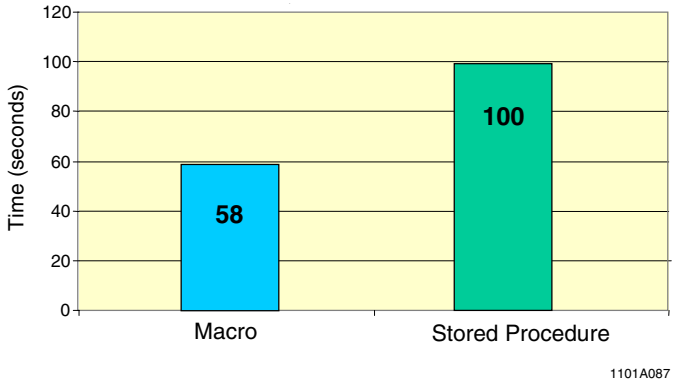


A similar approach could be taken to validate that certain data in the supplied_parts table can only be viewed by certain users, but not by others.

When Macros Are a Better Choice Than Stored Procedures for Tactical Queries

For very simple requests, macros are almost always a better choice than stored procedures. This is because stored procedures require certain overhead that macros do not.

In the test represented in the following illustration, a macro that does a single primary index-based SELECT that returns only one row was performed 10 000 times. Different input variables were supplied for each iteration. The total time to perform this work using a macro is then compared with the time it took a stored procedure that performs the same SELECT statement to complete the task.



The numbers show clearly that for a simple request, a macro is nearly twice as efficient as an equivalent stored procedure.The following table defines the code used to compare these two applications:

Macro	Stored Procedure
CREATE MACRO cabmac0 (key1 INTEGER) AS (SELECT c_acctbal FROM customer WHERE c_custkey = :key1;);	CREATE PROCEDURE cabproc0 (IN key1 INTEGER, OUT price1 DECIMAL(15,2)) BEGIN SELECT c_acctbal INTO :price1 FROM customer WHERE c_custkey=:key1; END;
EXEC cabmac0 (55555);	CALL cabproc0 (55555, price1);

As concurrency levels increase, the performance benefit of macros is even greater because of the increased overhead of launching stored procedures.

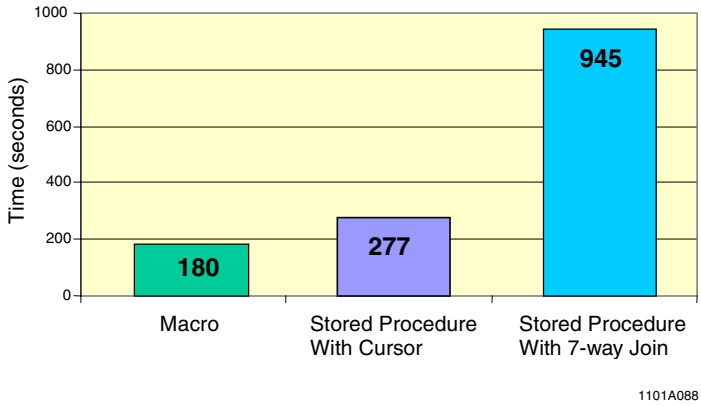
A second situation where macros offer an advantage for tactical queries is when several independent statements can be performed in parallel. The multistatement request capabilities of a macro give it an advantage over the stored procedure requirement to submit each SQL statement individually.

A third case exists when a statement returns multiple rows. A stored procedure uses OUTPUT parameters to return information. If more than row is expected, the macro simply returns all the rows to the requestor.

In contrast, a stored procedure must incorporate additional complexity to return multiple rows, such as:

- Inserting the rows into a temporary table to be read in subsequent processing.
- Creating multiple sets of output parameters and looping through the multiple rows with a cursor.
- Performing an *n*-way join to produce one large answer set that incorporates all NUPI rows.

The following graph illustrates test results using some of these different approaches when 1 000 jobs of each type are submitted serially. In every case the macro performs better than its stored procedure counterpart.



The following table summarizes the differences between macros and stored procedures:

Macro	Stored Procedure
Limited procedural logic.	Sophisticated procedural logic.
Can return multirow answer sets for the same request.	Returns a single set of values.
Multistatement request parallelizes multiple single row statements.	One request per individually processed statement.
Macro text stored in dictionary.	Stored procedure text stored in user database.
Can EXPLAIN a macro.	Cannot EXPLAIN a stored procedure. Instead must EXPLAIN each individual stored procedure SQL statement individually.
Can be invoked by a trigger.	Cannot be invoked by a trigger.

Debugging Stored Procedures

Introduction

This section provides guidelines for debugging stored procedure-based applications.

The following act as debugging tools:

- SQL INSERT statement
- Special purpose stored procedure for logging information

Though no method is perfect, these debugging and testing techniques help minimize bugs.

Comparative Merits of the Debugging Methods

The following table describes the advantages and disadvantages of these two methods. The methods need to be evaluated based on your requirements.

This debugging method . . .	Offers these advantages . . .	And has these disadvantages . . .
INSERT statement	log entries can be inserted into a user-defined log table and the results can be viewed by querying the log table after execution of the stored procedure.	you need to disable or remove the INSERT statement(s) from the stored procedure body after debugging and recompile the procedure.
'Debug' stored procedure	defines a standard procedure to debug stored procedures.	you need to disable or remove the CALL statement(s) from the stored procedure body after debugging and recompile the procedure.

Using SQL INSERT Statements for Debugging

You can use the SQL INSERT statements in the stored procedure to insert any values or text in any table.

Example: INSERT

Consider the following stored procedure definition:

```
REPLACE PROCEDURE spRow (OUT pRowCount INTEGER)
BEGIN
   DECLARE EXIT HANDLER
   FOR SQLEXCEPTION
   BEGIN
      INSERT ErrorLogTable ('spRow', :SQLSTATE,
         :SQLCODE);
   END;
```

```
   SET pRowCount = 0;
   FOR vFor AS cName CURSOR FOR
      SELECT c1 as a, c2 * 10 as b
         FROM ValueTable
   DO
      SET pRowCount = pRowCount + 1;
         INSERT LogTable (:pRowCount, :vFor.a, :vFor.b);
         …
         …
         …
   END FOR;
END;
```

When the stored procedure is performed, the INSERT statement specified in the FOR statement is performed for each row fetched from the cursor. It inserts the values of row count, column c1, and column c2 of table ValueTable.

If, during the stored procedure execution, an exception or completion condition is raised and it is handled using the declared generic condition handler, the INSERT statement specified within the handle is performed. It inserts the stored procedure name, and values of SQLCODE and SQLSTATE status variables in the ErrorLogTable. This table can be queried to identify if there was any exception during stored procedure execution.

Using Debug Stored Procedures

You can write a site-specific or application-specific stored procedure that logs any given text in a standard log table with user-id and so on. Alternatively, errors and tracing information can be reported using the results viewed from the DBS I/O window.

Example: Debug Stored Procedure with INSERTs

Consider the following stored procedure definition:

```
CREATE PROCEDURE LogDatabase.spDebug (IN spName CHAR(30), IN Text
CHAR(255))
BEGIN
   -- Exit in case of any exception.
   DECLARE EXIT HANDLER
   FOR SQLEXCEPTION
    BEGIN
      INSERT ErrorLogTable (:spName, :SQLSTATE,
            :SQLCODE);
      END;

   -- Log the text in the DebugTable.
   INSERT INTO LogDatabase.DebugTable(:spName,
      USER, CURRENT_DATE, CURRENT_TIME, :Text)
END;
```

The procedure spDebug is created in a predefined database, LogDatabase. The procedure inserts row in an existing table DebugTable. It accepts two input arguments, a stored procedure name and the text to be logged. The caller needs EXECUTE PROCEDURE privilege on this stored procedure in order to use it in any other stored procedure.

The following stored procedure calls the LogDatabase.spDebug:

```
CREATE PROCEDURE spRow (OUT pRowCount INTEGER)
BEGIN
   DECLARE ErrorText CHAR(255) DEFAULT NULL:
   DECLARE EXIT HANDLER
   FOR SQLEXCEPTION
   BEGIN
      SET ErrorText = 'In exception handler…' ||       'SQLCODE:
' || SQLCODE || ' SQLSTATE: ' ||       SQLSTATE;
      CALL LogDatabase.spDebug ('spRow', :ErrorText);
   END;
   SET pRowCount = 0
   FOR vFor AS cName CURSOR FOR
       SELECT c1 as a, c2 * 10 as b
          FROM ValueTable
      DO
      SET pRowCount = pRowCount + 1;
      SET ErrorText = 'RowCount: ' || pRowCount ||
      'Values: ' || vFor.a || ' ' || vFor.b;
      CALL LogDatabase.spDebug ('spRow', :ErrorText);
            …
            …
            …
   END FOR;
END;
```

Sample Stored Procedure

Introduction

This section provides a sample stored procedure that uses most of the features of Teradata stored procedures. The sample stored procedure is not recommended for real use.

The sample stored procedure includes multiple parameters, local variable declarations, cursors (FOR cursor and cursor declaration), condition handlers, nested compound statements, control statements, DML statements, and ANSI style comments.

Usage Notes

With respect to the sample stored procedure given here, we assume that:

- The user has CREATE PROCEDURE privilege on the current default database.
- The procedure is being created in the database owned by the user, so that the creator is also the immediate owner of the procedure.

  That enables the user to include DDL, DCL and dynamic SQL statements in the stored procedure, and avoid many possible compilation errors.
- The tables tBranch, tAccounts, tDummy, tDummy1 and Proc_Error_Tbl exist in the current default database.
- The stored procedure GetNextBranchId also exists in the current default database.
- The new stored procedure supports only 1 000 accounts per branch.

Example Table Definitions

The following CREATE TABLE statements define two important tables, tAccounts and Proc_Error_Tbl, that are used in the sample stored procedure. Note that all tables and the stored procedures referenced in the stored procedure body must also be created.

This DDL defines the accounts table:

```
CREATE MULTISET TABLE sampleDb.tAccounts, NO FALLBACK,
     NO BEFORE JOURNAL,
     NO AFTER JOURNAL
    (
     BranchId INTEGER,
     AccountCode INTEGER,
     Balance DECIMAL(10,2),
         AccountNumber INTEGER,
        Interest DECIMAL(10,2))
PRIMARY INDEX (AccountNumber) ;
```

This DDL defines the error table:

```
CREATE MULTISET TABLE sampleDb.Proc_Error_Tbl ,NO FALLBACK ,
     NO BEFORE JOURNAL,
     NO AFTER JOURNAL
     (
     sql_state CHAR(5) CHARACTER SET LATIN CASESPECIFIC,
     time_stamp TIMESTAMP(6),
     Add_branch CHAR(15) CHARACTER SET LATIN CASESPECIFIC,
     msg VARCHAR(40) CHARACTER SET LATIN CASESPECIFIC)
PRIMARY INDEX ( sql_state );
```

Stored Procedure Definition

The CREATE PROCEDURE statement below creates the sample stored procdure. The definition is also called the "source text" for the stored procedure.

This CREATE PROCEDURE statement creates a procedure named *AddBranch* that supports the internal functions of a bank. Its purposes are to:

- Capture and adds details of the new branch to the table tBranch
- Assign a BranchId to a new branch
- Add details of new accounts of a branch to the table tAccounts
- Update balances and interest in the accounts contained in the table tAccounts for the new branch.

```
CREATE PROCEDURE AddBranch (
                        OUT  oBranchId    INTEGER,
                         IN  iBranchName  CHAR(15),
                         IN  iBankCode    INTEGER,
                         IN  iStreet      VARCHAR(30),
                         IN  iCity        VARCHAR(30),
                         IN  iState       VARCHAR(30),
                         IN  iZip         INTEGER
                        )
Lmain: BEGIN
     -- Lmain is the label for the main compound statement

   -- Local variable declarations follow
   DECLARE hMessage CHAR(50) DEFAULT
              'Error: Database Operation …';
   DECLARE hNextBranchId INTEGER;
   DECLARE hAccountNumber INTEGER DEFAULT 10;
   DECLARE hBalance INTEGER;

    -- Condition Handler Declarations
   DECLARE CONTINUE HANDLER FOR SQLSTATE '21000'

   -- Label compound statements within handlers as HCS1, etc.
   HCS1: BEGIN
          INSERT INTO Proc_Error_Tbl
      (:SQLSTATE, CURRENT_TIMESTAMP, 'AddBranch', :hMessage);
      END HCS1;
   DECLARE CONTINUE HANDLER FOR SQLSTATE '42000'
      HCS2: BEGIN
         SET hMessage = 'Table Not Found … ';
         INSERT INTO Proc_Error_Tbl
          (:SQLSTATE, CURRENT_TIMESTAMP, 'AddBranch', :hMessage);
      END HCS2;

   -- Get next branch-id from tBranchId table

CALL GetNextBranchId (:hNextBranchId);

   -- Add new branch to tBranch table
   INSERT INTO tBranch ( BranchId, BankId, BranchName, Street,
               City, State, Zip)
   VALUES ( :hNextBranchId, :iBankId, :iBranchName, :iStreet,
               :iCity, :iState, :iZip);

  -- Assign branch number to the output parameter;
  -- the value is returned to the calling procedure

   SET oBranchId = hNextBranchId;
```

```
 -- Insert the branch number and name in tDummy table
 INSERT INTO tDummy VALUES(:hNextBranchId, :iBranchName);

 -- Insert account numbers pertaining to the current branch
SELECT max(AccountNumber) INTO :hAccountNumber FROM tAccounts;

 WHILE (hAccountNumber <= 1000) DO
    INSERT INTO tAccounts (BranchId, AccountNumber)
  VALUES ( :hNextBranchId, :hAccountNumber);
     -- Advance to next account number
    SET hAccountNumber = hAccountNumber + 1;
 END WHILE;

-- Update balance in each account of the current branch-id
 SET hAccountNumber = 1;

 L1: LOOP
     UPDATE tAccounts SET Balance = 100000
       WHERE BranchId = :hNextBranchId AND
        AccountNumber = :hAccountNumber;
     -- Generate account number
     SET hAccountNumber = hAccountNumber + 1;

     -- Check if through with all the accounts
     IF (hAccountNumber > 1000) THEN
      LEAVE L1;
     END IF;
 END LOOP L1;

 -- Update Interest for each account of the current branch-id
 FOR fAccount AS cAccount CURSOR FOR
 -- since Account is a reserved word
    SELECT Balance AS aBalance FROM tAccounts
       WHERE BranchId = :hNextBranchId
 DO

 -- Update interest for each account
    UPDATE tAccounts SET Interest = :fAccount.aBalance * 0.10
       WHERE CURRENT OF cAccount;
 END FOR;

-- Inner nested compound statement begins
Lnest: BEGIN
  -- local variable declarations in inner compound statement
  DECLARE Account_Number, counter INTEGER;
  DECLARE Acc_Balance DECIMAL (10,2);

  -- cursor declaration in inner compound statement
  DECLARE acc_cur CURSOR FOR
```

```
          SELECT AccountCode, Balance FROM tAccounts
        ORDER BY AccountNumber;

   -- condition handler declarations in inner compound statement
    DECLARE CONTINUE HANDLER FOR NOT FOUND
    HCS3: BEGIN
              DECLARE h_Message VARCHAR(50);
              DECLARE EXIT HANDLER FOR SQLWARNING
                 HCS4: BEGIN
                      SET h_Message = 'Requested sample is larger
                       than table rows';
                      INSERT INTO Proc_Error_Tbl (:SQLSTATE,
          CURRENT_TIMESTAMP, 'AddBranch', :h_Message);
                 END HCS4;

       SET h_Message = 'Data not Found ...';
             INSERT INTO Proc_Error_Tbl (:SQLSTATE,
         CURRENT_TIMESTAMP, 'AddBranch', :h_Message);
         SELECT COUNT(*) INTO :counter FROM Proc_Error_Tbl
         SAMPLE 10;
       -- Raises a warning. This is a condition raised by
       -- a handler action statement. This is handled.
        END HCS3;

        DELETE FROM tDummy1;
        -- Raises "no data found" warning
        OPEN acc_cur;
        L2: REPEAT
        BEGIN
            FETCH acc_cur INTO Account_code, Acc_Balance;
            CASE
              WHEN (Account_code  <= 1000) THEN
               INSERT INTO dummy1 (:Account_code, :Acc_Balance);
               ELSE
               LEAVE L3;
            END CASE;
        END;
        UNTIL (SQLCODE = 0)
        END REPEAT L2;
     CLOSE acc_cur;
  END Lnest; --- end of inner nested block.

END Lmain; -- This comment is part of stored procedure body
-- End-of-Create-Procedure.
```

Compiling the Procedure From an Input File

If you want to create the stored procedure *AddBranch* using the COMPILE command in BTEQ, you must submit the entire stored procedure definition in an input file.

The procedure is compiled with some warnings if any of the database objects referenced in the stored procedure body are missing or deleted when the creator is also the immediate owner of the procedure.

When the creator is not the immediate owner, compilation results in errors. The stored procedure is not created if:

- some referenced database objects are missing
- DDL, DCL or dynamic SQL statements are specified
- any database object referenced in the cursor SELECT statement is missing.

  This situation always results in an error irrespective of whether the creator is the owner of the procedure or not.

When CREATE PROCEDURE is performed from CLIv2, ODBC, or JDBC, an SPL compiler in the RDBMS server compiles the stored procedure.

Section 5:

Appendixes

Section Contents

Appendix A:

Notation Conventions

This appendix describes the notation conventions in this book.

Introduction

This book uses three conventions to describe the SQL syntax and code:

- Syntax diagrams describe SQL syntax form, including options. See "Syntax Diagram Conventions" on page A-2.
- Square braces in the text represent options. The indicated parentheses are required when you specify options.

  For example:

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

  The values for *n* and *m* are integers in all cases
- Japanese character code shorthand notation represents unprintable Japanese characters. See "Character Shorthand Notation Used In This Book" on page A-6.

This book also occasionally uses symbols from the predicate calculus to describe logical operations. See "Predicate Calculus Notation Used in This Book" on page A-8.

Syntax Diagram Conventions

Notation Conventions

The following table defines the notation used in this section:

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 entering a number with more than three digits.
Word	Variables and reserved words.

IF a word is shown in . . .	THEN it represents . . .
UPPERCASE LETTERS	a keyword. Syntax diagrams show all keywords in uppercase, unless operating system restrictions require them to be in lowercase. If a keyword is shown in uppercase, you may enter it in uppercase or mixed case.
lowercase letters	a keyword that you must enter in lowercase, such as a UNIX command.
lowercase italic letters	a variable such as a column or table name. You must substitute a proper value.
lowercase bold letters	a variable that is defined immediately following the diagram that contains it.
UNDERLINED LETTERS	the default value. This applies both to uppercase and to lowercase words.

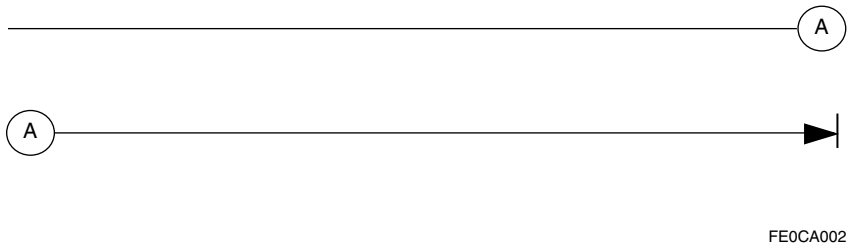
Spaces	Use one space between items, such as keywords or variables.
Punctuation	Enter all punctuation exactly as it appears in the diagram.

Paths

The main path along the syntax diagram begins at the left, 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.

Paths that are too long for one line use continuation links. Continuation links are small circles with letters indicating the beginning and end of a link:

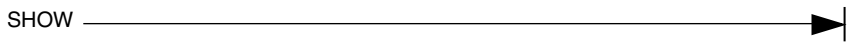


FE0CA002

When you see a circled letter in a syntax diagram, go to the corresponding circled letter and continue.

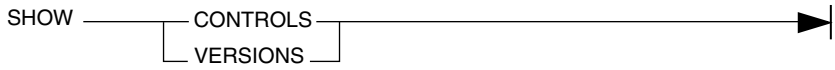
Required Items

Required items appear on the main path:



FE0CA003

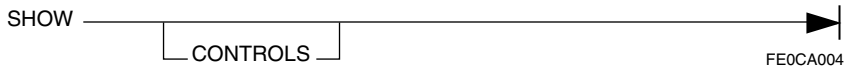
If you can choose from more than one item, the choices appear vertically, in a stack. The first item appears on the main path:



FE0CA005

Optional Items

Optional items appear below the main path:



FE0CA004

If choosing one of the items is optional, all the choices appear below the main path:



FE0CA006

You can choose one of the options, or you can disregard all of the options.

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.

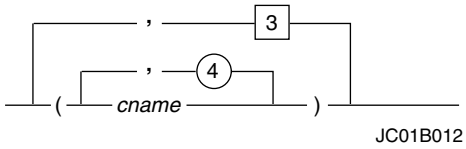


In the above syntax, the following formats are valid:

- 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.



The following rules apply to loops:

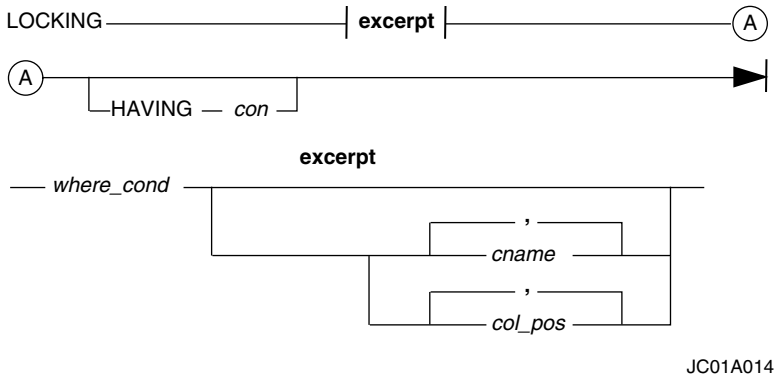
IF . . .	THEN . . .
there is a maximum number of entries allowed	the number appears in a circle on the return path. In the example, you may enter *cname* a maximum of 4 times.
there is a minimum number of entries required	the number appears in a square on the return path. In the example, you must enter at least 3 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 either side of the break. A name for the excerpted piece appears between the break marks in boldface type.

The named phrase appears immediately after the complete diagram, as illustrated by the following example.



Character Shorthand Notation Used In This Book

Introduction

This book uses the UNICODE naming convention for characters. For example, the lowercase character ‘a’ is more formally specified as either LATIN SMALL LETTER A or U+0041. The U+xxxx notation refers to a particular code point in the Unicode standard, where xxxx stands for the hexadecimal representation of the 16-bit value defined in the standard.

In parts of the book, it is convenient to use a symbol to represent a special character, or a particular class of characters. This is particularly true in discussion of the following Japanese character encodings.

- KanjiEBCDIC
- KanjiEUC
- KanjiShift-JIS

These encodings are further defined in Appendix B: “Japanese Character Sets," in *Teradata RDBMS International Character Set Support*.

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). Used to indicate transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEBCDIC	Shift In [SI] (0x0F). Used to indicate transition from multibyte to single byte KanjiEBCDIC.

Symbol	Encoding	Meaning
T	Any	Any multibyte character. Its encoding depends on the current character set. For KanjiEUC, code set 3 characters are sometimes shown 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 either a single or multibyte pad character, depending on context.
ss_2	KanjiEUC	Represents the EUC code set 2 introducer (0x8E).
ss_3	KanjiEUC	Represents the EUC code set 3 introducer (0x8F).

For example, string "TEST", where each letter is intended to be a fullwidth character, is written as **TEST**. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set

LMN<**TEST**>QRS

is represented as:

```
D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2
```

Pad Characters

The following table lists the pad characters for the various character data types.

Character Data Type	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

Predicate Calculus Notation Used in This Book

Relational databases are based on the theory of relations as developed in set theory. Predicate calculus is often the most unambiguous way to express certain relational concepts.

Occasionally this book uses the following predicate calculus notation to explain concepts.

This symbol ...	Represents this phrase ...
iff	If and only if
∀	For all
∃	There exists

Appendix B:

SQL Descriptor Area (SQLDA)

This appendix describes the SQL Descriptor Area (SQLDA).

The SQL Descriptor Area

Definition

The SQL Descriptor Area (SQLDA) is a data structure that contains a list of individual item descriptors for each of the values to be produced by a dynamically performed single row SELECT.

The application needs to have information such as the number of columns that will be in a retrieved row, their data types, size, and precision so it can know how to process values to be retrieved dynamically at run time.

ANSI Compliance

The SQL Descriptor Area is ANSI SQL-99-compliant.

SQL Statements That Use SQLDA

SQLDA is required for the DESCRIBE statement.

SQLDA is optional for the following statements:

- EXECUTE
- Dynamic FETCH
- Dynamic OPEN
- PREPARE

How SQL Statements Use SQLDA

Different SQL statements use the information in SQLDA differently.

FOR this statement …	SQLDA provides information about …
DESCRIBE	a prepared SQL statement.
PREPARE	
EXECUTE	host variables.
Dynamic FETCH	
Dynamic OPEN	

Defining the SQLDA for an Application

An application that issues dynamic SQL statements must define its own SQLDA and maintain the contents of the SQLDA fields.

You can code the SQLDA structure directly or by means of the INCLUDE SQLDA statement (see "INCLUDE SQLDA" on page 4-46).

You cannot use INCLUDE SQLDA to define SQLDA in COBOL.

SQLDA Structure

The fields of SQLDA are described in the following table.

Field Name	Format	Description
SQLDAID	CHAR(8)	Contains the characters SQLDA.
SQLDABC	INTEGER	Length of the SQLDA calculated as (16 + (44 * SQLN value)) and set by the preprocessor when a DESCRIBE statement or PREPARE statement with an INTO clause is performed. For input, the application must set this field to the size of the SQLDA structure. For output, the preprocessor sets the size necessary to contain the number of columns (SQLD) to be returned by the DESCRIBE or PREPARE INTO request.
SQLN	SHORT INTEGER	*Total* number of elements in the SQLVAR array. The application sets this field to the number of elements available for use by the preprocessor (SQLVAR). For input, SQLN must be set prior to an OPEN or EXECUTE statement. For output, SQLN must be set prior to a DESCRIBE or PREPARE INTO request. If the BOTH option of the USING clause is used, then you must specify *at least* twice the number of SQLVAR elements as columns to be returned.
SQLD	SHORT INTEGER	Number of elements in the SQLVAR array (see "SQLVAR" on page B-5) currently used to hold variable descriptions. For input, the application sets this field to the number of SQLVAR elements used for input variable information. Value must be set prior to an OPEN or EXECUTE statement. For output, the preprocessor returns the number of SQLVAR elements that the DESCRIBE or PREPARE INTO request used. If too few elements are defined to satisfy the DESCRIBE, SQLD is set to the number required and no SQLVAR elements are returned. If the BOTH option of the USING clause is used, then you must specify *at least* twice the number of SQLVAR elements as columns to be returned.

Field Name	Format	Description
SQLVAR	array	Contains a repeating second level structure that describes the columns of the result data. The structure of an SQLVAR element is as follows: • SQLTYPE (SHORT INTEGER) Contains a code indicating the data type of the column and its nullability attribute. See "SQLDA Data Type Codes" on page B-7 for a discussion of data type codes. For input, the application sets the input host variable type prior to an OPEN or EXECUTE statement. For output, the Teradata type is returned by performing a DESCRIBE statement. If the type of the host variable to receive the data differs from the value returned, the application must ensure the correct data type is placed in the field prior to executing the FETCH. • SQLLEN (SHORT INTEGER) Data length for all data types except DECIMAL. For DECIMAL, SQLLEN is overdefined as SQLPRCSN and SQLSCALE. • SQLPRCSN (BYTE INTEGER - high order byte of SQLLEN) Decimal precision (total number of digits) • SQLSCALE (BYTE INTEGER -- low order byte of SQLLEN) Decimal scale (number of digits to the right of the decimal point) For input, the application sets the input host variable length prior to an OPEN or EXECUTE statement. For output, the preprocessor returns the data length of the column. If the length of the host variable differs from the value returned, the application must ensure that the correct length is placed in the field prior to the FETCH.
SQLDATA	pointer	Indicate to the preprocessor either: • the address of the input host variable from which the value is to be taken • the output host variable where the result value is to be stored. The application must set this field prior to performing the OPEN, EXECUTE, or FETCH statements. See *Teradata Preprocessor2 Programmer Guide* for examples of how to set addresses in COBOL.

Field Name	Format	Description
SQLIND	pointer	Indicates to the preprocessor the address of the indicator variable associated with the input/output host variable pointed to by SQLDATA. The application sets this field to the address of the associated indicator variable (if any) to be used with the field whose address is in SQLDATA. This field should be set to x'00' if you do not use an indicator variable. The application must set this field prior to performing the OPEN, EXECUTE, or FETCH statements.
SQLNAME	VARCHAR (30)	Contains either: • the column name • the column title For input this field has no meaning as an input host variable. For output, the preprocessor sets this field based on information in the USING clause. This field is used only by the application and has no further meaning to the preprocessor.

SQLDA Data Type Codes

Introduction

This topic lists the data type codes used by the Teradata database and the embedded SQL preprocessor.

Teradata returns these values to the SQLDA specified by the application with a PREPARE or DESCRIBE statement.

The preprocessor uses these values for both input and output SQLDA fields generated by the precompiler and recognized at execution.

Data Type Codes Stored in SQLTYPE

The data type is contained in the two byte SQLTYPE integer field of the SQLDA.

How to Interpret SQL Data Type Codes

Use these guidelines in interpreting the tables in "SQL Data Type Codes" on page B-7 and "Unused and Internally Used SQL Data Type Codes" on page B-8.

IF the code is ...	THEN the variable is ...
even-numbered	not nullable (no indicator variables are allowed).
odd-numbered	nullable (indicator variables are allowed).

SQL Data Type Codes

This table lists the SQL data types and their SQLDA codes:

Data Type	Code
BYTE	692 693
VARBYTE	688 689
LONG VARBYTE	696 697
• CHARACTER • DATE (ANSI) • TIME • TIMESTAMP • INTERVAL	452 453

Data Type	Code
VARCHARACTER	448 449
LONG VARCHARACTER	456 457
GRAPHIC	468 469
VARGRAPHIC	464 465
LONG VARGRAPHIC	472 473
DATE (Teradata)	752 753
DECIMAL	484 485
FLOAT	480 481
BYTEINT	756 757
SMALLINT	500 501
INTEGER	496 497

Unused and Internally Used SQL Data Type Codes

This table lists SQLDA data type codes that are either not used or are used internally so are not user-visible.

Data Type	Codes
CHARACTER (array)	460 461
IBM DATE	384 385
INTEGER (byte flipped)	498 499
SMALLINT (byte flipped)	502 503

Data Type	Codes
ZONED DECIMAL (sign trailing)	432 433
ZONED DECIMAL (sign trailing separate)	436 437
ZONED DECIMAL (sign leading)	440 441
ZONED DECIMAL (sign leading separate)	444 445

Appendix C:

SQL Communications Area (SQLCA)

This appendix describes the SQL Communications Area (SQLCA).

The SQL Communications Area

Introduction

The embedded SQL preprocessor can return program status and error information to applications in several possible ways.

The primary communication method for applications written to run in Teradata transaction mode has been the SQLCA structure. SQLCA is a data structure that contains a list of return codes and other status information about a completed DML statement.

SQLCA provides the following support to embedded SQL applications:

- Result reporting
- Warning condition reporting
- Support for DSNTIAR

Embedded SQL application programs can interrogate the fields of SQLCA for the return codes that indicate the results of having performed an executable embedded SQL statement.

Embedded SQL applications can also retrieve full diagnostic text by using a supplied routine (see "PPRTEXT" on page C-5).

SQLCA cannot be used with stored procedures.

ANSI Compliance

The SQL Communications Area and the INCLUDE SQLCA statement are not ANSI-compliant. When the preprocessor TRANSACT or -tr option is set to ANSI, it flags INCLUDE SQLCA as an error.

ANSI mode applications report program status and errors by means of the SQLSTATE and SQLCODE[1] status variables (see Appendix D: "SQLSTATE Mappings" for information about mapping SQLCODE and SQLSTATE variables to one another and to Teradata error messages).

1 The ANSI SQL-92 standard explicitly deprecates SQLCODE. The ANSI SQL-99 standard does not even mention SQLCODE.

You should always use SQLSTATE when writing stored procedures and embedded SQL applications to run in ANSI transaction mode. You can also use SQLSTATE in your stored procedures and embedded SQL applications written to run in Teradata transaction mode. To ensure maximum portability, you should always use SQLSTATE, not SQLCODE, to monitor application status.

When you are developing stored procedures or embedded SQL applications in the ANSI environment, use the status variable SQLSTATE in place of SQLCA and define it explicitly in your code. See "SQLSTATE" on page 8-2, "SQLCODE" on page 8-6 and "SQLSTATE Mappings" on page D-1 for further information.

Defining the SQLCA for an Application

An application program typically defines the SQLCA using an INCLUDE SQLCA statement (see "INCLUDE SQLCA" on page 4-44).

Because this data structure is read-only, an application program should never attempt to write values into the SQLCA.

Checking Status Variables

Include a test of SQLCODE (or SQLSTATE if you are not using SQLCA) after each performance of an executable embedded SQL or stored procedure statement to ensure that the statement completes successfully. You also should always write application code[2] to handle unacceptable status variable values.

2 Or use appropriate condition handlers if the application is a stored procedure.

Result Reporting

Introduction

The results of SQL requests sent from an embedded SQL application are reported in the SQLCODE field of the SQLCA structure if the application is written to check SQLCODE values.

If the application is a stored procedure written to use SQLCODE, then status is reported to the SQLCODE status variable.

What Various Categories of SQLCODE Mean

The following table explains the general meanings of the three basic SQLCODE categories:

WHEN SQLCODE is . . .	THEN . . .
negative	an error occurred during processing. the application can determine the source of the error using the SQLCODE in conjunction with the first SQLERRD element. SQLERRD shows the following: • Error conditions detected by the precompiler execution environment • Error conditions reported by CLI/TDP/Teradata The first SQLERRD element is zero when the error is detected directly by the preprocessor. These items are listed below: • A list of error codes • The text associated with the code • A possible explanation • A possible solution for these errors When an error condition is detected by CLI, TDP, or the Teradata database, the SQLCODE is set to the negative of the sum of the error code plus 10000. The application can look at the SQLCODE without interrogating the first SQLERRD element.

WHEN SQLCODE is . . .	THEN . . .
negative (cont.)	Teradata error conditions are reported in the following distinct styles: **1** Teradata codes that have similar or equivalent DB2 values are mapped to the DB2 value in the SQLCODE field. See "Retryable Errors" on page C-6 **2** Teradata codes that have no similar or equivalent code have the SQLCODE value set to the negative of the Teradata code. In either case, the first SQLERRD element contains the actual value returned by Teradata. Use caution in distinguishing between a precompiler execution time error and a Teradata error because some SQLCODE values are found in both categories of error.
zero	processing was successful, though there might be warnings.
positive	termination was normal. Positive values other than 0 or +100 indicate Teradata-issued warnings, such as the end-of-data reached for a request.

PPRTEXT

Applications can obtain additional diagnostic help for a non-zero SQLCODE by invoking PPRTEXT.

PPRTEXT returns the error code (normally the same value as in the first SQLERRD element) and the textual message associated with the condition.

The following four parameters are required to execute PPRTEXT:

- The address of the runtime context area, SQL-RDTRTCON for COBOL and SQL_RDTRTCON for C and PL/I.
- A 4-byte integer field to contain the actual error code.
- A varying character field up to 255 characters long to contain the error text.
- A 2-byte integer field which contains the maximum size of the error text to be returned.

You can find examples of PPRTEXT usage in *Teradata Preprocessor2 Programmer Guide*.

Warning Condition Reporting

Warning conditions are reported to the application by means of the SQLWARNn fields in SQLCA.

SQLWARN0 contains the value W if any warning condition is detected during execution.

The warning fields are used to signal such conditions as implicit rollbacks and data truncation.

See *Teradata Preprocessor2 Programmer Guide* for details of the exact values and conditions of each warning field.

Retryable Errors

Error codes for retryable events also are indicated via the SQLWARNn fields.

SQLWARN6 is set to R when such an error is detected. The application can then take the appropriate action.

The following bullets list the retryable error codes:

- 2631
- 2639
- 2641
- 2659
- 2825
- 2826
- 2827
- 2828
- 2834
- 2835
- 3111
- 3120
- 3598
- 3603
- 3897

These codes are found in the first SQLERRD field of SQLCA.

Support for DSNTIAR

Whenever possible, the SQLERRM field of SQLCA contains message inserts usable by the IBM-supplied routine DSNTIAR. Details of the SQLERRM field are found in *Teradata Preprocessor2 Programmer Guide.*

Refer to the IBM documentation for information about DSNTIAR.

SQLCA Fields

SQLCA is used with embedded SQL applications only. Stored procedures using SQLCODE to report status declare an SQLCODE variable and interrogate it to determine statement status.

The SQLCA fields are described in the following table:

<table>
<tr><th>Field Name</th><th>Format</th><th>Description</th></tr>
<tr><td>SQLCAID</td><td>CHARACTER(8)</td><td>Contains the characters ‘SQLCA. ’</td></tr>
<tr><td>SQLCABC</td><td>INTEGER</td><td>Length of the SQLCA (136 (x’88’)).</td></tr>
<tr><td>SQLCODE</td><td>INTEGER</td><td>Primary indicator of the result of SQL statement execution.
<table>
<tr><th>IF the value of SQLCODE is ...</th><th>THEN ...</th></tr>
<tr><td>0</td><td>the statement performed normally.</td></tr>
<tr><td>positive</td><td>a non-error exception occurred.
Examples are no more data was found or various non-fatal warnings.</td></tr>
<tr><td>negative</td><td>the statement failed because of an error condition.</td></tr>
</table></td></tr>
<tr><td></td><td></td><td>The possible values for SQLCODE and their definitions are detailed in Teradata RDBMS Messages.</td></tr>
</table>

Field Name	Format	Description
SQLERRM	VARCHAR(70)	Contains the error message inserts for the SQLCODE associated with variable information. The SQLERRM field inserts are presented to the application as a single character string. The length of the string is provided, but the lengths of the individual inserts is not. The string begins with a 16-bit word that contains the length of the remaining data. The data consists of as many as 70 characters of insert text, with the character X'FF' serving as a separator between inserts. If the inserts and separator characters are greater than 70 characters, then the array is truncated at the right. As an example, with a SQLCODE of -552, which is an access right violation, SQLERRM contains the following three or four inserts: • The user name for the user who does not have the required privilege • The name of the privilege that was unavailable • The name of the database on which the privilege was required • The name of the table, view or macro or stored procedure on which the privilege was required, unless it was a database level privilege
SQLERRP	CHARACTER(8)	Contains the name of the preprocessor module that detected the error
SQLERRD	6-word array	Contains miscellaneous information stored in an array. Because array addressing nomenclature differs among C, COBOL, and PL/I, the following description of the six SQLERRD words is not numbered. In order, the six words are the following: • The CLIv2, TDP or Teradata error code. • Reserved for future use. • The number of rows processed, where applicable. This field is generally referred to as the Activity Count. As an example, the number of rows selected upon OPEN of a selection cursor is returned to the application in this word. • The relative cost estimate. Its value, as returned by Teradata, is set by the PREPARE statement and can be used to compare the estimated cost of different dynamic SQL statements, in CPU cycles, and so forth. • Reserved for future use. • Reserved for future use.

Field Name	Format	Description
SQLWARN	CHARACTER(11) array	Indicates the presence of warning conditions. Except for SQLWARN6, each character takes the value pad character or 'W'. The 11 characters of SQLWARN are defined as follows: • SQLWARN0 indicates whether any of the remaining ten warnings have been set. W indicates that one or more of the other ten characters contains a 'W' or that SQLWARN6 contains an 'R'. A pad character indicates that the remaining ten characters are also pad characters. • SQLWARN1 W indicates that one or more output character values or byte strings were truncated because the host variables designated to receive them were too small. If this condition occurs, the indicator variables for the truncated values contain the lengths before truncation. A pad character indicates no truncation. • SQLWARN2 W indicates that a warning has been issued by Teradata. SQLCODE contains the warning code. A pad character indicates no warning.

Field Name	Format	Description
SQLWARN (continued)		• SQLWARN3 W indicates that the number of columns returned by a SELECT was not equal to the number of host variables in the INTO clause. The number of variables actually returned to the application program is the lesser of these two values. A pad character indicates a match. • SQLWARN4 Reserved for future use. • SQLWARN5 Reserved for future use. • SQLWARN6 W indicates that the statement caused Teradata SQL implicitly to roll back a unit of work, for example, because deadlock was detected. R indicates that a retryable error has occurred A pad character indicates no rollback or error. • SQLWARN7 Reserved for future use. • SQLWARN8 Reserved for future use. • SQLWARN9 Reserved for future use. • SQLWARNA Reserved for future use.
SQLEXT	CHARACTER(5)	Contains the SQLSTATE value associated with the SQLCODE.

Mapping SQLCODE Values to Teradata Error Message Numbers

The following table lists Teradata error messages mapped to their corresponding SQLCODE values. The information in this table is not useful if you are running an embedded SQL or stored procedure application in ANSI transaction mode using SQLSTATE to communicate program status.

The table is ordered by Teradata error message number within SQLCODE. For those Teradata error messages that do not map directly to an SQLCODE, the SQLCODE is set to the negative of the Teradata error message number. For example, Teradata error message 3868 is set to SQLCODE -3868.

SQLCODE	Teradata Error Message	SQLCODE	Teradata Error Message	SQLCODE	Teradata Error Message	SQLCODE	Teradata Error Message
+000	2938	+901 (cont.)	5817	+901 (cont.)	5838	-101 (cont.)	3714
	3110		5818		5839		3741
	3513		5819		5840		3850
	3514		5820		5841		3851
+901	5800		5821	-010	3760		3867
	5801		5822	-060	3527		3896
	5802		5823		3528	-102	3738
	5803		5824		3529	-103	3751
	5804		5825		3530		3759
	5805		5826		3617	-107	3737
	5806		5827	-101	2664	-110	3704
	5807		5828		3509		3775
	5808		5829		3540	-112	3568
	5809		5830		3597		3627
	5810		5831		3609		3628
	5811		5832		3629	-117	3812
	5812		5833		3702		3813
	5813		5834		3705	-119	3554
	5814		5835		3710	-120	3569
	5815		5836		3712		3574
	5816		5837		3714		3872

SQLCODE	Teradata Error Message	SQLCODE	Teradata Error Message	SQLCODE	Teradata Error Message	SQLCODE	Teradata Error Message
-121	3606	-171 (cont.)	3857	-408	3641	-802 (cont.)	2162
	3885	-172	3732		3642		2163
-122	3504	-182	2665	-415	3654		2164
	3883		2666	-421	3653		2165
-125	3637	-203	3809	-551	3523		2166
-128	3731		3822		3524		2232
-138	2662	-204	3526		3856		2233
	2663		3537		3865		2239
-150	3823		3538		3866		2240
-151	3659		3539		3880		2614
-170	3816		3656		3881		2615
	3817		3802	-552	3545		2617
	3818		3807	-601	3519		2618
	3820		3824		3534		2619
	3821	-205	3810		3744		2620
-171	2603	-207	3848		3801		2621
	2604	-401	2147		3803		2661
	2605		2149		3804		2674
	2606		3639		3805		2675
	2607		3640	-602	3518		2676
	2608	-402	3622	-612	3515		2677
	2622		3643		3517		2678
	2623		3644		3560		2679
	3580	-404	3520	-637	3733		2682
	3581		3564		3789		2683
	3647	-405	3752		3889		2684
	3660		3753	-680	3556		2685
	3662	-407	2689		3582		2686
	3663		3604		3919		2687

SQLCODE	Teradata Error Message	SQLCODE	Teradata Error Message	SQLCODE	Teradata Error Message	SQLCODE	Teradata Error Message
-171 (cont.)	3819		3811	-802	2161		3532
-802 (cont.)	3535	-811	3669	-905 (cont.)	3577	-922 (cont.)	3015
	3754	-901	2827		3638		3016
	3755		2828		3661		3023
	3756		3120	-911	2450	-923	3006
	3757		3897		2631		3007
	3758	-905	2805	-913	3111	-925	3827
-803	2801		2843	-922	3002		3829
	2802		3130		3003		3833
	2803		3541		3004		
	2980		3543		3014		

Mapping SQLCODE Values to SQLSTATE Values

SQLCODE Rules

SQLCODE is not defined by the ANSI SQL-99 standard. Its use was deprecated by the ANSI SQL-92 standard and abandoned by the SQL-99 standard.

The following rules apply to SQLCODE status variables:

- For precompiler validation purposes, SQLCODE must be defined as a 32-bit signed INTEGER value for embedded SQL applications.
- SQLCODE and SQLSTATE can both be specified in the same compilation unit. Both status variables subsequently contain valid status variable codes.

SQLSTATE Rules

SQLSTATE is defined by the ANSI SQL-99 standard as a 5-character string value. The value is logically divided into a 2-character class and a 3-character subclass.

The following rules apply to SQLSTATE status variables:

- Status code values can be integers or a mix of integers with Latin uppercase characters.
- Unless otherwise specified, CLI/TDP and Teradata error messages always map into SQLSTATE values.
- Unmapped CLI/TDP errors have a class of T0 and a subclass containing a 3-digit CLI error code.

  For example, a CLI error of 157 (invalid Use_Presence_Bits option) produces a class of T0 and a subclass of 157.
- Unmapped Teradata errors have classes of T1 through T9, with the digit in the class corresponding to the first digit of the Teradata error code.

  The subclass contains the remaining 3 digits of the Teradata error code.

  For example: A Teradata error code of 3776 (unterminated comment) maps to a class of T3 and a subclass of 776.
- For precompiler validation purposes, SQLSTATE must be defined as a fixed-length CHAR(5) array.

  For C language programs, SQLSTATE must be defined as CHAR(6) to accommodate the C null terminator.
- SQLCODE and SQLSTATE can both be specified in the same compilation unit. Both status variables subsequently contain valid result codes.

Mapping SQLCODE Values to SQLSTATE Values

The following table maps SQLCODE values to SQLSTATE values for those SQLCODE values that are not generated as the result of a CLI, TDP, or Teradata SQL error:

SQLCODE	SQLSTATE		Notes	SQLCODE	SQLSTATE		Notes
	Class	Subclass			Class	Subclass	
100	02	000		-742	08	000	
901	01	901		-743	08	000	
902	01	004		-744	08	000	
-101	54	001		-752	08	752	
-104	42	512	1, 2	-804	T0	804	
-302	22	024		-811	21	000	
-303	22	509		-822	51	004	
-304	22	003		-840	21	840	
-305	22	002		-901	T0	T10	
-413	22	003		-925	56	021	
-501	24	501		-926	56	021	
-502	24	502		-942	T0	T12	
-504	52	008		-943	24	000	
-508	**24**	**508**		-1001	T0	T13	
-510	**53**	**028**		-1002	T0	T14	
-514	24	000		-1003	T0	T15	
-515	07	515		-1005	T0	T16	
-563	08	003		-1006	07	T17	
-650	04	000		-1007	22	007	
-651	03	000		-1009	22	T04	
-652	04	000		-1010	T0	T18	
-653	41	000		-1013	22	023	

SQLCODE	SQLSTATE		Notes	SQLCODE	SQLSTATE		Notes
	Class	Subclass			Class	Subclass	
-740	08	003					
-741	08	002					

Note	Description
1	This usage is not consistent with IBM DB2. DB2 uses class 37, which applies only to dynamic SQL. Code 2A applies to static SQL.
2	This code should be -84, not -104.

Appendix D:

SQLSTATE Mappings

This appendix provides the complete set of SQLSTATE mappings for embedded SQL and stored procedure applications.

SQLSTATE Codes

Definition

SQLSTATE codes are status values in embedded SQL programs and stored procedures that reflect the status of an SQL statement execution.

Unlike SQLCODE codes, which are integer values, SQLSTATE codes are character strings. Because of this, they are always displayed between APOSTROPHE characters like the following dummy SQLSTATE value: 'xxxxx'.

The characters of an SQLSTATE code are divided logically into two categories:

- A 2-character class value.

  The first two characters of the SQLSTATE code are any one of the ANSI SQL-99-defined SQLSTATE classes (see "SQLSTATE Class Definitions" on page D-2).
- A 3-character subclass value.

  Subclass values can be any numeric or simple uppercase LATIN character string.

SQLSTATE Code Values

Successful completion of an SQL request with warning code = 0 returns the SQLSTATE code value '00000'.

For all other situations, see "Mapping Teradata Error Messages to SQLSTATE Values" on page D-5.

SQLSTATE Class Definitions

ANSI defines the SQLSTATE class definitions provided in the following table. Teradata does not support all the classes listed.

Class Code	Definition
00	Successful completion
01	Warning
02	No data found
03	SQL statement not yet complete
07	Dynamic SQL error
08	Connection exception
09	Triggered action exception
0A	Unsupported feature

Class Code	Definition
0B	Non-valid transaction initiation
0D	Non-valid target type specification
0E	Non-valid schema name list specification
0F	Locator exception
0K	Resignal when handler not active
0L	Non-valid grammar
0M	Non-valid SQL-invoked procedure reference
0N	SQL/XML mapping error
0P	Non-valid role specification
0S	Non-valid transform group name specification
0T	Target table disagrees with cursor specification
0U	Attempt to assign to nonupdatable column
0V	Attempt to assign to ordering column
0W	Prohibited statement encountered during trigger performance
0X	Non-valid foreign server specification
0Y	Pass-through specific condition
20	Case not found for case statement
21	Cardinality violation
22	Data exception
23	Constraint violation
24	Non-valid cursor state
25	Non-valid transaction state
26	Non-valid statement name
27	Triggered data change violation
28	Non-valid authorization ID specification[a]
2B	Dependent privilege descriptors still exist
2C	Non-valid character set name
2D	Non-valid transaction termination
2E	Non-valid connection name
2F	SQL routine exception
30	Non-valid SQL statement
31	Non-valid target specification value
33	Non-valid SQLDA name

Class Code	Definition
34	Non-valid cursor name
35	Non-valid condition number
36	Cursor sensitivity exception
38	External routine exception
39	External routine invocation exception
3B	Savepoint exception
3C	Ambiguous cursor name
3D	Non-valid catalog name[b]
3F	Non-valid schema name[c]
40	Transaction rollback
42	Syntax error or access rule violation
44	With check option violation
45	Unhandled user-defined exception
46	Java DDL or Java execution
HV	Foreign data wrapper-specific condition
HW	Datalink exception
HY	Call-Level Interface condition[d]
HZ	Remote database access condition

a. Teradata SQL does not directly support the ANSI concept of an authorization ID. The ANSI authorization ID is essentially a Teradata user.
b. Teradata SQL does not directly support the ANSI concept of a catalog. The ANSI catalog is essentially the same thing as the Teradata data dictionary.
c. Teradata SQL does not directly support the ANSI concept of a schema. The ANSI schema is essentially the same thing as the Teradata constructs User and Database.
d. The Call-Level Interface referred to is not the Teradata CLIv2, but rather an ANSI-standard CLI that is a dialect of the Microsoft Open Database Connectivity, or ODBC, specification.

Mapping Teradata Error Messages to SQLSTATE Values

The following table lists all cases of Teradata return codes mapped to their corresponding SQLSTATE code, except for successful completion of an SQL request with warning code = 0, which returns the SQLSTATE code '00000'.

For any return codes not listed in this table, Teradata sets SQLSTATE to a character string in the format of the literal character T (LATIN CAPITAL LETTER T) followed by the 4-digit return code in the Success, Failure or Error parcels.

Teradata Error Message	SQLSTATE Code		Teradata Error Message	SQLSTATE Code	
	Class Value	Subclass Value		Class Value	Subclass Value
2147	53	018	2149	53	018
2161	22	012	2162	22	012
2163	22	012	2164	22	003
2165	22	003	2166	22	003
2232	22	003	2233	22	003
2239	22	003	2240	22	003
2450	40	001	2603	53	015
2604	53	015	2605	53	015
2606	53	015	2607	53	015
2608	53	015	2614	22	003
2615	22	003	2616	22	003
2617	22	003	2618	22	012
2619	22	012	2620	22	021
2621	22	012	2622	53	015
2623	53	015	2631	40	001
2661	22	000	2662	22	011
2663	22	011	2664	54	001
2665	22	007	2666	22	007
2674	22	003	2675	22	003

Teradata Error Message	SQLSTATE Code		Teradata Error Message	SQLSTATE Code	
	Class Value	Subclass Value		Class Value	Subclass Value
2676	22	012	2679	22	012
2680	22	023	2682	22	003
2683	22	003	2684	22	012
2687	22	012	2689	23	502
2700	23	700	2726	23	726
2727	23	727	2728	23	728
2801	23	505	2802	23	505
2803	23	505	2805	57	014
2827	58	004	2828	58	004
2843	57	014	2892	01	003
2893	22	001	2894	24	894
2895	24	895	2896	24	896
2938	00	000	2980	23	505
3002	46	000	3003	46	000
3004	46	000	3006	08	T06
3007	08	T07	3014	46	000
3015	46	000	3016	46	000
3023	46	000	3025	08	T25
3026	46	000	3110	00	000
3111	40	502	3120	58	004
3121	02	000	3130	57	014
3504	53	003	3509	54	001
3513	00	000	3514	00	000
3515	52	011	3517	52	011
3518	54	008	3519	52	010
3520	22	003	3523	42	000
3524	42	000	3526	52	004
3527	42	015	3528	22	012

Teradata Error Message	SQLSTATE Code		Teradata Error Message	SQLSTATE Code	
	Class Value	Subclass Value		Class Value	Subclass Value
3529	42	015	3530	42	015
3532	53	021	3534	52	010
3535	22	018	3539	52	004
3540	54	001	3541	57	014
3543	57	014	3545	42	502
3554	53	003	3556	54	011
3560	52	011	3564	44	000
3568	42	507	3569	56	003
3574	56	003	3577	57	014
3580	53	015	3581	53	015
3582	42	582	3597	54	001
3604	23	502	3606	52	001
3609	54	001	3617	42	015
3622	53	019	3627	42	507
3628	42	507	3629	54	001
3637	53	005	3638	57	014
3639	53	018	3640	53	018
3641	53	021	3642	53	021
3643	53	019	3644	53	019
3653	53	026	3654	53	025
3656	52	004	3659	53	008
3660	53	015	3661	57	014
3662	53	015	3663	53	015
3669	21	000	3702	54	001
3704	42	506	3705	54	001
3710	54	001	3712	54	001
3714	54	001	3721	24	721
3731	42	501	3732	0A	732

Teradata Error Message	SQLSTATE Code		Teradata Error Message	SQLSTATE Code	
	Class Value	Subclass Value		Class Value	Subclass Value
3733	42	514	3735	01	004
3737	01	004	3738	01	004
3741	54	001	3744	52	010
3747	01	003	3751	42	504
3752	22	003	3753	22	003
3754	22	003	3755	22	003
3756	22	012	3757	22	003
3758	22	003	3759	42	504
3760	42	503	3775	42	506
3789	42	514	3801	52	010
3802	52	004	3803	52	010
3804	52	010	3805	52	010
3807	42	000	3809	52	002
3810	52	003	3811	23	502
3812	42	000	3813	42	000
3816	42	505	3817	42	505
3818	42	505	3819	53	015
3820	42	505	3821	42	505
3822	52	002	3823	53	007
3824	52	004	3827	56	021
3829	56	021	3833	56	021
3848	52	006	3850	54	001
3851	54	001	3856	42	501
3857	53	015	3858	42	501
3865	42	501	3866	42	501
3867	54	001	3872	56	003
3880	42	501	3881	42	501
3883	53	003	3885	52	001

Teradata Error Message	SQLSTATE Code		Teradata Error Message	SQLSTATE Code	
	Class Value	Subclass Value		Class Value	Subclass Value
3889	42	514	3896	54	001
3897	58	004	3919	54	011
3968	42	968	3969	42	969
3970	42	970	3971	42	971
3973	42	973	3974	42	974
3975	42	975	3976	53	030
3977	42	977	3978	42	978
3979	42	979	3980	42	980
3981	42	981	3982	42	982
3989	01	001	3990	42	990
3991	42	991	3992	42	992
3993	42	993	3994	42	994
3995	42	995	3996	22	001
3997	22	019	3998	22	025
3999	01	999	5300	42	J00
5301	42	J01	5302	52	009
5303	42	J03	5304	42	J04
5305	42	J05	5306	42	J06
5307	42	J07	5308	42	J08
5309	42	J09	5310	42	J10
5311	42	J11	5312	42	J12
5313	42	J13	5314	42	J14
5315	42	J15	5316	44	000
5317	23	000	5800	01	800
5801	01	801	5802	01	802
5803	01	803	5804	01	804
5805	01	805	5806	01	806
5807	01	807	5808	01	808

Teradata Error Message	SQLSTATE Code		Teradata Error Message	SQLSTATE Code	
	Class Value	Subclass Value		Class Value	Subclass Value
5809	01	809	5810	01	810
5811	01	811	5812	01	812
5813	01	813	5814	01	814
5815	01	815	5816	01	816
5817	01	817	5818	01	818
5819	01	819	5820	01	820
5821	01	821	5822	01	822
5823	01	823	5824	01	824
5825	01	825	5826	01	826
5827	01	827	5828	01	828
5829	01	829	5830	01	830
5831	01	831	5832	01	832
5833	01	833	5834	01	834
5835	01	835	5836	01	836
5837	01	837	5838	01	838
5839	01	839	5840	01	840
5841	01	841	7601	20	000
7610	24	502	7627	21	000
7631	24	501	7632	02	000

Index

Numerics

2PC, with EXPLAIN 3–58

A

ABORT statement 3–3
  correlated subqueries 3–4
  WHERE clause 3–4
ABORT statement, correlated subqueries and 1–56
ALL, SELECT option 1–3, 1–20
ASC, SELECT option 1–6
ASYNC statement modifier 6–6

B

BEGIN - END statement 9–40
BEGIN DECLARE SECTION statement 4–29
BEGIN TRANSACTION statement 3–7
Bit mapping, EXPLAIN modifier and 3–56
BY, SELECT option 1–7

C

CALL statement 3–13
Cartesian product 2–8
CASE statement 9–46
  searched CASE 9–49
  simple CASE 9–49
CHECKPOINT statement 3–28
CHECKSUM locking 7–11
CLI, error codes C–4
CLOSE statement 7–15
COMMENT statement (placing form) 3–31
COMMENT statement (returning form) 4–30
COMMIT statement 3–33
CONNECT statement, SQL 5–5
Connection, Teradata DBS from PP2 5–3
  explicit connection 5–3
  implicit connection 5–4
  runtime execution connection 5–2
CONTINUE handler type. See DECLARE HANDLER statement
Control statements in stored procedures. See Stored procedure control statements
Correlated subqueries
  ABORT/ROLLBACK statements and 1–56
  defined 1–41
  DELETE statement and 1–54, 3–38
  equality/ inequality constraints in 1–48
  EXISTS predicate and 1–50
  noncorrelated subqueries compared 1–46
  outer references, behavior of 1–42
  rules for 1–44
  SELECT statement and 1–49
  UPDATE statement and 1–52, 3–155
Cross join 2–8
CROSS JOIN, SELECT option 1–5
Cursors 4–24, 7–2
  CHECKSUM locking 7–11
  DELETE statement and cursors, restrictions on usage 7–12
  function and purpose 7–6
  positioned 7–9
  SELECT statement and cursors 7–2
  updatable 7–9
  updatable, functions of 7–2

D

Data type codes B–7
DATABASE statement 4–32
DECLARE CURSOR statement 7–17
DECLARE CURSOR statement (dynamic SQL form) 7–19
DECLARE CURSOR statement (macro form) 7–21
DECLARE CURSOR statement (request form) 7–23
DECLARE CURSOR statement (selection form) 7–26
DECLARE CURSOR statement (stored procedures form) 7–29
DECLARE HANDLER statement 9–58
  CONTINUE type 9–60, 9–80
  control statement handling 9–105
  EXIT type 9–60, 9–85
  NOT FOUND form 9–60, 9–101
  SQLEXCEPTION form 9–60, 9–92
  SQLWARNING form 9–60, 9–97
DECLARE statement 9–55
DECLARE STATEMENT statement 4–34
DECLARE TABLE statement 4–35

DELETE statement 3–36
  correlated subqueries 3–36
DELETE statement (basic/searched form) 3–42
DELETE statement (join condition form) 3–44
DELETE statement (positioned form) 3–46
DELETE statement, correlated subqueries and 1–54
Derived tables 1–29, 2–26
DESC, SELECT option 1–6
DESCRIBE statement 4–24, 4–55
DISTINCT, SELECT option 1–3, 1–20
DSNTIAR C–6
Dynamic SQL 4–52
  DECLARE CURSOR (Dynamic SQL Form) statement 7–19
  DESCRIBE statement 4–55
  EXECUTE (Dynamic SQL Form) statement 4–59
  EXECUTE IMMEDIATE statement 4–61
  immediate form 4–52
  PREPARE statement 4–63
  prepared form 4–52
  statement syntax 4–54
Dynamic SQL in stored procedures. See Stored procedures, dynamic SQL

E

ECHO statement 3–48
Embedded SQL
  client languages supported 4–3
  DECLARE CURSOR (Dynamic SQL Form) statement 7–19
  DESCRIBE statement 4–55
  dynamic SQL 4–52
  dynamic SQL statement syntax 4–54
  EXECUTE (Dynamic SQL Form) statement 4–59
  EXECUTE IMMEDIATE statement 4–61
  host variables 4–7
  immediate dynamic SQL 4–52
  indicator variables 4–20
  input host variables 4–12
  multistatement requests 4–24
  null handling 4–20
  PREPARE statement 4–63
  prepared dynamic SQL 4–52
  preprocessor 4–3
  rules and characteristics 4–4
  SQL character string variables 4–19
Embedded SQL, rules for 1–7, 1–17
END DECLARE SECTION statement 4–37
END TRANSACTION statement 3–50
END-EXEC statement terminator 4–38
EXEC SQL statement prefix 4–40
EXEC statement 4–39
EXECUTE IMMEDIATE statement 4–61
EXECUTE statement (dynamic SQL form) 4–59
EXECUTE statement (macro form) 3–52
EXISTS predicate, WHERE clause and 1–34
EXIT handler type. See DECLARE HANDLER statement
EXPLAIN modifier 3–55
  explanation of output 3–59
  hash joins 3–73
  indexed access 3–77
  join processing 3–71
  MERGE conditional steps 3–90
  parallel steps 3–80
  partitioned primary index 3–82
  time estimates 3–56
  UPDATE (Upsert Form) 3–97

F

FETCH statement (Embedded SQL Form) 7–33
FETCH statement (Stored Procedures Form) 7–37
FOR statement 9–111
FROM clause 1–22
FROM, SELECT option 1–3
Full outer join 2–24
FULL OUTER, SELECT option 1–4

G

GET CRASH statement, SQL 5–8
GROUP BY
  Effect of sort key and row size on ability to perform 1–58
GROUP BY clause 1–57, 3–104
GROUP BY, SELECT option 1–5

H

HAVING clause 1–61, 3–105
HAVING, SELECT option 1–5
Host structures 4–5

Host variables 4–7
general rules for usage 4–8
application storage areas 4–9
host structures 4–5
indicator variables 4–20
input host variables 4–12
dynamic request input host variables 4–13
rules for use 4–12
static request input host variables 4–13
output host variables 4–15
assignment rules 4–16
valid data type combinations 4–15
SQL character strings 4–19
structured host variables 4–5

I

IF statement 9–118
INCLUDE SQLCA statement 4–44
INCLUDE SQLDA statement 4–46
INCLUDE statement 4–42
Indexed access
EXPLAIN modifier 3–77
Indicator variables 4–20
rules and guidelines 4–20
rules for use 4–20
Inner join
cross join 2–8
ordinary inner join 2–5
self-join 2–10
INNER, SELECT option 1–4
Input host variables 4–12
INSERT statement 3–106
ITERATE statement 9–125

J

Japanese character code notation, how to read A–6
Join
derived table and 2–26
multitable join 2–26
unconstrained 1–34
Join processing
EXPLAIN modifier 3–71
JOIN, SELECT option 1–4
Join. See Inner join, Outer join, Multitable join

L

LEAVE statement 9–129
Left outer join 2–20
LEFT OUTER, SELECT option 1–4
LOCKING modifier 3–119
LOGOFF statement, SQL 5–9
LOGON statement, SQL 5–11
LOOP statement 9–132

M

MERGE
EXPLAIN statement for conditional steps 3–90
MERGE statement 3–132
Multiple session programming 6–2
Multistatement requests 4–24
DESCRIBE statement 4–24
dynamic SQL 4–24
FOR STATEMENT clause 4–24
PREPARE statement 4–24
Multitable join 2–26
ON clause evaluation order 2–26
spool file and 2–26

N

Noncorrelated subqueries
correlated subqueries compared 1–46
DELETE statement and 3–38
rules for 1–45
UPDATE statement and 3–155
NOT FOUND handler form. See DECLARE HANDLER statement
Null handling in embedded SQL 4–20
Null, sorting and 1–78

O

ON, SELECT option 1–5
OPEN statement (Embedded SQL Form) 7–42
OPEN statement (Stored Procedures Form) 7–45
ORDER BY
Effect of sort key and row size on ability to perform 1–78
ORDER BY clause 1–77, 3–139
ORDER BY, SELECT option 1–6

Ordinary inner join 2–5
Outer join
  case study 2–38
  coding ON clauses for 2–30
  coding ON clauses with WHERE clauses for 2–33
  components of 2–13
  defined 2–12
  full outer join 2–24
  left outer join 2–20
  nulls and 2–16
  projected column list 2–16
  relational algebra for 2–18
  right outer join 2–22
  terminology 2–15
  types of 2–14
  views and 2–13
OUTER, SELECT option 1–4

P

Parallel steps
  EXPLAIN modifier 3–80
PARTITION column 1–11, 1–34
Partitioned primary index
  EXPLAIN modifier 3–82
POSITION statement 7–47
Positioned cursors 7–9
PP2
  error
    calculation scheme C–4
    codes, retryable C–6
    mapping C–5
PP2, data type codes B–7
PPRTEXT C–5
PREPARE statement 4–24, 4–63
Privileges, SELECT statement 1–7
Proportional row allocation in OLAP 1–70

Q

QUALIFY clause 3–140
  SELECT statement 1–64
QUALIFY, SELECT option 1–6

R

Random row allocation in OLAP 1–70
Random sampling in OLAP
  simple 1–69
  stratified random 1–69
  with replacement 1–70
  without replacement 1–70
REPEAT statement 9–136
Requests, multistatement 4–24
Retryable errors C–6
REWIND statement 7–49
Right outer join 2–22
RIGHT OUTER, SELECT option 1–4
ROLLBACK statement 3–141
  correlated subqueries 3–142
ROLLBACK statement, correlated subqueries and 1–56
Row allocation in OLAP
  proportional 1–70
  randomized 1–70

S

SAMPLE clause 1–67, 3–145
SAMPLE with proportional allocation clause 1–68
SAMPLE with RANDOMIZED ALLOCATION clause 1–68
SAMPLE WITH REPLACEMENT clause 1–67
SAMPLE without replacement 1–67
SAMPLE, SELECT option 1–6
SAMPLEID expression 1–75
SELECT INTO statement 1–15, 3–147
SELECT statement 1–2, 3–146
  ALL option 1–20
  correlated subqueries 1–10, 1–41, 1–49
  DISTINCT option 1–20
  FROM clause 1–22
  GROUP BY clause 1–57
  HAVING clause 1–61
  joins and 1–9
  noncorrelated subqueries 1–45
  ORDER BY clause 1–77
  PARTITION column in select-list 1–11
  privileges 1–7
  QUALIFY clause 1–64
  SAMPLE clause 1–67
  SAMPLE with proportional allocation 1–68
  SAMPLE with RANDOMIZED ALLOCATION 1–68
  SAMPLE WITH REPLACEMENT 1–67

SAMPLE without replacement 1–67
SAMPLEID expression 1–75
set operators and 1–9
simple SELECT 1–13
subqueries and 1–9, 1–37
success response 1–9
WITH clause 1–85
SELECT statement, cursors and 7–2
Self-join 2–10
SET BUFFERSIZE statement, SQL 5–13
SET CHARSET statement, SQL 5–15
SET CONNECTION statement, SQL 5–17
SET CRASH statement, SQL 5–19
SET statement 9–139
SET table kind option 3–111
Simple SELECT 1–13
Sort order
international 1–80
Japanese 1–80, 1–81
nulls and 1–78
Spool file 2–26
SQL
return codes C–5
SQL character string variables 4–19
SQL Descriptor Area. See SQLDA
SQL statements
ABORT 3–3
ASYNC statement modifier 6–6
BEGIN DECLARE SECTION 4–29
BEGIN TRANSACTION 3–7
CALL 3–13
CHECKPOINT 3–28
CLOSE 7–15
COMMENT (placing form) 3–31
COMMENT (returning form) 4–30
COMMIT 3–33
CONNECT 5–5
correlated subqueries 3–38, 3–155
DATABASE 4–32
DECLARE CURSOR 7–17
DECLARE CURSOR (dynamic SQL form) 7–19
DECLARE CURSOR (macro form) 7–21
DECLARE CURSOR (request form) 7–23
DECLARE CURSOR (selection form) 7–26
DECLARE CURSOR (stored procedures form) 7–29
DECLARE STATEMENT 4–34
DECLARE TABLE 4–35
DELETE 3–36
DELETE (basic/searched form) 3–42
DELETE (join condition form) 3–44
DELETE (positioned form) 3–46
DESCRIBE 4–55
ECHO 3–48
END DECLARE SECTION 4–37
END TRANSACTION 3–50
END-EXEC statement terminator 4–38
EXEC 4–39
EXEC SQL statement prefix 4–40
EXECUTE (dynamic SQL form) 4–59
EXECUTE (macro form) 3–52
EXECUTE IMMEDIATE 4–61
EXPLAIN modifier 3–55
FETCH (Embedded SQL Form) 7–33
FETCH (Stored Procedures Form) 7–37
GET CRASH 5–8
GROUP BY clause 3–104
HAVING clause 3–105
INCLUDE 4–42
INCLUDE SQLCA 4–44
INCLUDE SQLDA 4–46
INSERT 3–106
LOCKING modifier 3–119
LOGOFF 5–9
LOGON 5–11
MERGE 3–132
noncorrelated subqueries 3–38, 3–155
OPEN (Embedded SQL Form) 7–42
OPEN (Stored Procedures Form) 7–45
ORDER BY clause 3–139
POSITION 7–47
PREPARE 4–63
QUALIFY clause 3–140
REWIND 7–49
ROLLBACK 3–141
SAMPLE clause 3–145
SELECT 1–2, 3–146
SELECT INTO 1–15, 3–147
SET BUFFERSIZE 5–13
SET CHARSET 5–15
SET CONNECTION 5–17
SET CRASH 5–19
TEST 6–9
UPDATE (positioned form) 3–157
UPDATE (searched form) 3–148
UPDATE (upsert form) 3–160
USING row descriptor 3–168
WAIT 6–13
WHENEVER 4–48
WHERE clause 3–180
WITH clause 3–181
SQL. See also Embedded SQL
SQLCA fields C–7

SQLCA structure C–2
SQLCABC field C–7
SQLCAID field C–7
SQLCODE field C–7
SQLERRD field C–4, C–8
SQLERRM field C–8
support for DSNTIAR C–6
SQLERRP field C–8
SQLEXT field C–10
SQLWARN field C–9, C–10
warning condition reporting C–6
SQLCA, defined C–2
SQLCODE values 8–7
SQLCODE variable 6–4, 8–6, C–2, C–4
SQLCODE field
result reporting C–4
Teradata DBS error code and 8–8
Teradata DBS error code mapping to SQLCODE C–11
SQLDA structure
SQLD field B–4
SQLDABC field B–4
SQLDAID field B–4
SQLDATA field B–5
SQLIND field B–6
SQLN field B–4
SQLNAME field B–6
SQLTYPE field B–7
SQLDA, defined B–2
SQLEXCEPTION handler form. See DECLARE HANDLER statement
SQLSTATE codes, defined D–2
SQLSTATE mappings D–5
SQLSTATE variable 6–4, 8–2, C–2
SQLCODE to SQLSTATE exception mapping 8–4
Teradata DBS error code mapping to SQLSTATE C–15
Teradata DBS error codes and 8–4
SQLWARNING handler form. See DECLARE HANDLER statement
Stored procedure control statements 9–24
BEGIN - END 9–40
BEGIN-END, and ITERATE 9–43
BEGIN-END, and LEAVE 9–43
CASE 9–46
condition handling
benefits of 9–60
exceptions, transaction semantics for 9–66
handler forms 9–60
handler types 9–60
handlers, conditions raised by 9–69
nested compound statements, condition handlers in 9–63
nested stored procedures 9–75
precedence 9–65
rules for 9–63
status variable values 9–65
stored procedure, multiple condition handlers in 9–77
terms associated with 9–61
DECLARE 9–55
DECLARE HANDLER 9–58
CONTINUE type 9–60, 9–80
control statement handling 9–105
EXIT type 9–60, 9–85
NOT FOUND form 9–60, 9–101
SQLEXCEPTION form 9–60, 9–92
SQLWARNING form 9–60, 9–97
FOR 9–111
IF 9–118
ITERATE 9–125
LEAVE 9–129
LOOP 9–132
REPEAT 9–136
SET 9–139
SQLSTATE and condition handling 9–62
WHILE 9–141
Stored procedure lexicon
alias names 9–13
column names 9–13
cursors 9–11
data types supported 9–13
delimiters 9–14
for-loop variables 9–11
keywords 9–9
labels 9–11
lexical separators 9–14
literals 9–9
local variables 9–9
locking modifiers 9–14
names 9–8
parameters 9–9
status variables 9–15
Stored procedures
ACTIVITY_COUNT and 9–15
archive operations on 9–38
benefits of 9–3
body, defined 9–2
body, elements of 9–3
CALL statement and 9–6
chain recursion 9–34
comments in 9–16
compared to macros for tactical queries 9–148
completion condition handlers 9–25
complex tactical updates 9–145
control statements in 9–24, 9–39
cursor declaration and 9–26

data type codes 9–6
parameters and 9–6
DCL statements in 9–21
DDL statements in 9–17
DDL statements not supported in 9–20
debugging 9–150
defined 9–2
DML statements in 9–22
dynamic SQL 9–31
invoking 9–31
ownership of objects, created by 9–33
rules for 9–32
SQL statements not supported in 9–32
example 9–153
exception condition handlers 9–25
executing 9–6
initiating 9–6
mutual recursion 9–34
objects created by 9–28
performing 9–6
privileges 9–28
privileges, granting 9–5
recursive 9–34
restore operations on 9–38
restrictions on 9–7
rules for, SELECT INTO statement 1–18
self-referencing 9–34
source text, defined 9–2
SQL operations on 9–23
SQL statement errors and 9–29
SQL warnings and 9–29
SQLCODE and 9–15
SQLSTATE and 9–15
structure of, overview 9–2
tactical queries 9–145
tactical queries for security and auditing 9–147
transaction mode and 9–20, 9–21
unqualified objects and 9–30
using SQL statements in, rules for 9–27
when creator is not owner 9–28
when creator is owner 9–27
Subqueries 1–37
Syntax, how to read A–1

T

Table, derived 1–29, 2–26
Tactical queries
  stored procedures 9–145
  stored procedures and auditing 9–147
  stored procedures and complex tactical updates 9–145
  stored procedures and security 9–147
  stored procedures compared to macros 9–148
TDP, error codes C–4
tell_about_crash (TAC) option
  displaying 5–8
  setting 5–19
Teradata DBS, return codes C–5
Teradata Director Program. See TDP
TEST statement, SQL 6–9
Time, shown in EXPLAIN 3–56
TITLE phrase, WITH clause and 1–86

U

Unconstrained join 2–8
Unconstrained joins 1–34
Unicode, notation A–6
Updatable cursors 7–9
Updatable cursors. See Cursors
UPDATE (Upsert Form)
  EXPLAIN modifier 3–97
UPDATE statement (Positioned Form)
  correlated subqueries 3–158
UPDATE statement (positioned form) 3–157
UPDATE statement (Searched Form)
  correlated subqueries 3–150
UPDATE statement (searched form) 3–148
UPDATE statement (upsert form) 3–160
UPDATE statement, correlated subqueries and 1–52
USING row descriptor 3–168

V

Variables. See also Indicator variables
Views, joins on 2–37

W

WAIT statement 6–13
wait_across_crash (WAC) option
  setting 5–19
WHENEVER statement 4–48
WHERE clause 1–32, 3–180
  using a PARTITION column 1–34
WHERE, SELECT option 1–5
WHILE statement 9–141
WITH clause 1–85, 3–181
  TITLE clause and 1–86
WITH, SELECT option 1–6