

Teradata Vantage™ - SQL Stored Procedures and Embedded SQL

Release 17.10




July 2021

Copyright and Trademarks

Copyright © 2000 - 2021 by Teradata. All Rights Reserved.

All copyrights and trademarks used in Teradata documentation are the property of their respective owners. For more information, see [Trademark Information](#).

Product Safety

Safety type	Description
	Indicates a situation which, if not avoided, could result in damage to property, such as to equipment or data, but not related to personal injury.
	Indicates a hazardous situation which, if not avoided, could result in minor or moderate personal injury.
	Indicates a hazardous situation which, if not avoided, could result in death or serious personal injury.

Third-Party Materials

Non-Teradata (i.e., third-party) sites, documents or communications ("Third-party Materials") may be accessed or accessible (e.g., linked or posted) in or in connection with a Teradata site, document or communication. Such Third-party Materials are provided for your convenience only and do not imply any endorsement of any third party by Teradata or any endorsement of Teradata by such third party. Teradata is not responsible for the accuracy of any content contained within such Third-party Materials, which are provided on an "AS IS" basis by Teradata. Such third party is solely and directly responsible for its sites, documents and communications and any harm they may cause you or others.

Warranty Disclaimer

Except as may be provided in a separate written agreement with Teradata or required by applicable law, the information available from the Teradata Documentation website or contained in Teradata information products is provided on an "as-is" basis, without warranty of any kind, either express or implied, including the implied warranties of merchantability, fitness for a particular purpose, or noninfringement.

The information available from the Teradata Documentation website or contained in Teradata information products may contain references or cross-references to features, functions, products, or services that are not announced or available in your country. Such references do not imply that Teradata Corporation intends to announce such features, functions, products, or services in your country. Please consult your local Teradata Corporation representative for those features, functions, products, or services available in your country.

The information available from the Teradata Documentation website or contained in Teradata information products may be changed or updated by Teradata at any time without notice. Teradata may also make changes in the products or services described in this information at any time without notice.

Machine-Assisted Translation

Certain materials on this website have been translated using machine-assisted translation software/tools. Machine-assisted translations of any materials into languages other than English are intended solely as a convenience to the non-English-reading users and are not legally binding. Anybody relying on such information does so at his or her own risk. No automated translation is perfect nor is it intended to replace human translators. Teradata does not make any promises, assurances, or guarantees as to the accuracy of the machine-assisted translations provided. Teradata accepts no responsibility and shall not be liable for any damage or issues that may result from using such translations. Users are reminded to use the English contents.

Feedback

To maintain the quality of our products and services, e-mail your comments on the accuracy, clarity, organization, and value of this document to: docs@teradata.com.

Any comments or materials (collectively referred to as "Feedback") sent to Teradata Corporation will be deemed nonconfidential. Without any payment or other obligation of any kind and without any restriction of any kind, Teradata and its affiliates are hereby free to (1) reproduce, distribute, provide access to, publish, transmit, publicly display, publicly perform, and create derivative works of, the Feedback, (2) use any ideas, concepts, know-how, and techniques contained in such Feedback for any purpose whatsoever, including developing, manufacturing, and marketing products and services incorporating the Feedback, and (3) authorize others to do any or all of the above.

Contents

Chapter 1: Introduction to SQL Stored Procedures and Embedded SQL	7
Changes and Additions	7
Chapter 2: Stored Procedure and Embedded SQL Overview	8
Stored Procedure Overview	8
Embedded SQL Overview	10
Chapter 3: SQL Cursors	14
Why Cursors Are Necessary	14
Types of Cursors	15
Cursor States and Positions	15
How Cursors Are Incremented	16
Cursors and Stored Procedures	16
Cursors and Embedded SQL	18
Transactions and Cursors	20
Positioned Cursors	21
Chapter 4: SQL Cursor Control and DML Statements	27
ALLOCATE	27
CLOSE	29
DEALLOCATE PREPARE	31
DECLARE CURSOR	32
DELETE (Positioned Form)	47
EXECUTE	48
EXECUTE IMMEDIATE	50
FETCH (Embedded SQL Form)	51
FETCH (Stored Procedures Form)	55
OPEN (Embedded SQL Form)	60
OPEN (Stored Procedures Form)	62
POSITION	64
PREPARE	66
REWIND	68
SELECT AND CONSUME ... INTO	68
SELECT ... INTO	71
UPDATE (Positioned Form)	78
Chapter 5: Result Code Variables	82
SQLSTATE	82
SQLCODE	84

ACTIVITY_COUNT	87
Result Code Variables in Stored Procedures	89
Chapter 6: SQL Stored Procedures	91
Granting Privileges on Stored Procedures	91
Checking Privileges for Stored Procedures	91
Rules for Using SQL Statements in Stored Procedures	93
Executing a Stored Procedure	95
Recompiling a Stored Procedure	96
Restrictions on Stored Procedures	97
Stored Procedure Lexicon	98
DDL Statements in Stored Procedures	106
DML Statements in Stored Procedures	109
DCL Statements in Stored Procedures	111
Diagnostics Statements in Stored Procedures	111
SQL Operations on Stored Procedures	112
Control Statements in Stored Procedures	113
Completion, Exception, and User-defined Condition Handlers	113
Cursor Declarations	114
Returning Result Sets from a Stored Procedure	114
Using Dynamic SQL in Stored Procedures	117
Recursive Stored Procedures	121
Stored Procedures and Tactical Queries	123
Debugging Stored Procedures	127
Sample Stored Procedure	129
Chapter 7: Condition Handling	135
Benefits of Condition Handling	135
Condition Handling Terms	135
SQLSTATE	136
Diagnostics Area	137
Conditions and Condition Handlers	137
Condition Handler Types	138
Raising Conditions	138
Control Statement Handling	139
Condition Handler Rules	139
Rules for Condition Handlers in Nested Compound Statements	143
Status Variable Values	144
Precedence of Specific Condition Handlers	145
Exception Condition Transaction Semantics	146
Conditions Raised by a Handler Action	149
Rules for Reporting Handler Action-Raised Conditions	151
Multiple Condition Handlers in a Stored Procedure	156
Statement-Specific Condition Handling	158
DECLARE CONDITION	164

DECLARE HANDLER (Basic Syntax)	169
DECLARE HANDLER (CONTINUE Type)	171
DECLARE HANDLER (EXIT Type)	175
DECLARE HANDLER (SQLEXCEPTION Type)	181
DECLARE HANDLER (SQLWARNING Type)	185
DECLARE HANDLER (NOT FOUND Type)	188
Diagnostics Area	191
Diagnostic Statements	194
Chapter 8: Host Variables and Multistatement Requests	228
Host Structures	228
Host Variables	229
Input Host Variables	233
Output Host Variables	235
SQL Character Strings as Host Variables	238
Indicator Variables	239
Multistatement Requests with Embedded SQL	242
Chapter 9: SQL Control Statements	245
BEGIN END	245
CASE	251
DECLARE	260
FOR	263
IF	270
ITERATE	274
LEAVE	277
LOOP	280
REPEAT	281
SET	283
WHILE	285
Chapter 10: Static Embedded SQL Statements	288
Statements for Positioned Cursors	288
BEGIN DECLARE SECTION	288
COMMENT (Returning Form)	289
DATABASE	290
DECLARE STATEMENT	291
DECLARE TABLE	292
END DECLARE SECTION	294
END-EXEC Statement Terminator	294
EXEC	295
EXEC SQL Statement Prefix	296
WHENEVER	299
INCLUDE	301
INCLUDE SQLCA	302

INCLUDE SQLDA	303
Chapter 11: Dynamic Embedded SQL Statements	305
Using Dynamic SQL	305
Performing SQL Statements Dynamically	305
Dynamic SQL Statement Syntax	306
Chapter 12: Client-Server Connectivity Statements	318
Connecting a Client Application to Vantage	318
CONNECT	320
GET CRASH	323
LOGOFF	323
LOGON	330
SET BUFFERSIZE	337
SET CHARSET	338
SET CONNECTION	339
SET CRASH	344
SET ENCRYPTION	347
Chapter 13: Multisession Asynchronous Programming With Embedded SQL	349
Multisession Programming With Embedded SQL	349
Multisession Asynchronous Request Programming Support	351
ASYNC Statement Modifier	351
TEST	356
WAIT	363
Appendix A: How to Read Syntax	373
Appendix B: SQL Descriptor Area (SQLDA)	375
Appendix C: SQL Communications Area (SQLCA)	381
Appendix D: SQLSTATE Mappings	389
Appendix E: SQL Stored Procedure Command Function Codes	401
Appendix F: Performance Considerations	405
Appendix G: Additional Information	406

Introduction to SQL Stored Procedures and Embedded SQL

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

Teradata Vantage™ - SQL Stored Procedures and Embedded SQL describes how to create server and client applications using SQL to manipulate data.

Because stored procedures and embedded SQL support some similar functionality, such as cursors and dynamic SQL, they are described in the same document.

Wherever possible, this document presents functionally similar statements in sections that cover both stored procedures and embedded SQL.

More information about developing applications using embedded SQL is found in *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446.

Changes and Additions

Date	Description
July 2021	Minor edits.

Stored Procedure and Embedded SQL Overview

Stored Procedure Overview

A *stored procedure*, a database object executed on the database, is a combination of SQL statements and control and condition handling statements that provide an interface to the database.

Note:

The term *stored procedure* refers to a stored procedure you write with SQL statements. The term *external stored procedure* refers to a stored procedure you write in C, C++, or Java.

Typically, a stored procedure consists of a:

- Procedure name
- Input and output parameters
- Procedure body.

For each stored procedure, the database includes a stored procedure table that contains the stored procedure body you write and the corresponding compiled stored procedure object code. Data dictionary tables contain stored procedure parameters and attributes.

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 (sometimes called a 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.

Procedure Body Elements

A procedure body can contain the following elements:

Stored procedure body of this type ...	Can contain ...
Single statement	one SQL statement or control statement, including dynamic SQL.

Stored procedure body of this type ...	Can contain ...
	Note: The following elements are <i>not</i> allowed: <ul style="list-style-type: none"> • Any declaration (local variable, condition, cursor, or condition handler) statement • A cursor statement (OPEN, FETCH, or CLOSE)
Compound statement	<ul style="list-style-type: none"> • Local variable declarations • Condition declarations • Cursor declarations • Condition handler declarations • Control statements • SQL Data Manipulation Language (DML), Data Definition Language (DDL), and Data Control Language (DCL) statements supported by stored procedures, including dynamic SQL.

You do not have to enclose a compound statement within BEGIN END keywords if the procedure body contains only one statement and does not contain any declarations.

Stored Procedure Benefits

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 provide the following benefits over equivalent embedded SQL applications:

- Better performance because of greatly reduced network traffic between the client and server.
- Better application maintenance because business rules are encapsulated and enforced on the server.
- Better transaction control.
- Better application security by restricting user access to procedures rather than requiring them to access data tables directly.
- Better application execution because all SQL language statements are embedded in a stored procedure to be executed 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 by handling errors internally in the nested procedures.

Related Information

- For more information about stored procedures, see [SQL Stored Procedures](#).
- For information about stored procedure control statements, see [SQL Control Statements](#).
- For information about external stored procedures, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

- For information on the syntax for creating stored procedures, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Embedded SQL Overview

Embedded SQL refers to SQL statements you execute or declare 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.

The 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 on. Aside from these rules, embedded SQL is host language-independent.

Special SQL Statements for Embedded SQL

Embedded SQL requires many SQL language constructs that are not supported for the interactive use of the language.

However, with few exceptions, you can use any SQL statement that can be executed interactively in an embedded SQL application. Exceptions include:

- Non-ANSI Teradata extensions ECHO and USING
- CREATE FUNCTION and REPLACE FUNCTION
- The following query logging statements: BEGIN QUERY LOGGING, END QUERY LOGGING, FLUSH QUERY LOGGING, and REPLACE QUERY LOGGING

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, and Cursors

Supported Host Languages

- C
- COBOL
- PL/I

Preprocessing Embedded SQL Statements

Because client programming languages do not understand SQL, a precompiler or preprocessor must preprocess SQL-containing source code to first comment out and then convert the SQL language elements into CLIV2 calls, before compiling them with the appropriate C, COBOL, or PL/I compiler.

Preprocessor2 is the Teradata precompiler and runtime SQL statement manager.

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:

- 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 and do not specify the INTO clause.
a data returning statement is executed dynamically	you <i>must</i> define a dynamic cursor, regardless of the number of response rows expected.

Rules

- EXEC SQL must prefix all embedded SQL statements.
- You must terminate all embedded SQL statements. 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.
- You must define a host variable between BEGIN DECLARE SECTION and END DECLARE SECTION statements.
- You must define a host variable before any SQL statement reference to it.
- You should draw all host variables from the same domain as their target columns.
- UDTs are not specifically supported for any form of embedded SQL.

However, embedded SQL applications can use SQL statements that reference UDTs, provided that the UDTs have a defined tosql or fromsql transform as appropriate.

You must have, at minimum, the UDTUSAGE privilege on any UDT columns you access from an application.

Additionally, the application must send and receive UDT data in the form of its external (non-UDT) data type.

- Host variables and columns can have the same names.
- All embedded SQL programs must contain one or both of the SQLSTATE and SQLCODE host variables to communicate status between the database and the client application:

The ANSI/ISO SQL-92 standard deprecates the use of SQLCODE and the ANSI/ISO 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 might also find it useful to include an ACTIVITY_COUNT result code variable in your embedded SQL applications.

- You should always test the value for SQLCODE or SQLSTATE (or both) after executing an embedded SQL statement.

Related Information

- Cursors: [SQL Cursors](#)
- Embedded SQL:
 - [Static Embedded SQL Statements](#)
 - [Dynamic Embedded SQL Statements](#)
 - *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446 for examples of embedded SQL applications in the supported client languages.
- Declaratives:
 - [SQL Cursors](#) for cursor declarations.
 - [Static Embedded SQL Statements](#) for all other embedded SQL declaratives.
- UDTs:
 - Information about CREATE TRANSFORM in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

- *Teradata Vantage™ - SQL Data Control Language*, B035-1149 for details about the privileges required to access and manipulate UDT column values.
- SQLCODE and SQLSTATE:
 - [SQLSTATE](#)
 - [SQLCODE](#)
 - [WHENEVER](#) for more information about testing the values of SQLCODE and SQLSTATE.
- ACTIVITY_COUNT: [ACTIVITY_COUNT](#)

SQL Cursors

This section describes SQL cursors, including what they are and when and how to use them to point to rows in an SQL response set.

A *cursor* is a data structure that stored procedures and Preprocessor2 use at runtime to point to the result rows in a response set returned by an SQL query.

Stored procedures and embedded SQL also use cursors to manage inserts, updates, execution of multistatement requests, and SQL macros.

The syntax of functionally similar cursor control statements sometimes varies depending on whether they are used for stored procedures or embedded SQL. This document presents variant syntax forms with the individual cursor control statements. Several cursor control statements are valid only with embedded SQL.

Cursors are not valid for sessions you conduct interactively from a terminal using a query manager like BTEQ.

Why Cursors Are Necessary

Declared Cursors

This information does not apply to result set cursors.

An embedded or stored procedure SQL SELECT statement can 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 these applications.

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, which are SELECT statements that you write so that they 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.

Traditional programming languages such as COBOL, C, and PL/I are record-oriented, while relational databases and their operators are inherently set-oriented.

Cursors enable record-oriented languages to process set-oriented data. 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 held in a data structure known as a *pool file*, one row at a time.

Result Set Cursors

You can specify that a stored procedure return up to 15 result sets.

Related Information

For information on specifying a stored procedure to return results, see [Returning Result Sets from a Stored Procedure](#) and [DECLARE CURSOR \(Stored Procedures Form\)](#).

Types of Cursors

You can declare the following types of cursors in a DECLARE CURSOR statement:

- Dynamic
- Macro
- Request
- Selection
- Stored procedure

Note:

Stored procedures only support stored procedure-type cursors.

The following types refer to ways you can use a cursor to manipulate data:

- Positioned (updatable)
- Non-positioned (read only)

Both types are supported in embedded SQL and stored procedures.

Cursor States and Positions

A cursor can either be open or closed.

An open cursor can point to a 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, as executed by non-relational programming languages, requires 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.

Leave the result set cursors open to return the result sets to the caller or client. If the stored procedure closes the result set cursors, the result sets are deleted and not returned. The result sets are returned in the order they were opened.

How Cursors Are Incremented

The FETCH statement increments cursors to step through the response set rows.

Stored procedure cursors have some additional facilities that enable more flexibility than embedded SQL cursors possess:

- The FIRST and NEXT options of the FETCH statement and the SCROLL and NO SCROLL options of the DECLARE CURSOR statement 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.

Cursors and Stored Procedures

General Rules

No more than 15 cursors can be open at any one time in a stored procedure.

Construct cursor names from the following list of valid characters:

- uppercase letters
- lowercase letters
- \$
- @
- #
- digits
- underscores

The following statements with open cursors are not allowed in a stored procedure:

- POSITION
- REWIND
- SQL transaction statements

DECLARE CURSOR Statement and FOR Statement Cursors

- You can declare cursors in a DECLARE CURSOR statement or FOR loop control statement.
- The SELECT statement you specify in the FOR statement or DECLARE CURSOR (stored procedures form) statement is called the cursor specification.
- You can use both positioned and non-positioned cursors in a stored procedure.

- Cursors declared in a FOR statement and in a DECLARE CURSOR statement differ in the following ways.

FOR Loop Cursor	DECLARE CURSOR Cursor
<p>The scope of the cursor is confined to the FOR statement in which it is defined.</p> <p>The scope of column name or its correlation name in a FOR loop cursor is restricted to the body of the FOR statement.</p> <p>In the case of nested FOR statements, you can reference the cursor name you specify in an outer FOR statement in statements inside the inner FOR statement(s).</p>	<p>The scope of the cursor is the BEGIN END compound statement in which it is declared.</p> <p>The scope of a cursor is the compound statement and its nested compound statements, if any.</p> <p>In nested compound statements, the scope of a cursor you specify in an outer compound statement includes all the inner compound statements.</p>
A positioned DELETE or UPDATE statement referencing the cursor makes it updatable.	The FOR UPDATE option makes the cursor updatable.
<p>OPEN, FETCH and CLOSE take place implicitly as part of the FOR loop execution.</p> <p>Each iteration of the FOR statement fetches the next row, if it exists, for an open cursor.</p>	<p>You must explicitly specify OPEN, FETCH or CLOSE.</p> <p>If you specify CLOSE for a result set cursor, the result set will not be returned.</p>
You can label FOR statements in which cursors are declared.	You cannot label DECLARE CURSOR statements.
The FOR <i>cursor_name</i> statement implicitly opens a cursor for the SELECT statement you specify as the cursor specification.	The OPEN <i>cursor_name</i> statement opens a cursor for the SELECT statement you specify as the cursor specification.

Cursor Support

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.

FOR Loop Cursor Support

For a FOR loop 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 executes as follows:

- Fetches one row of data from the result set into the *for_loop_variable* on each iteration.
- 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.

DECLARE CURSOR Cursor Support

For cursors defined by a DECLARE CURSOR statement, you must submit explicit OPEN *cursor_name* and FETCH *cursor_name* statements.

Note that for a result set cursor, if you specify CLOSE, the result set will not be returned.

Related Information

For more details on the use of cursors in stored procedures, see [DECLARE CURSOR \(Stored Procedures Form\)](#), [Cursors](#) and [FOR](#).

Cursors and Embedded SQL

Cursor Rules

- No more than 16 cursors can be open at any one time in a given application.
- Whether or not a cursor can be positioned depends on how you set the precompiler directives TRANSACT or -tr, as specified by the following table:

Setting	Default Type
ANSI	Positioned
BTET	Not positioned

Teradata SQL does not support the ANSI/ISO SQL standard FOR READ ONLY and FOR UPDATE clauses for cursors.

- An application that opens 16 cursors can only issue one of the following as the next statement:
 - CLOSE
 - COMMIT (if in COMMIT mode)
 - FETCH
 - LOGOFF
 - POSITION
 - REWIND
- Construct cursor and dynamic statement identifiers 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 Support Statements in Preprocessor2

This section explains how the various SQL statements that support cursors fit into a coherent whole in embedded SQL.

1. Use DECLARE *cursor_name* CURSOR FOR the data returning statement to associate a cursor name with a multirow data returning statement.

You do not need to use a cursor to process a singleton SELECT.

2. Use the following statements to manipulate the declared cursor.

Statement	Function
OPEN <i>cursor_name</i>	Executes the request (or requests) defined by the DECLARE CURSOR statement
FETCH <i>cursor_name</i> 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 or SQLSTATE after each FETCH
DELETE ... WHERE CURRENT OF <i>cursor_name</i>	Deletes the currently fetched row from its base table
UPDATE ... WHERE CURRENT OF <i>cursor_name</i>	Updates the currently fetched row
POSITION <i>cursor_name</i>	Moves the cursor either forward or backward to the first row of the specified statement
REWIND <i>cursor_name</i>	Moves the cursor to the first row of the first (or only) statement of a request
CLOSE <i>cursor_name</i>	Closes the open <i>cursor_name</i> and terminates the data returning statement specified by the DECLARE CURSOR statement

Cursor Actions and Outcomes

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	Executes 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	Updates the contents of the current row.
Delete a row.	DELETE ... WHERE CURRENT OF	Deletes the current row from the table.
Close the cursor.	CLOSE	Terminates the retrieval process.

Transactions and Cursors

SQL Terminating Statements and Cursors

- COMMIT terminates all open cursors and commits the changes made by the cursors while a transaction was in progress. (ANSI session mode only).
- ROLLBACK (ANSI and Teradata session modes) or ABORT (Teradata session 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 session mode only).

Cursor Semantics for Implicit Transactions

For implicit transactions (cursor opened in Teradata session mode only):

- A FOR CURSOR loop opens the cursor as a holdable cursor, and its sensitivity is asensitive.
- A FOR CURSOR loop also supports transaction control statements.
- In the case of a DECLARE, OPEN, or FETCH CURSOR, the cursor is holdable and its sensitivity is asensitive.

Cursor Semantics for Explicit Transactions

For explicit transactions (cursor opened in Teradata mode or in ANSI session mode):

- A FOR CURSOR loop opens the cursor as a without hold cursor, and its sensitivity is asensitive. This implies that when the transaction is closed, the cursor is closed. If a cursor is asensitive, the visibility of significant changes to SQL data is implementation-dependent.
- A FOR CURSOR loop does not permit a COMMIT, ROLLBACK, or ABORT within the FOR loop. If the system detects a COMMIT, ROLLBACK, or ABORT during compile time, it returns an error and does not create the stored procedure.

If the system does not detect a COMMIT, ROLLBACK, or ABORT during compile time, it returns a run-time error. This is a failure in Teradata session mode, and the system closes the transaction and the cursor.

If the ROLLBACK or ABORT occurs in a nested call (in Teradata session mode), the system reports the nested call as having failed. The failure still applies for a subsequent FETCH CURSOR.

- In the case of a DECLARE, OPEN, OR FETCH CURSOR, the cursor is without hold and its sensitivity is asensitive. The system executes transaction control statements successfully, but the next FETCH or CLOSE CURSOR causes the system to return an error.

Cursor Holdability and Transaction and Session Termination

- The system does not close a holdable cursor if that cursor is open at the time the transaction terminates with a COMMIT.
- A holdable cursor that is closed at the time the transaction terminates remains closed.
- The system closes a holdable cursor if the transaction terminates with a ROLLBACK.
- The system closes and destroys a holdable cursor when the session in which it was created terminates.
- The system closes a cursor that is without hold when the transaction in which it was created terminates.

Cursor Sensitivity

If a cursor is open and the transaction in which the cursor was opened makes a significant change to data, the system determines whether that change is visible through that cursor before it is closed as follows:

- If a cursor is asensitive, the visibility of significant changes to data is implementation-dependent.
- If a cursor is insensitive, significant changes to data are not visible.
- If a cursor is sensitive, significant changes to data are visible.

Positioned Cursors

The ANSI/ISO SQL standard defines an updatable (positioned) cursor. This means that an application can define a cursor for a query and then update results rows using the same cursor.

Using a Positioned Cursor to Update or Delete a Row in a Stored Procedure

Following is the general process flow for updating or deleting a row using a FOR loop cursor in a stored procedure:

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 response 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.

Following is the general process flow for updating or deleting a row using a DECLARE CURSOR cursor in a stored procedure.

1. Specify a DECLARE CURSOR statement with the appropriate cursor specification.
2. Open the cursor with an OPEN *cursor_name* statement.
3. Execute FETCH statements to fetch one row at a time from the response set.

The next cursor movement depends on the scrollability option you specify.

Option	Next Cursor Movement
FIRST	First row in the result set
NEXT	Next row in the result set

4. Update or delete a 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.

Using a Cursor to Update a Row in Preprocessor2

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.

Positioned Cursors for SELECT AND CONSUME Statements in Preprocessor2

SELECT AND CONSUME statements in positioned cursors are not valid.

When you select a row from a queue table using a SELECT AND CONSUME statement, the system automatically consumes that row; therefore, it makes no sense to execute SELECT AND CONSUME statements in positioned cursors because once selected, a consumed row cannot be deleted or updated. This means that any SELECT AND CONSUME statement executed from a positioned cursor fails, even though it does not attempt to delete or update any rows.

Because all cursors coded in a Preprocessor2 application default to being positioned cursors when you precompile the code with the TRANSACT or -tr preprocessor directives set to ANSI, this means that you cannot execute SELECT AND CONSUME statements from a cursor in an ANSI-style embedded SQL application.

Features Supporting Positioned Cursors

Several features enable ANSI/ISO SQL:2011-standard positioned cursor functionality, including:

- WHERE CURRENT OF clause in DELETE and UPDATE statements
- FOR CHECKSUM clause of the LOCKING request modifier (embedded SQL only)
- FOR UPDATE clause in SELECT statements (stored procedures only)

WHERE CURRENT OF Clause

After you declare a positioned cursor, the WHERE CURRENT OF clause allows the DELETE and UPDATE statements to act on the row pointed to by the cursor.

For example, the following DELETE statement deletes the current customer row from the cursor named x01.

```
EXEC SQL
DELETE FROM customer
WHERE CURRENT OF x01;
```

FOR CHECKSUM Clause

Positioned 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.

Note:

The FOR CHECKSUM clause of the LOCKING request modifier, a Teradata extension to the ANSI/ISO SQL:2011 standard adds to this functionality.

When you do not specify a LOCKING request modifier, all SELECT statements use a READ level lock. Positioned updates and deletes both default to a WRITE severity 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.

The system returns an error to the application whenever any of the following requirements for CHECKSUM locking are not met:

- The object locked must be a table

- You must follow the LOCKING request modifier by a positioned cursor SELECT
- The table you specify in the LOCKING request 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 a positioned cursor.

Example: LOCKING with CHECKSUM

This example uses CHECKSUM locking on the table named t.

```
LOCKING TABLE t
FOR CHECKSUM
SELECT i, text
FROM t;
```

Rules for Using Positioned Cursors

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 a positioned 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, or functions

In a stored procedure, you can specify the FOR UPDATE clause in the DECLARE CURSOR statement to define a positioned cursor. If you do not specify the FOR UPDATE clause, the system returns a warning 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 an invalid SELECT, the system returns an error and rolls back the impacted transaction.

When a program attempts to UPDATE or DELETE a row using a WHERE CURRENT OF clause for a non-positioned cursor, the system returns an error stating that the cursor is not updatable.

The following table describes whether positioned cursors are valid based on the session mode under which they are created:

Session mode	Validity
ANSI	valid
Teradata	not valid

Performance Optimization Guidelines for Positioned Cursors

Row Hash Locks

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 positioned cursors, or use CHECKSUM locking to avoid locking conflicts that might prevent other applications from reading or updating the tables used by your application. Note that CHECKSUM locking is not supported for stored procedures.

When the number of row hash locks becomes excessive and a lock table overflow error occurs, Vantage issues a transaction level abort for any SQL application that receives the error.

Cursor Conflicts

Cursor conflicts can occur within a single transaction. Such conflicts occur when the system opens:

- The system opens two cursors 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.
- The system opens a cursor on a table, makes a searched update or delete on that table, and then the cursor attempts to execute a positioned update or delete on the table.
- The system opens a cursor on a table, makes a positioned update or delete through the cursor, and then attempts a searched update or delete on the same table.

The system returns a cursor conflict warning in all these situations, but executes the requested delete or update.

Related Information

- SELECT AND CONSUME, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

- Cursors and Preprocessor2, see [Cursor Rules](#) and *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446.
- Examples of implementing positioned cursors in an embedded SQL client application, see *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446.
- Examples of implementing positioned cursors in stored procedures, see [FOR](#).

SQL Cursor Control and DML Statements

This section describes the SQL cursor control statements and several SQL DML statements that use cursors.

ALLOCATE

Allows a calling stored procedure to fetch result sets returned by a stored procedure it calls.

ANSI Compliance

ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Stored procedures.

Syntax

```
ALLOCATE cursor_name CURSOR FOR PROCEDURE procedure_name ;
```

Syntax Elements

cursor_name

The name of a previously opened cursor to be referenced.

procedure_name

The name of the SQL stored procedure being called.

Usage Notes

- If the cursor was originally opened with NO SCROLL, then the cursor is positioned at “first row” of the result set.

For example, if the original cursor returned 10 rows and the stored procedure read 3 of the rows, then 7 rows are returned. The result set begins at the 4th row, but it appears as if it's the first row.

- If the cursor was originally opened with SCROLL, then the cursor is positioned immediately after the most recent row fetched.

For example, if the cursor returned all 10 rows, the initial position is the 4th row.

- If the cursor was opened with SCROLL, then the caller can reposition prior to the recent row.
- If the cursor was opened with NO SCROLL or SCROLL and there are multiple result sets, each set is fetched sequentially.
- If the procedure did not create any result sets or all of the result sets have been fetched, the SQLSTATE completion condition of '02001' is returned (that is, no additional dynamic result sets returned).
- Upon reaching the end of the first result set, the FETCH statement will be given an SQLSTATE of '02000' (that is, no data). To get the next result set, if any, the open cursor should be closed.
- If there are no additional result sets, the closing of the cursor will cause the CLOSE statement to return with an SQLSTATE completion condition of '02001' (that is, no additional dynamic result sets returned).
- If there are additional results sets, then a warning will be returned: '0100D' (that is, additional dynamic result sets returned).

Example: Using ALLOCATE

```

REPLACE PROCEDURE alloc007()
DYNAMIC RESULT SETS 1
BEGIN
    DECLARE EmpNo0    SMALLINT;
    DECLARE ProjId0   CHAR(8);
    DECLARE WkEnd0    DATE;
    DECLARE Hours0    DECIMAL(4,1);
    DECLARE ee0       CHAR(8);
    DECLARE ff0       VARCHAR(25);
    DECLARE gg0       DATE;
    DECLARE hh0       DATE;
    DECLARE ii0       DATE;
    CALL drs_temp5();
    ALLOCATE my_fetch CURSOR FOR PROCEDURE drs_temp5;
    FETCH FIRST FROM my_fetch INTO empno0,projid0,wkend0,hours0;
    INSERT INTO charges_temp2(empno0,projid0,wkend0,hours0);
    WHILE (SQLCODE = 0)
    DO
        FETCH NEXT FROM my_fetch INTO empno0,projid0,wkend0,hours0;
        IF (SQLCODE = 0)
        THEN
            INSERT INTO charges_temp2(empno0,projid0,wkend0,hours0);
        END IF;
    END WHILE;
    -- close the current result set cursor
    CLOSE my_fetch;
    -- see if there are result sets
    WHILE (SQLSTATE = '0100D')
    DO

```

```

-- allocate the next one.
ALLOCATE sp2 CURSOR FOR PROCEDURE drs_temp5;
WHILE (SQLCODE = 0)
DO
    FETCH NEXT FROM sp2 into ee0,ff0,gg0,hh0,ii0;
    IF (SQLCODE = 0)
    THEN
        INSERT INTO project_temp1(ee0,ff0,gg0,hh0,ii0);
    END IF;
END WHILE;
CLOSE sp2;
END WHILE;
END;

```

CLOSE

Closes an open cursor and releases the resources held by the cursor while it was open.

ANSI Compliance

ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Stored procedures and embedded SQL.

Syntax

```
CLOSE cursor_name
```

Syntax Elements

cursor_name

The name of an open cursor to be closed.

Usage Notes

Do not CLOSE a result set cursor. If the stored procedure closes the result set cursor, the result set is deleted and not returned.

The cursor identified by *cursor_name* must be 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 runtime exception occurs:

- SQLCODE is set to 7631
- SQLSTATE is set to '24501'

When control passes from a compound stored procedure statement, the stored procedure implicitly closes all open cursors declared within the body of that compound statement.

You cannot execute CLOSE as a dynamic SQL statement.

Example: Opening the Cursor

The following CLOSE example is valid because the cursor identified by cursor name projcursor is OPEN before the CLOSE.

```
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: Closing the Cursor

In the following example, CLOSE explicitly closes projcursor. The empcursor cursor is OPEN and there is no explicit CLOSE. In this case, empcursor closes 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

- For CLOSE, see [OPEN \(Embedded SQL Form\)](#) and [OPEN \(Stored Procedures Form\)](#).
- For COMMIT and ROLLBACK statements closing open cursors, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- For other DML statements that can be used by stored procedure and embedded SQL applications, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

DEALLOCATE PREPARE

Releases a previously prepared statement.

ANSI Compliance

ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Stored procedures.

Syntax

```
DEALLOCATE PREPARE statement_name ;
```

Syntax Elements

statement_name

The same identifier as *statement name* in a PREPARE statement.

Example: Using DEALLOCATE PREPARE

```

DECLARE sql_stmt1 VARCHAR(100);
DECLARE item INTEGER;

```

```

DECLARE price DECIMAL(8,2);
SET sql_stmt1 = 'INSERT INTO T1 (?,?);';
PREPARE stmt1 FROM sql_stmt1;
SET item = 1052;
SET price = 3.95;
EXECUTE stmt1 USING item, price;
SET item = 3967;
SET price = 10.85;
EXECUTE stmt1 USING item, price;
DEALLOCATE PREPARE stmt1;

```

DECLARE CURSOR

Defines and assigns a name to a cursor.

Note:

FOR statements also define cursors.

Invocation

Nonexecutable declaration.

Stored procedures and embedded SQL.

Usage Notes (All Forms)

- Each cursor declaration must specify a different cursor name.
- A cursor name cannot exceed 18 characters.
- 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, you can specify the DECLARE CURSOR statement either in the DATA DIVISION or in the PROCEDURE DIVISION.

DECLARE CURSOR (Dynamic SQL Form)

The dynamic SQL form of DECLARE CURSOR 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 macros and data returning statements
- A Teradata stored procedure

ANSI Compliance

ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Syntax

```
DECLARE cursor_name [SCROLL] CURSOR FOR statement_name
```

Syntax Elements

cursor_name

Any valid SQL identifier.

SCROLL

The declared cursor can fetch the row in the response set based on the FETCH orientation declared.

If you do not specify SCROLL, the cursor can only scroll forward to the next row in the response set.

Use SCROLL only when the dynamic SQL is a SELECT statement.

statement_name

The name associated with a previously prepared statement.

Usage Notes

- Valid prepared dynamic SQL statements are:
 - A single, non-data-returning, non-macro statement
 - A single SELECT statement (which you must specify without an INTO clause)
 - A single EXEC *macro_name* statement
 - A multistatement request, which can include any of the foregoing statements
- [Usage Notes \(All Forms\)](#)
- You must PREPARE the statement specified by *statement_name* before you OPEN the dynamic cursor within the same transaction.
- You can declare only one dynamic cursor for a given *statement_name*.
- You cannot specify a DELETE or UPDATE embedded SQL statement on a SELECT AND CONSUME cursor.

A cursor for a queue table is always read-only in PP2 ANSI mode. Therefore, a positioned DELETE or UPDATE (that is, deleting or updating the most current fetched cursor row) is not allowed for a queue table cursor in PP2 ANSI mode.

- A scrollable cursor is not allowed for multistatement requests in PP2 ANSI mode.

Example: Using Dynamic DECLARE CURSOR Statements

Dynamic DECLARE CURSOR statements take the following form:

```
DECLARE Ex CURSOR FOR DynStmt7
```

Related Information

- The dynamic SQL form of DECLARE CURSOR, see [DECLARE CURSOR \(Dynamic SQL Form\)](#)
- SCROLL, see [FETCH \(Embedded SQL Form\)](#).

DECLARE CURSOR (Macro Form)

The macro form of DECLARE CURSOR associates a cursor with a Teradata SQL macro.

ANSI Compliance

A Teradata extension to the ANSI/ISO SQL:2011 standard because macros are not defined in ANSI/ISO SQL.

Required Privileges

None.

Syntax

```
DECLARE cursor_name CURSOR FOR EXEC
[ database_name. ] macro_name [ ( parameter_list ) ]
```

Syntax Elements

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 executed.

parameter_list

The Teradata SQL macro parameters.

Usage Notes

- [Usage Notes \(All Forms\)](#)
- The system executes the macro when the cursor is opened. Then the application program accesses the results 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: Structuring the Macro DECLARE CURSOR Statement

Structure the macro DECLARE CURSOR statement as follows:

```
DECLARE Ex CURSOR FOR EXEC NewEmp
```

DECLARE CURSOR (Request Form)

The request form of DECLARE CURSOR associates a cursor with an arbitrary Teradata SQL request, typically a multistatement request specified within an SQL string literal.

ANSI Compliance

ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Syntax

```
DECLARE cursor_name CURSOR FOR 'request_specification'
```

Syntax Elements

cursor_name

The name of the cursor to be declared.

request_specification

A literal character string enclosed in apostrophes comprising any number of SQL statements separated by semicolons.

By default, the string is delimited by apostrophes (' ').

You can override this default using the QUOTESQL preprocessor parameter. Apostrophes syntactically distinguish the declaration of a request cursor from the other categories of cursor.

Usage Notes

- [Usage Notes \(All Forms\)](#)
- Statements in *request_specification* 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
- You can continue *request_specification* from line to line according to the syntax for continuation of string literals in the client language (embedded SQL only).
- Statements in *request_specification* cannot be Preprocessor2 declaratives (embedded SQL only).
- When the system opens the cursor, it updates the SQLCA to reflect the success (SQLCODE in the SQLCA is 0, SQLSTATE is set to '00000') of one of the following:
 - The first statement of the request
 - The failure of the request, where failure is defined as an implicit rollback of the transaction
- A failure condition always overrides a success report. If successful, the activity count displays 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 *request_specification* are data returning statements, the application program must use the POSITION statement to position to the appropriate result set to retrieve the response data set.
- OPEN automatically sets the position to the first statement of the request, so a 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: Omitting Details of Continuation of a Literal Character String

The following 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)

The selection form of DECLARE CURSOR associates a cursor with a SELECT or other data returning statement.

ANSI Compliance

ANSI/ISO SQL:2011-compatible with Teradata extensions.

Required Privileges

None.

Syntax

```
DECLARE cursor_name [ SCROLL ] CURSOR FOR {
  COMMENT |
  EXPLAIN |
  HELP |
  SHOW |
  SELECT |
  SELECT AND CONSUME
}
```

Syntax Elements

cursor_name

The name you assign to this cursor.

The name can be any valid SQL identifier.

SCROLL

The declared cursor can fetch the row in the response set based on the FETCH orientation declared.

If you do not specify SCROLL, the cursor can only scroll forward to the next row in the response set. This is the default.

Use SCROLL only when the SQL statement is a SELECT statement.

COMMENT

A valid SQL comment-returning COMMENT statement.

EXPLAIN

A valid SQL EXPLAIN request modifier.

HELP

A valid SQL HELP statement.

SHOW

A valid SQL SHOW statement.

SELECT

A valid embedded SQL SELECT statement.

SELECT AND CONSUME

A valid embedded SQL SELECT AND CONSUME statement.

Usage Notes

- [Usage Notes \(All Forms\)](#)
- You cannot specify the SQL WITH ... BY clause.
- The SELECT privilege is required on all tables specified in the cursor declaration or on the database containing the tables.
- You must define each host variable referenced in the cursor specification before the selection DECLARE CURSOR statement.
- If the table you identify in the FROM clause is a grouped view (a view defined using a GROUP BY clause), *table_expression* cannot contain any of the following clauses:
 - WHERE
 - GROUP BY
 - HAVING
- If you specify a UNION operator, the description of each result table in the union must be identical except for column names. All columns of the spool table formed by the union of the result tables are unnamed.
- The result is the union of the individual result tables for each query in the union, with any duplicate rows eliminated.
- If you specify an ORDER BY clause, each of its column specifications must specify a column of the spool table by name.
- Any unsigned integer column reference in the ORDER BY clause must specify a column of the spool table by relative number.
- You can reference a named column either by a column specification or an unsigned integer.
- You must reference an unnamed column by an unsigned integer.
- You cannot specify a DELETE or UPDATE embedded SQL statement on a SELECT AND CONSUME cursor.
- A cursor for a queue table is always read-only in PP2 ANSI mode. Therefore, a positioned DELETE or UPDATE (that is, deleting or updating the most current fetched cursor row) is not allowed for a queue table cursor in PP2 ANSI mode.
- A scrollable cursor is not allowed for multistatement requests in PP2 ANSI mode.

Example: Using DECLARE CURSOR to Order a Project by proj_id

```
DECLARE ex1 CURSOR FOR
SELECT *
FROM project
ORDER BY proj_id
```

Example: Using DECLARE CURSOR to Order a Project by Numbers

```
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: Using DECLARE CURSOR to Order a Project by deptname and name

```
DECLARE ex2 CURSOR FOR
EXPLAIN SELECT deptname, name
FROM employee, department
WHERE employee.deptno = department.deptno
ORDER BY deptname, name
```

Example: Using DECLARE CURSOR for an Employee Help Table

```
DECLARE ex4 CURSOR FOR
HELP TABLE employee
```

Related Information

- SCROLL, see [FETCH \(Embedded SQL Form\)](#).

- COMMENT and SHOW, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- EXPLAIN, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- SELECT and SELECT AND CONSUME, see the restrictions in the usage notes.

DECLARE CURSOR (Stored Procedures Form)

The stored procedures form of DECLARE CURSOR associates a cursor with a SELECT or other data returning statement within the body of a stored procedure FOR statement.

ANSI Compliance

The ONLY keyword, and the TO CALLER and TO CLIENT options are Teradata extensions to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Syntax

```
DECLARE DECLARE cursor_name [ [NO] SCROLL ] CURSOR
  [ WITHOUT RETURN |
    WITH RETURN [ONLY] [ TO { CALLER | CLIENT } ]
  ]
  FOR { cursor_specification [ FOR { READ ONLY | UPDATE } ] |
       statement_name
    } [;]
```

Syntax Elements

cursor_name

The name of the cursor to be declared.

SCROLL

Specifies whether the declared cursor can fetch the next row in the result set, or fetch the first row in the result set from any location in that set.

- SCROLL can either scroll forward to the next row in a result set or scroll directly to the first row in the result set.
- NO SCROLL, which is the default, can only scroll forward to the next row in the result set.

NO SCROLL

Specifies whether the declared cursor can fetch the next row in the result set, or fetch the first row in the result set from any location in that set.

- SCROLL can either scroll forward to the next row in a result set or scroll directly to the first row in the result set.
- NO SCROLL, which is the default, can only scroll forward to the next row in the result set.

WITHOUT RETURN

The procedure does not return a result set.

WITHOUT RETURN is the default.

WITH RETURN

Specifies:

- that the cursor is a result set cursor.
- to return a result set to the current stored procedure (the procedure that opened the cursor) and to the caller of the procedure.

WITH RETURN TO CALLER

Specifies:

- that the cursor is a result set cursor.
- to return a result set to the current stored procedure (the procedure that opened the cursor) and to the caller of the procedure.

WITH RETURN ONLY

Specifies:

- that the cursor is a result set cursor.
- to return a result set only to the caller of the stored procedure.

WITH RETURN ONLY TO CALLER

Specifies:

- that the cursor is a result set cursor.
- to return a result set only to the caller of the stored procedure.

WITH RETURN TO CLIENT

Specifies:

- that the cursor is a result set cursor.
- to return a result set to the client (application such as BTEQ) and to the current stored procedure (the procedure that opened the cursor).

WITH RETURN ONLY TO CLIENT

Specifies:

- that the cursor is a result set cursor.
- to return a result set only to the client (application such as BTEQ).

cursor_specification

The SELECT statement that retrieves the rows the cursor reads or updates.

READ ONLY

Specifies that you can only use the cursor to read the rows in the result set.

This is the default.

UPDATE

Specifies that you can use the cursor to update or delete the rows in the result set.

If UPDATE is specified, one of the following must also be specified:

- WITH RETURN ONLY
- WITH RETURN ONLY TO CALLER
- WITH RETURN TO CLIENT
- WITH RETURN ONLY TO CLIENT

statement_name

The identifier for the dynamic form of the DECLARE CURSOR statement.

Usage Notes

- [Usage Notes \(All Forms\)](#)
- General Rules
 - You must specify a cursor declaration 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, NO SCROLL is the default and the cursor can only scroll forward.
 - If you do not specify an explicit updatability clause, FOR READ ONLY is the default.
 - To create a positioned cursor, specify an explicit FOR UPDATE clause. That is, the cursor can be used for delete and update operations on its result rows.
- Rules for Returning Result Sets
 - Specify the number of result sets with the DYNAMIC RESULT SETS clause in the CREATE/REPLACE PROCEDURE statement.

- If you specify one of the WITH RETURN clauses, the stored procedure returns a result set to the current procedure, to the client, or to the caller for each result set cursor you declare.
- Specifying WITH RETURN is the same as specifying WITH RETURN TO CALLER.
- Specifying WITH RETURN ONLY is the same as specifying WITH RETURN ONLY TO CALLER.
- If you specify WITH RETURN ONLY, the stored procedure that opens the cursor cannot use the cursor to fetch rows from the result set.
- If you specify WITH RETURN or WITH RETURN TO CALLER, you cannot specify FOR UPDATE.
- If you specify TO CLIENT, the result set is returned to the client application even if called from a nested stored procedure.
- If you specify WITH RETURN ONLY TO CLIENT, the stored procedure returns the result set to the client, not to the stored procedure or external stored procedure that called the target procedure.
- If more than one stored procedure specifies WITH RETURN, the system returns the result sets in the order opened.
- Leave the result set cursors open to return the result sets to the current stored procedure, caller, or client. The system does not return a result set if the result set cursor is closed.

The returned result set:

- Inherits the response attributes (response mode, keep response, LOB response mode) of the caller, not of the stored procedure that created it. For example, if you submit a CALL in BTEQ, the system sends the result set to the stored procedure in Indicator mode and sends the result set to BTEQ in Field mode.
- Is based on the collation of the stored procedure, not the caller or session collation.

Example: Using a Cursor in a Stored Procedure

The following example illustrates the correct use 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: Using an Implicit FOR READ ONLY Cursor

The following example illustrates an implicit FOR READ ONLY cursor. The stored procedure does not specify a FOR UPDATE clause in the declaration of empcursor, so it 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: Using an Explicitly Declared FOR READ ONLY Cursor

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: Using a FOR UPDATE Cursor

The following example illustrates a FOR UPDATE cursor.

```

CREATE PROCEDURE sp1()
BEGIN
    DECLARE empcursor CURSOR FOR
        SELECT *
        FROM employee

```

```

    WHERE deptcode = 101
  FOR UPDATE;
    ...
END;

```

Example: Using WITH RETURN ONLY TO CLIENT

The following example illustrates the use of WITH RETURN ONLY TO CLIENT.

```

DECLARE results1 CURSOR WITH RETURN ONLY TO CLIENT FOR
    SELECT store, item, on_hand
    FROM inventory
    ORDER BY store, item;
OPEN results1;

```

Example: Using Dynamic SQL Statements in a Stored Procedure

The following example illustrates the use of dynamic SQL statements in a stored procedure defined without a WITH RETURN clause.

```

CREATE PROCEDURE GetEmployeeSalary
(IN EmpName VARCHAR(100), OUT Salary DEC(10,2))
BEGIN
    DECLARE SqlStr VARCHAR(1000);
    DECLARE C1 CURSOR FOR S1;
    SET SqlStr = 'SELECT Salary FROM EmployeeTable WHERE EmpName = ?';
    PREPARE S1 FROM SqlStr;
    OPEN C1 USING EmpName;
    FETCH C1 INTO Salary;
    CLOSE C1;
END;

```

Example: Using the Dynamic Form of the DECLARE CURSOR Statement

The following example illustrates the dynamic form of the DECLARE CURSOR statement. The cursor statement specifies a result cursor with a dynamic SELECT.

```

DECLARE statement1_str VARCHAR(500);
DECLARE result_set CURSOR WITH RETURN ONLY FOR stmt1;

SET statement1_str = 'SELECT store, item, on_hand FROM inventory ORDER BY
store, item;';
PREPARE stmt1 FROM statement1_str;
OPEN result_set;

```

Example: Using a Dynamic Parameter Marker

The following example illustrates the use of a dynamic parameter marker. The data for the dynamic parameter marker is passed in the OPEN statement.

```
DECLARE Store_num INTEGER;
DECLARE statement1_str VARCHAR(500);
DECLARE result_set CURSOR WITH RETURN ONLY FOR stmt1;

SET statement1_str = 'SELECT store, item, on_hand'
                    ' FROM inventory WHERE store = ? ORDER BY store, item;';
PREPARE stmt1 FROM statement1_str;
SET Store_num = 76;
OPEN result_set USING Store_num;
```

Related Information

- SCROLL, see [FETCH \(Stored Procedures Form\)](#).
- NO SCROLL, see [FETCH \(Stored Procedures Form\)](#).
- Specifying the number of result sets with the DYNAMIC RESULT SETS clause in the CREATE/REPLACE PROCEDURE statement, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- Differences between DECLARE CURSOR and FOR statements, see [DECLARE CURSOR Statement and FOR Statement Cursors](#).
- Positioned cursors, see [Positioned Cursors](#).
- Dynamic result sets, see [Returning Result Sets from a Stored Procedure](#).

DELETE (Positioned Form)

Deletes the most current fetched row from an updatable cursor invocation.

ANSI Compliance

ANSI/ISO SQL:2011-compliant.

This form is not valid in Teradata session mode.

Required Privileges

You must have the DELETE privilege on the table.

Use care 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.

Invocation

Executable.

Stored procedures and embedded SQL.

Syntax

```
{ DELETE | DEL } FROM table_name WHERE CURRENT OF cursor_name
```

Syntax Elements

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.

Usage Notes

- 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 on Using WHERE CURRENT OF
 - *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 that you SELECT in the updatable cursor request.
 - You must position the referenced cursor at a valid row via the FETCH statement.

Example: Deleting a Row with a Cursor

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

- *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146

EXECUTE

Executes a prepared statement in a stored procedure.

ANSI Compliance

ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Stored procedures only.

Syntax

```
EXECUTE statement_name [ USING { SQL_identifier | SQL_parameter } [,...] ]
```

Syntax Elements***statement_name***

The name associated with the previously prepared statement.

SQL_identifier

A valid SQL identifier.

SQL_parameter

An SQL parameter.

Usage Notes

- EXECUTE cannot be used with any of the following: a dynamic data returning statement or a dynamic multistatement request
- EXECUTE itself cannot be performed as a dynamic SQL statement.

The following rules apply to the USING clause:

- The USING clause identifies variables used as input to the SQL statement specified by *statement_name*.
- The specified *statement-name* must be a valid and declared prior to the EXECUTE statement.
- 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 must correspond to the nth parameter marker.
- The arguments must be compatible with the target. Necessary compatible conversion will be performed.

Example: Executing a Prepared Statement in a Stored Procedure

```

CREATE PROCEDURE sales_update(store_table VARCHAR(10),
                                item INTEGER,
                                price DECIMAL(8,2) )
    BEGIN
        DECLARE sql_stmt VARCHAR(100);
        SET sql_stmt = 'UPDATE ' || store_table || ' SET
            store_price=' || price || '
        WHERE store_item =' || item;
        PREPARE stmt1 FROM SQL_stmt;
        EXECUTE stmt1;
    END;

```

Example: Executing a Prepared Statement in a Stored Procedure to Update the Price of an Item

```

CREATE PROCEDURE sales_update(store_table VARCHAR(10),
                                item INTEGER,
                                price DECIMAL(8,2) )
    BEGIN
        DECLARE price_read DECIMAL(8,2);
        DECLARE sql_stmt VARCHAR(100);
        SET sql_stmt = 'UPDATE ' || store_table || ' SET store_price=?
        WHERE store_item = ?;';
        PREPARE stmt1 FROM sql_stmt;
        EXECUTE stmt1 USING price, item;
    END;

```

EXECUTE IMMEDIATE

Prepares and executes a statement in a stored procedure.

ANSI Compliance

ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Stored procedures only.

Syntax

```
EXECUTE IMMEDIATE statement_name
```

Syntax Elements

statement_name

One of the following:

- a literal
- a reference to an SQL string variable
- a parameter

Example: Using EXECUTE IMMEDIATE

```
CREATE PROCEDURE sales_update(store_table VARCHAR(10),
                                item INTEGER,
                                price DECIMAL(8,2) )
    BEGIN
        DECLARE sql_stmt VARCHAR(100);
        SET
            SET sql_stmt = 'UPDATE ' || store_table || ' SET
store_price=' || price ||
                                ' WHERE store_item =' || item;
        EXECUTE IMMEDIATE sql_stmt;
    END;
```

FETCH (Embedded SQL Form)

Positions a cursor on the next row (default) or any specified row of a response set, and assigns the values in that row to host variables.

ANSI Compliance

ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```

FETCH [
  NEXT |
  PRIOR |
  FIRST |
  LAST |
  ABSOLUTE n |
  RELATIVE n
] cursor_name
[
  INTO [,...] |
  USING DESCRIPTOR [:] descriptor_area
]

```

Syntax Elements

into_spec

```
[:] host_variable_name [ [INDICATOR] :host_indicator_name ]
```

NEXT

Fetches the next row from the response set relative to the current cursor position.

NEXT is the default.

PRIOR

Fetches the prior row from the response set relative to the current cursor position.

FIRST

Fetches the first row of the response set.

LAST

Fetches the last row of the response set.

ABSOLUTE *n*

Fetches the n^{th} row of the response set relative to:

- The first row of the set, if *n* is a positive number.
- The last row of the set, if *n* is a negative number.

n can be a *host_variable_name* or an *integer_constant*.

The data types for the host variable can be any 8-byte numeric data type with zero scale.

An integer_constant can be up to 31 digits.

RELATIVE *n*

Fetches the n^{th} row of the response set:

- forward by the value of *n*, if *n* is a positive number,
- backward by the value of *n*, if *n* is a negative number,

relative to the current cursor position.

n can be a host_variable_name or an integer_constant.

The data types for the host variable can be any 8-byte numeric data type with zero scale.

An integer_constant can be up to 31 digits.

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 character preceding the name is optional.

host_indicator_variable

The indicator variable.

The colon character 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.

Usage Notes

- Scrollable Cursors

When you open a scrollable cursor, the cursor is positioned before the first row of the response set. You can fetch using one of the FETCH orientation keywords.

You can open scrollable cursors in a multisession connection to enhance performance for access. When an application does not access rows sequentially, you may achieve better performance by setting the response buffer size equal to the fetch row size. You can try different response buffer sizes to achieve the best performance.

- You should define an SQLDA.

- You cannot execute FETCH as a dynamic SQL statement.
- Multistatement requests are not allowed in scrollable cursor FETCH.
- Scrollable cursor FETCH is not allowed in PP2 COMMITTED mode.
- You must previously declare the cursor identified by *cursor_name*.
- Use the INTO clause with cursors that you declared with either static or dynamic SQL statements.

The USING DESCRIPTOR clause is intended for use with selection cursors that you declared with dynamic SQL statements.

- The number of columns the request returns must match the number of host variable specifications or the number of elements in the SQLVAR array of the SQLDA. In other words, the number of columns returned in the result set must equal the value of the SQLD field.
- The main host variable you 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.

The only valid exception is if the main host variable data type is approximate numeric, in which case the spool table column data type can be either approximate numeric or exact numeric.

- If you specify the USING DESCRIPTOR clause, verify that the SQLDATA and SQLIND pointer fields in SQLDA are set to point to the appropriate host variables.

Because the COBOL language provides no support for setting pointer values, the database 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. Programmers coding in this COBOL 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: <ul style="list-style-type: none"> ◦ Requested data for successful data returning statements. ◦ SQLCODE +100 and SQLSTATE '02xxx' for no data. ◦ An error for non-rollback inducing SQL < 0 and SQLSTATE > '02xxx'.
<ul style="list-style-type: none"> ◦ is positioned on or after the last row or ◦ does not return data 	<ul style="list-style-type: none"> ◦ the cursor is positioned after the last row ◦ the system assigns +100 to SQLCODE ◦ the system assigns '02xxx' to SQLSTATE ◦ the host variables remain unchanged
is positioned before a row	<ul style="list-style-type: none"> ◦ the cursor is positioned on that row ◦ the system assigns values from the row to the host variables you specified in the INTO clause or in the output SQLDA.
is positioned on a row other than the last row	<ul style="list-style-type: none"> ◦ the cursor is positioned on the row immediately following that row ◦ the system assigns values from the new current row to the host variables you specified in the INTO clause or in the output SQLDA.

- The system assigns values to the host variables you specified in the INTO clause, or in the SQLVAR array in the SQLDA in the order in which you specified the host variables. The system assigns a value to SQLSTATE and SQLCODE last.
- If an error occurs in assigning a value to a host variable, the system stops assigning values to host variables, and assigns one of the following values to the result code variables.

SQLCODE	SQLSTATE
-303	'22509'
-304	'22003'
-413	'22003'

- The following table indicates what the system assigns if a field in the returned data is NULL, depending on whether or not you specified a corresponding host variable.

IF a corresponding host variable is ...	THEN the system assigns ...
specified	-1 to the host indicator variable.
not specified	<ul style="list-style-type: none"> -305 to SQLCODE. '22002' to SQLSTATE.

In either case, the host variables remain unchanged.

- The following table indicates the host indicator value the system sets if a column value in the temporary table row is NOT NULL and you specified a corresponding indicator host variable.

IF ...	THEN the system sets the host indicator value 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.

- The system sets column values in the corresponding main host variables according to the rules for host variables.

Related Information

- descriptor_area*, see *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446.
- Using a FETCH orientation other than NEXT, you must have declared a scrollable cursor. See [DECLARE CURSOR \(Selection Form\)](#).
- Scrollable cursors, see [SET BUFFERSIZE](#).

FETCH (Stored Procedures Form)

Positions a cursor to the next row (default) or any specified row of a response set and assigns the values in that row to local variables or parameters.

ANSI Compliance

ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Stored procedures only.

Syntax

```
FETCH [ [ NEXT | FIRST ] FROM ] cursor_name
      INTO { local_variable_name | parameter_reference } [, ...] ;
```

Syntax Elements

NEXT

Fetches the next row from the response set, if it exists.

NEXT is the default.

FIRST

Fetches the first row from the response set.

cursor_name

The name of an open selection cursor, including a cursor that was allocated, 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.

Both predefined data types and UDTs (except VARIANT_TYPE UDTs) are supported.

parameter_reference

The name of the INOUT or OUT parameter into which the fetched row is to be assigned.

Usage Notes

- When There Are No Rows in the Response Set

If there are no rows in the response set at the time you execute FETCH, the system returns the following runtime exception:

- SQLCODE is set to 7362
- SQLSTATE is set to '02000'

The system handles this runtime warning condition within the procedure. If it is not handled, the procedure continues from the next statement following the failed fetch operation.

- Assignment Order for Fetched Rows

The system assigns row values to local variables or output parameters in the order you declared those variables and parameters in the INTO list.

- General Rules

The specified cursor must be open when you submit FETCH.

If the cursor is not open, the system returns the following runtime exception:

- SQLCODE is set to 7631
- SQLSTATE is set to '24501'

The number of values FETCH returns must match the number of local variables and output parameters you declared in the INTO list.

IF a mismatch is identified at ...	THEN the database returns ...
compilation	compilation error SPL1027.
runtime	a runtime exception: <ul style="list-style-type: none"> SQLCODE is set to 7608 SQLSTATE is set to 'T7608'

The data types of the fetched columns must match the data types you specified for the local variables or OUT parameters to which they are assigned.

This is particularly true for UDT types, because the system does not implicitly apply any casts defined for a type.

To work around this restriction, you can do either:

- Explicitly CAST data types in the cursor select list from a predefined type to a UDT or from a UDT to a predefined type if you have also defined a cast to the target type that specifies the AS ASSIGNMENT option.
- Call a method that returns the target type.

You cannot indicate a simple target specification that names table columns in the INTO list. If you specify a non-valid INTO list, the system returns error SPL1028 during compilation.

Instead, you must specify output parameters (INOUT and OUT) or local variables.

- Rules for FIRST and NEXT

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 stored procedure positions the cursor after the last row and the system returns the following completion condition:

- SQLCODE is set to 7632
- SQLSTATE is set to '02000'

The output parameter or local variable you specified in the INTO list for this value is not changed in this case.

If you specify FIRST, you must specify SCROLL in the declaration for the referenced cursor.

If you do not specify SCROLL, the system returns error SPL1132 at compilation.

If you specify FIRST, but there is no data, the system returns the following completion condition:

- SQLCODE is set to 7632
- SQLSTATE is set to '02000'

The output parameter or local variable you specified in the INTO list for this value is not changed in this case.

Example: Returning Columns Assigned to Local Variables with Matching Data Types

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)
  DO
    FETCH projcursor INTO var1, var2;
  END WHILE;
  CLOSE projcursor;
END;
```

Example: Using the FETCH Statement and the WHILE Loop

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 CHARACTER(50))
BEGIN
  DECLARE var1 INTEGER;
  DECLARE projcursor CURSOR FOR
    SELECT projid, projectdesc
    FROM project;
  OPEN projcursor;
  WHILE (SQLCODE = 0)
  DO
    FETCH projcursor INTO var1, par1;
  END WHILE;
  FETCH projcursor INTO var1, par1;
  CLOSE projcursor;
END;
```

Example: Using Fetch Orientation in the FETCH Statement

The following example illustrates the usage of fetch orientation in the FETCH statement. Assume that the project table contains 10 rows at the time execution 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: Using FETCH FIRST

The following example illustrates the usage of FETCH FIRST. Assume that the project table is empty at the time execution 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)

Opens a declared cursor for an embedded SQL application and executes the SQL statement specified in its declaration.

ANSI Compliance

ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
OPEN cursor_name [
  USING {
    using_spec [,...] |
    DESCRIPTOR [:] descriptor_area
```

```
}
]
```

Syntax Elements

using_spec

```
[ : ] host_variable_name [ [INDICATOR] :host_indicator_name ]
```

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 character preceding the name or names is optional.

host_indicator_name

The indicator variable.

The colon character preceding the name is mandatory.

descriptor_area

An 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.

Usage Notes

- You should define an SQLDA.
- You must previously declare the cursor identified by *cursor_name*.
- The cursor identified by *cursor_name* must be closed.
- Once the cursor is open, the system executes the associated static or dynamic SQL statement or statements. The system then creates the result spool file and positions the cursor before the first row of the spool file.
- OPEN processing returns a 0 in the SQLCODE field of the SQLCA and '00000' to SQLSTATE, unless a failure (implicit rollback) occurs. For an SQLCODE of 0, the system places the activity count in the third SQLERRD element of the SQLCA structure.
- If the cursor is updatable, or a C or COBOL application program contains a REWIND or POSITION TO STATEMENT request for the cursor, execute the OPEN statement with KEEPRESP; otherwise, execute it with NOKEEPRESP. For PL/I applications, KEEPRESP is the default.

- You cannot execute OPEN as a dynamic SQL statement.
- No more than 16 cursors can be open at one time because the processing of non-cursor-related statements is increasingly affected for the worse as more cursors are opened.

If an application has 16 cursors open, no other request can be issued until one or more cursors are closed.

- 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 you declared before the OPEN statement, to be used as an input variable.

You can use a client structure to identify the input variables.

The number of variables you specify must be the same as the number of parameter markers (the question mark character) in the identified statement.

The n^{th} variable corresponds to the n^{th} marker.

Use of the colon character prefix for *host_variable_name* is optional.

- *descriptor_area* identifies an input SQLDA structure, previously defined by the application, that contains necessary information about the input variable set.

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 n^{th} variable described by the SQLDS corresponds to the n^{th} marker.

Related Information

- Cursors referenced in the ALLOCATE statement, see [ALLOCATE](#).
- Creating casts and using the AS ASSIGNMENT option, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- *descriptor_area*, see *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446.
- OPEN (Embedded SQL Form), see [CLOSE](#).

OPEN (Stored Procedures Form)

Opens a declared cursor in a stored procedure and executes the SQL statement specified in its declaration.

ANSI Compliance

ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Stored procedures only.

Syntax

```
OPEN cursor_name [ USING { SQL_identifier | SQL_parameter } [, ...] ] ;
```

Syntax Elements

cursor_name

The name of the cursor to be opened.

USING

Variables used as input to the SQL statement specified by *cursor_name* that must be declared before the OPEN statement.

SQL_identifier

A valid SQL identifier.

SQL_parameter

An SQL parameter.

Usage Notes

- Returning a Result Set

The OPEN statement opens a result set cursor and executes the static or dynamic SELECT statement, which produces the result set. The system creates the result spool file, and positions the cursor before the first row of the spool file.

- General Rules

- You must previously declare the cursor identified by *cursor_name*.
- The cursor identified by *cursor_name* must not already be open.

- Rules for USING Clause

- The number of variables specified must be the same as the number of parameter markers (the question mark character) in the identified statement. The n^{th} variable corresponds to the n^{th} marker.
- You cannot execute OPEN as a dynamic SQL statement.
- You can only use the USING clause in an OPEN cursor when the cursor is a dynamic SQL form.
- No more than 15 cursors can be open at one time. If an application has 15 cursors open, no other request can be issued until one or more cursors are closed.

Example: Using the OPEN Statement

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;
```

Example: Using the OPEN Cursor Statement and the USING Clause

In this example, the OPEN cursor statement is extended to allow a USING clause.

```
CREATE PROCEDURE abc (IN data1v VARCHAR(10), IN data2v VARCHAR(10) )
  DYNAMIC RESULT SETS 1
BEGIN
  DECLARE sql_stmt1 VARCHAR(100);
  DECLARE sales DECIMAL(8,2);
  DECLARE item INTEGER;
  DECLARE cstmt CURSOR WITH RETURN ONLY FOR stmt1;
  SET sql_stmt1 = 'SELECT T1.item, T1.sales FROM T1 WHERE ?'
                  '= store_name AND ? = region;';
  PREPARE stmt1 FROM sql_stmt1;
  OPEN cstmt USING data1v, data2v;
END;
```

Related Information

- Returning a result set, see [Returning Result Sets from a Stored Procedure](#).
- OPEN (Stored Procedures Form), see [CLOSE](#).

POSITION

Positions a cursor to the beginning of the next statement or to the results of a specified statement.

ANSI Compliance

POSITION is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
POSITION cursor_name
  [ TO
    { NEXT |
      [STATEMENT] { statement_number | [:] numeric_variable }
    }
  ]
```

Syntax Elements***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 a colon character is optional.

Usage Notes

The cursor is repositioned before the first result row (if any) of the statement specified and SQLSTATE, 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 specified statement.

If the specified statement number does not exist (for example, TO STATEMENT 3 is coded, but the cursor controls only two statements), the following runtime exception occurs, leaving the position of the cursor undefined:

- SQLCODE is set to -501
- SQLSTATE is set to '24501'

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 valid with any cursor except an insertion cursor.

The statement set found by the cursor is *not* re-executed, but the cursor position in the spool file is adjusted accordingly.

You cannot execute POSITION as a dynamic SQL statement.

Related Information

- POSITION being valid with any cursor except an insertion cursor, see [DECLARE CURSOR](#), [DECLARE CURSOR \(Macro Form\)](#), [DECLARE CURSOR \(Request Form\)](#) and [DECLARE CURSOR \(Selection Form\)](#)
- POSITION, see [REWIND](#).

PREPARE

Prepares the dynamic DECLARE CURSOR statement to allow the creation of different result sets. Allows dynamic parameter markers.

ANSI Compliance

PREPARE is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Stored procedure only.

Syntax

```
PREPARE statement_name FROM { 'statement_string' | statement_string_variable } ;
```

Syntax Elements

statement_name

the same identifier as *statement_name* in a DECLARE CURSOR statement.

statement_string

the SQL text that is to be executed dynamically.

statement_string must be enclosed within apostrophes.

statement_string_variable

the name of an SQL local variable, or an SQL parameter or string variable, that contains the SQL text string to be executed dynamically.

This element should be a CHAR/VARCHAR variable less than 32000 characters.

Usage Notes

- The Parser checks the syntax of the PREPARE statement. If there is a syntax error, the system returns a syntax exception.
- You cannot execute PREPARE as a dynamic SQL statement.
- The statement must be a dynamic cursor SELECT statement. If this is not the case, the system returns '07005' dynamic SQL error, prepared statement not a cursor specification.
- The dynamic SQL statement text can be as long as 64 Kbytes (including SQL text, USING data, and parcel overhead).
- You cannot specify multistatement requests.
- The dynamic SQL statement can include parameter markers or placeholder tokens (the question mark), where any literal reference, particularly an SQL variable, is legal except in the select list.
- The USING clause of the OPEN statement supplies values to the statement.

Example: Using PREPARE

```
CREATE PROCEDURE abc (IN data1v VARCHAR(10), IN data2v VARCHAR(10) )
  DYNAMIC RESULT SETS 1
BEGIN
  DECLARE sql_stmt1 VARCHAR(100);
  DECLARE sales DECIMAL(8,2);
  DECLARE item INTEGER;
  DECLARE cstmt CURSOR WITH RETURN ONLY FOR stmt1;
  SET sql_stmt1 = 'SELECT  T1.item, T1.sales FROM T1 WHERE' data1v
                  || '= store_name AND '    || data2v || '= region;';
  PREPARE stmt1 FROM sql_stmt1;
  OPEN cstmt;
END;
```

Example: Using PREPARE with Parameter Markers

In this example, the PREPARE statement is written using parameter markers:

```
SET sql_stmt1 = 'SELECT  T1.item, T1.sales FROM T1 WHERE ?'
               '= store_name AND ? = region;';
```

```
PREPARE stmt1 FROM sql_stmt1;
OPEN cstmt USING data1v, data2v;
```

Related Information

[PREPARE \(Dynamic\)](#).

REWIND

Positions or repositions a cursor to the beginning of the results of its first or only statement.

ANSI Compliance

REWIND is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
REWIND cursor_name
```

Syntax Elements

cursor_name

The name of an open cursor other than an Insertion cursor.

Usage Notes

The statement `REWIND cursor_name` is equivalent to the statement `POSITION cursor_name TO STATEMENT 1.`

Related Information

[POSITION](#).

SELECT AND CONSUME ... INTO

Selects data from the row with the oldest insertion timestamp in the specified queue table, deletes the row from the queue 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.

ANSI Compliance

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

Required Privileges

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

Invocation

Executable.

Stored procedures and embedded SQL.

Stored Procedure Syntax

```
{ SELECT | SET } AND CONSUME TOP 1 select_list
  INTO into_spec [,...] FROM queue_table_name
```

into_spec

```
[:] { local_variable_name | parameter_name }
```

Embedded SQL Syntax

```
{ SELECT | SET } AND CONSUME TOP 1 select_list
  INTO into_spec [,...] FROM queue_table_name
```

into_spec

```
[:] host_variable_name [ [INDICATOR] :host_indicator_name ]
```

Syntax Elements

select_list

An ASTERISK character (*) or a comma-separated list of valid SQL expressions.

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

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

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.

queue_table_name

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

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.

host_variable_name

The name of the host variable into which the selected data is to be placed.

host_indicator_name

The name of the host indicator variable.

Usage Notes

- Attributes of a Queue Table

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

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

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

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

- Using a Colon Character in Embedded SQL

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

- Rules for Embedded SQL

The same rules that apply to SELECT ... INTO apply to SELECT AND CONSUME ... INTO.

Related Information

- *queue_table_name*, see the information about CREATE TABLE (Queue Table Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- Usage notes, information on transaction processing, locks, and restrictions, see the information about SELECT AND CONSUME in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- The rules that apply to SELECT ... INTO and SELECT AND CONSUME ... INTO, see the rules for embedded SQL in [SELECT ... INTO](#).

SELECT ... INTO

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.

ANSI Compliance

SELECT ... INTO is ANSI/ISO SQL:2011-compliant.

Required Privileges

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.

For stored procedures, the local variables and parameters in the select and INTO lists can be UDTs, except VARIANT_TYPE UDTs.

You must have the UDTUSAGE privilege on any local variable or parameter that has a UDT data type.

Invocation

Executable.

Stored procedures and embedded SQL.

Stored Procedure Syntax

```
[ with_[recursive]_modifier ] { SELECT | SET }
[ ALL | DISTINCT ] select_list
INTO { local_variable_name | parameter_name } [, ...]
[ from_clause ]
[ where_clause ]
```

Embedded SQL Syntax

```
[ with_[recursive]_modifier ] { SELECT | SET }
  select_list
  INTO into_spec [ , ... ]
  [ from_clause ]
  [ where_clause ]
```

Syntax Elements

into_spec

```
[ : ] host_variable_name [ [ INDICATOR ] :host_indicator_name ]
```

with_[recursive]_modifier

A recursive query that provides a way to search a table using iterative self-join and set operations if WITH RECURSIVE is used.

The nonrecursive WITH request modifier is similar to a derived table.

select_list

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

The select list can contain instances of the DEFAULT function, but must not contain aggregate or ordered analytical functions.

Stored procedures only:

Vantage performs implicit conversions for DateTime data types. For all other data types, the select list data types must match the INTO clause target list data types.

If the select list data types do not match the INTO clause target list data types, you can specify an explicit CAST to the target type to enable the operation to succeed.

Columns specified in the select and INTO lists can have UDT data types, except for the VARIANT_TYPE UDT data type. The system automatically applies any implicit conversions defined for the UDT if they exist.

The system applies implicit casting of the select list data types from UDTs to predefined data types or from predefined types to UDTs only if a CAST to the target type exists and was created with the AS ASSIGNMENT option specified.

host_variable_name

The name of the host variable into which the selected data is to be placed.

host_indicator_name

The name of the host indicator variable.

from_clause

A clause listing the tables or views referenced by the SELECT.

where_clause

A clause narrowing a SELECT to those rows that satisfy a conditional expression that it specifies.

The WHERE clause can contain the DEFAULT function as a component of its predicate.

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.

Stored procedures only: A local variable can have a UDT type, except for the VARIANT_TYPE UDT data type.

You must have the UDTUSAGE privilege on any UDT used as a local variable.

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.

Stored procedures only: A parameter can have a UDT type, except for the VARIANT_TYPE UDT data type.

You must have the UDTUSAGE privilege on any UDT used as a parameter.

Usage Notes

- Rules for Using a Colon Character

Following are the rules for using a colon character in embedded SQL:

- Pad characters preceding and following a colon character are optional.
- A prepending colon character for *host_variable_name* is optional.
- A prepending colon character *must* precede a *host_indicator_name*.

Following are the rules for using a colon character in stored procedures:

- A prepending colon character preceding a *local_variable_name* is optional.
- A prepending colon character preceding a *param_name* is optional.

- Rules for 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:

- `WITH [RECURSIVE]` request modifier
- `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.

For stored procedures, you must also have the `UDTUSAGE` privilege on any UDT used as the data type for any column in the select and `INTO` lists.

`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. Vantage performs implicit conversions for `DateTime` data types when the data type of the local variable or parameter differs from the corresponding column data type.

For stored procedures, you must have the `UDTUSAGE` privilege on any UDT used as a local variable or parameter.

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

The `SELECT ... INTO` statement is normally expected to return at most one row. One of the following actions is taken after executing the statement:

IF <code>SELECT ... INTO</code> returns ...	The stored procedure status variables show these values ...	Which mean ...
more than one row	<code>SQLCODE = 7627</code> <code>SQLSTATE = '21000'</code> <code>ACTIVITY_COUNT</code> = number of rows found.	an exception condition (a failure in Teradata session mode and error in ANSI session 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, <i>without</i> an execution warning	<code>SQLCODE = 7632</code> <code>SQLSTATE = '02000'</code> <code>ACTIVITY_COUNT = 0</code>	a completion condition other than successful completion. A specific condition handler can be specified to handle this completion condition. The

IF SELECT ... INTO returns ...	The stored procedure status variables show these values ...	Which mean ...
		values of local variables and parameters do not change.
no rows, <i>with</i> 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 <i>without</i> 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 <i>with</i> 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.

- Rules for Embedded SQL

UDTs are not specifically supported.

Note, however, that UDTs for which `tosql` and `fromsql` transforms have been defined can be externally referenced by means of their transform target data types. As a result, embedded SQL applications can use SQL statements that reference UDTs provided that the UDTs have a defined `tosql` or `fromsql` transform as appropriate.

Additionally, the application must send and receive UDT data in the form of its external (non-UDT) data type.

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 `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.

You cannot execute SELECT ... INTO as a dynamic SQL statement.

SELECT ... INTO supports browse mode SELECT operations for queue tables.

- Rules for Using the DEFAULT Function With SELECT Statements
 - The DEFAULT function takes a single argument that identifies a relation column by name. The function evaluates to a value equal to the current default value for the column. For cases where the default value of the column is specified as a current built-in system function, the DEFAULT function evaluates to the current value of system variables at the time the request is executed.
 - The resulting data type of the DEFAULT function is the data type of the constant or built-in function specified as the default, unless the default is NULL. If the default is NULL, the resulting data type is the data type of the column or expression for which the default is being requested.
 - The DEFAULT function has two forms. It can be specified as DEFAULT or DEFAULT (*column_name*). When no column name is specified, the system derives the column based on context. If the column context cannot be derived, the request aborts and an error is returned to the requestor.
 - You can specify a DEFAULT function with a column name in the select list of a SELECT statement. The DEFAULT function evaluates to the default value of the specified column.
 - You cannot specify a DEFAULT function without a column name in the expression list. The system aborts the request and returns an error to the requestor.

- If you specify a SELECT statement that does not also specify a FROM clause, the system always returns a single row with the default value of the column, regardless of how many rows are in the table.

This is similar to the existing TYPE function.

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

Example: Recursive Query

The following example shows a recursive query used inside a client application:

```
EXEC SQL
    WITH RECURSIVE Reachable_From (Source, Destin, mycount)AS
    (
        SELECT Root.Source, Root.Destin, 0 as mycount
        FROM Flights Root
        WHERE Root.Source = 'Paris'
    UNION ALL
        SELECT in1.Source, out1.Destin, in1.mycount + 1
        FROM Reachable_From in1, Flights out1
        WHERE in1.Destin = out1.Source
        AND in1.mycount <= 100
    )
    SELECT Source, Destin
    INTO :intosource INDICATOR :indvar1
    :intodestin INDICATOR: indvar2
    FROM Reachable_From;
END-EXEC
```

In this example, the host variables *intosource* and *intodestin* and indicator variables *indvar1* and *indvar2* are being used in the final SELECT of the recursive query. These variables cannot be used inside the recursive query definition.

Related Information

- *with_[recursive]_* modifier, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- Implicit data type conversions and the CAST function, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- See the CALL statement in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 for more details on authorization required.
- Creating a cast and using the AS ASSIGNMENT option, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- *from_clause*, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

- Performing implicit conversions for DateTime data, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- SELECT ... INTO supporting browse mode SELECT operations for queue tables, see the information about CREATE TABLE (Queue Table Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- the DEFAULT function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

UPDATE (Positioned Form)

Updates the most current fetched cursor row.

ANSI Compliance

The positioned form of UPDATE is ANSI/ISO SQL:2011-compliant.

Required Privileges

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* 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.

Invocation

Executable.

Stored procedures and embedded SQL.

Syntax

```
{ UPDATE | UPD } table_name [ alias_name ]
  SET set_spec [, ...]
  WHERE CURRENT OF cursor_name [;]
```

Syntax Elements

set_spec

```
column_name = expression
```

table_name

The name of the table to be updated.

alias_name

An alias for the table name.

Correlation names 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.

Usage Notes

- UPDATE With Correlated Subqueries

See the information about correlated subqueries and the UPDATE statement in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for information about using correlated subqueries with UPDATE statements.

- Large Objects and UPDATE

The behavior of truncated LOB updates differs in ANSI and Teradata session modes. The following table explains the differences in truncation behavior.

In this session mode ...	When non-pad bytes are truncated on insertion ...
ANSI	an exception condition is raised.

In this session mode ...	When non-pad bytes are truncated on insertion ...
	The UPDATE fails.
Teradata	no exception condition is raised. The UPDATE succeeds: the truncated LOB is stored.

- Update of GENERATED ALWAYS Identity Columns and PARTITION

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.

- Support of Mutator SET Clauses

Because UDTs are not supported in embedded SQL, the mutator SET clause is not supported for the positioned form of UPDATE.

- Rules for UPDATE in ANSI Session 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 value that is not null between 1 and 65535.

- Rules for Using the DEFAULT Function

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

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

- The DEFAULT function has two forms. It can be specified as DEFAULT or DEFAULT (*column_name*). When no column name is specified, the system derives the column based

on context. If the column context cannot be derived, the request aborts and an error is returned to the requestor.

- You can specify a DEFAULT function without a column name argument as the expression in the SET clause. The column name for the DEFAULT function is the column specified as the *column_name*. The DEFAULT function evaluates to the default value of the column specified as *column_name*.
- You cannot specify a DEFAULT function without a column name argument as part of the expression. Instead, it must be specified by itself. This rule is defined by the ANSI/ISO SQL:2011 specification.
- You can specify a DEFAULT function with a column name argument in the source expression. The DEFAULT function evaluates to the default value of the column specified as the input argument to the DEFAULT function.

For example, DEFAULT(col2) evaluates to the default value of col2. This is a Teradata extension to the ANSI/ISO SQL:2011 specification.

- You can specify a DEFAULT function with a column name argument anywhere in an update expression. This is a Teradata extension to the ANSI/ISO SQL:2011 specification.
- When no explicit default value has been defined for a column, the DEFAULT function evaluates to null when that column is specified as its argument.

Example: Using the Cursor to Update the Table

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;
```

Related Information

- The mutator SET clause, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for details about mutator SET clauses.
- General information about the DEFAULT function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Result Code Variables

This section describes a set of result code variables, also known as status parameters, shared by stored procedures and embedded SQL applications.

Note:

Result code variable information for stored procedures, including initial values and restrictions, is included in [Result Code Variables in Stored Procedures](#).

SQLSTATE

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 (error or warning code and the condition of an SQL statement, including control statements).

ANSI Compliance

SQLSTATE is ANSI/ISO SQL:2011-compliant. You should use it instead of SQLCODE for any new applications you develop.

The SQLSTATE status variable used by stored procedure programs is non-ANSI/ISO SQL:2011 standard.

Either SQLSTATE or SQLCODE must be declared for all embedded SQL applications written for ANSI session mode.

SQLSTATE is not valid for embedded SQL applications running in Teradata session mode.

Where SQLSTATE Receives Error Codes

- CLI/TDP
- Database
- Preprocessor2 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](#) lists the ANSI/ISO SQL:2011-defined SQLSTATE classes.

Subclass values can be any numeric or simple uppercase Latin character string.

SQLSTATE Type Definition for Embedded SQL

Preprocessor2 requires the following SQLSTATE data type definitions for the indicated client application language.

Client Language	Data Type Definition
<ul style="list-style-type: none"> • COBOL • PL/I 	CHARACTER(5)
C	CHARACTER(6) The sixth character is always a null terminator.

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 Database Error Codes Map to SQLSTATE

Unless otherwise specified, CLI/TDP, preprocessor runtime, and database 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 database errors are class T1 through class T9.
The differentiating digit in the class number corresponds to the first digit of the database error code.
The subclass contains the remaining 3-digit database error code.
For example, error 3776 (unterminated comment) produces class T3 and subclass 776.

SQLCODE to SQLSTATE Exception Mapping

Some SQLCODE values are generated by errors not originating within CLI, TDP, or Teradata SQL.

SQLSTATE Usage Constraints in Stored Procedures

The following usages of SQLSTATE are valid within a stored procedure:

- As the operand of a SET statement.
For example, the following statements are valid:

```
SET hErrorMessage = 'SQLSTATE' || sqlstate;
SET hSQLSTATE = SQLSTATE;
```

- 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.

Related Information

- The complete set of SQLSTATE class definitions and mappings for embedded SQL and stored procedure applications, see [SQLSTATE Mappings](#).
- Database error codes, see *Teradata Vantage™ - Database Messages*, B035-1096.
- Exception mappings for the codes, see [SQL Communications Area \(SQLCA\)](#).

SQLCODE

In ANSI session mode, SQLCODE is a host variable (embedded SQL) or status variable (stored procedures) that receives SQL statement status information (error or warning code and the condition of an SQL statement, including control statements). The status codes permit an application program to test whether an executable embedded SQL statement completed successfully or not.

In Teradata session mode (for embedded SQL), SQLCODE is a field within SQLCA.

ANSI Compliance

SQLCODE is not ANSI/ISO SQL:2011-compliant. SQLCODE was deprecated in the ANSI/ISO SQL-92 standard and is not defined in the SQL:2011 standard. The ANSI/ISO SQL committee recommends that new applications be written using SQLSTATE in place of SQLCODE.

SQLCODE is required for all embedded SQL applications written for ANSI session 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/ISO SQL-99 and SQL:2011 standards, nor is SQLCA. Stored procedures do not use SQLCA.

SQLCODE in ANSI Session 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 Preprocessor2 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 Session Mode

The SQLCODE field within SQLCA communicates the result of executing an SQL statement to an embedded SQL application program. Stored procedures do not use SQLCA.

When the Preprocessor2 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 executed 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 system warnings. For example, an SQLCODE value of +100 indicates one of the following results: <ul style="list-style-type: none"> • 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.

When to Test SQLCODE

Test SQLCODE after each execution of an 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.

SQLCODE Testing Example

Consider an application that creates a temporary table and then populates it using an INSERT ... SELECT statement.

You would write your application code to execute an SQLCODE check immediately after executing 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 execute some appropriate action to prevent executing 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 Database Error Messages Map to SQLCODE Values

For information on mapping SQLCODE values to database error message numbers, see [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.

Related Information

- SQLCODE, see [SQL Communications Area \(SQLCA\)](#).
- SQLSTATE, see [SQLSTATE](#).
- For embedded SQL applications, see [WHENEVER](#) for information about condition handling.
- For stored procedure applications, see [Completion, Exception, and User-defined Condition Handlers](#) for information about condition handling.

ACTIVITY_COUNT

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.

ANSI Compliance

ACTIVITY_COUNT is a Teradata extension to the ANSI/ISO SQL:2011 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.

There can be 2 different limits of activity count, depending on the client application and the server's capability:

- 4-byte limit that is $2^{32}-1$ rows
- 8-byte limit that is $2^{64}-1$ rows

If the client application is coded to request for the 8-byte activity count and the server is capable of returning 8-byte activity count, you will no longer see the numeric overflow warning. This means the activity count is always a true activity count (not activity count modulo 2^{32}). If, however, the client application is not coded to request for 8-byte activity count or the server is not capable of returning 8-byte activity count, you will still see numeric overflow behavior.

When to Test **ACTIVITY_COUNT**

Test **ACTIVITY_COUNT** after each execution of an 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.

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.
- If the activity count for a query exceeds a limit of $2^{32}-1$ rows, the system returns the true activity count modulo 2^{32} along with the following warning message:

```
Numeric overflow has occurred internally. The number of rows returned is
actual number of rows returned, modulo 2^32.
```

To determine the actual activity count in this situation, you must add the modulo 2^{32} value returned to 2^{32} as follows:

True activity count = returned_value + 2^{32}

This is true for both SQL stored procedure and embedded SQL applications.

Note:

There can be 2 different limits of activity count, depending on the client application and the server's capability:

- 4-byte limit that is $2^{32}-1$ rows
 - 8-byte limit that is $2^{64}-1$ rows
-

Related Information

- Activity Count, see [SQL Communications Area \(SQLCA\)](#)
- For embedded SQL applications, see [WHENEVER](#) for information about condition handling.
- For stored procedure applications, see [Completion, Exception, and User-defined Condition Handlers](#) for information about condition handling.

Result Code Variables in Stored Procedures

Result code variables in an SQL statement, other than a control statement, must be prefixed with a colon character (:) when used in a stored procedure.

Initial Values of Result Code Variables

Result code variables are mapped to the database 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.

Result Code 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 execution 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 result code variables are set to appropriate values for SQL statements other than control statements within the stored procedure. The result code variables do not change for control statements.

Restrictions on Result Code Variables in Stored Procedures

The following constraints apply to result code variables in a stored procedure:

The result code 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 result code variables.

You cannot specify result code variables 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

SQL Stored Procedures

This section describes the SQL form of stored procedures.

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	<ul style="list-style-type: none"> • database • user
ALTER PROCEDURE DROP PROCEDURE EXECUTE PROCEDURE	<ul style="list-style-type: none"> • 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.

Vantage does not grant this privilege automatically to the owner of the stored procedure when the owner and creator are not the same.

The immediate owner of a stored procedure is the user or database space where the stored procedure is created. The creator is the user who creates the stored procedure in any database.

Related Information

- Stored procedures, see [Stored Procedure Overview](#).
- External stored procedures, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- SQL forms of the GRANT and REVOKE statements and the ALTER PROCEDURE, CREATE PROCEDURE, DROP PROCEDURE, and EXECUTE PROCEDURE privileges, see *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

Checking Privileges for Stored Procedures

To create or execute a stored procedure, you must have the appropriate privileges to execute the SQL statements in the procedure and privileges to the database objects referenced in the stored procedure body. Vantage also checks for the appropriate CREATE and DROP privileges during the creation of the procedure, and the EXECUTE privilege during the execution of the procedure.

You can specify how privilege checking is handled by defining the SQL SECURITY clause in the CREATE/REPLACE PROCEDURE statement. When the stored procedure is compiled or executed, Vantage checks for the required privileges based on the following options of the SQL SECURITY clause:

- CREATOR
- DEFINER
- INVOKER
- OWNER

Note:

You must have the CREATE OWNER PROCEDURE privilege to specify the OWNER option if the creator is different from the immediate owner of the stored procedure.

The SQL SECURITY option determines which of the following privileges are checked when the stored procedure is compiled or executed:

- the privileges of the user that created the stored procedure (no matter where the stored procedure resides)
- the privileges of the current user that invoked the stored procedure
- the privileges of the immediate owner of the stored procedure (the user or database space where the stored procedure resides)
- the privileges of the creator and the owner of the stored procedure

The SQL SECURITY option also determines the default database used to implicitly qualify any unqualified object references within the SQL statements in the procedure body.

The SQL SECURITY clause is optional. If you do not include the clause, Vantage uses the SQL SECURITY DEFINER option as the default.

IF the SQL SECURITY option is...	THEN...
CREATOR	<ul style="list-style-type: none"> • the privileges of the user that created the stored procedure are checked, no matter where the stored procedure resides. • the default database used to implicitly qualify any unqualified object references within the SQL statements in the procedure body is that of the creator.
DEFINER	<p>for stored procedures with dynamic SQL, the following privileges are checked:</p> <ul style="list-style-type: none"> • If the creator is different from the immediate owner, then both the creator and owner privileges are checked upon execution of the procedure. • If the creator is the same as the immediate owner, then either the creator or the owner privileges are checked upon execution of the procedure. <p>For stored procedures with static SQL, the following privileges are checked:</p> <ul style="list-style-type: none"> • If the creator is different from the immediate owner, then both the creator and owner privileges are checked upon compilation and execution of the procedure. • If the creator is the same as the immediate owner, then either the creator or the owner privileges are checked upon compilation and execution of the procedure.

IF the SQL SECURITY option is...	THEN...
	The default database used to implicitly qualify any unqualified object references within the SQL statements in the procedure body is that of the creator.
INVOKER	<ul style="list-style-type: none"> the privileges of the user creating the stored procedure are checked when the procedure is created. the privileges of the user that called the stored procedure are checked upon execution of the procedure. the default database used to implicitly qualify any unqualified object references within the SQL statements in the procedure body is that of the current user.
OWNER	<ul style="list-style-type: none"> the privileges of the immediate owner of the stored procedure are checked. the default database used to implicitly qualify any unqualified object references within the SQL statements in the procedure body is that of the immediate owner. <p>Note: You must have the CREATE OWNER PROCEDURE privilege to specify this option if the creator is different from the immediate owner of the procedure.</p>
Not specified	the DEFINER SQL SECURITY option is used as the default.

If you are accessing Vantage through a proxy connection, privilege checking for stored procedures is still done based on the SQL SECURITY clause. The privileges of the proxy user are used for checking the required privileges of referenced SQL statements and objects only if the SQL SECURITY INVOKER option is specified.

Because stored procedures use the SQL SECURITY option to determine the default database, the default database is not updated if a proxy user is set in a stored procedure. If the query band with the proxy user remains when exiting the stored procedure, the session is set to the default database for the proxy user.

If the stored procedure includes a SET QUERY_BAND statement that sets a proxy user, Vantage validates the CONNECT THROUGH privilege of the trusted user when the stored procedure is executed.

If the CONNECT THROUGH privilege of the trusted user includes the WITH TRUST-ONLY option, then all SET QUERY_BAND statements that set, change, or remove a proxy user or proxy role must be performed from a trusted request. A SET QUERY_BAND statement in an

Related Information

- SQL SECURITY clause, see the information about CREATE PROCEDURE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- Granting the CONNECT THROUGH privilege to a trusted user, see *Teradata Vantage™ - SQL Data Control Language*, B035-1149.
- Setting a proxy user with SET QUERY_BAND, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Rules for Using SQL Statements in Stored Procedures

The rules governing the use of any statement within a stored procedure, including static and dynamic SQL statements, control statements, condition handler, cursor declaration, condition declaration, and local declaration statements, depend on the option specified in the SQL SECURITY clause.

The following rules apply to the use of statements within a stored procedure:

If any SQL statement specified in the stored procedure references a missing database object and the SQL security clause is CREATOR, OWNER, or INVOKER, an SPL compilation warning is reported during the procedure creation.

If any SQL statement specified in the stored procedure references a missing database object and SQL security clause is DEFINER and:

- the creator and owner are the same, an SPL compilation warning is reported during the stored procedure creation.
- the creator and owner are not the same, an error is reported and the stored procedure is not created.

If the referenced object does not exist when the stored procedure is executed, a runtime exception is reported.

If the cursor SELECT statement references a missing database object, an SPL compilation error is reported.

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 user does not have the required privileges on the objects referenced in the stored procedure, appropriate warnings or errors are reported during stored procedure creation. Vantage checks the privileges based on the definition of the SQL SECURITY clause.

If the required privileges do not exist when the stored procedure is executed, a runtime exception is reported.

If the creator does not have the required privileges on the objects referenced in the cursor SELECT statement, an SPL compilation error is reported.

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. This is true even if you are accessing Vantage through a proxy connection. (Note that volatile tables are not permanent objects.)

Other users executing the stored procedure do not get any automatic rights on the newly created objects. The immediate owner can explicitly grant privileges on the newly created objects to other users.

- If a database object in an SQL statement is not explicitly qualified by a database name, the default database used to implicitly qualify the object depends on the SQL SECURITY option.

If a DDL statement is creating the database object, the qualifying database (either implicit or explicit) is the immediate owner of the object created.

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	<ul style="list-style-type: none"> a compilation error is reported. the procedure is not created.
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 default database of the stored procedure. The default database depends on the SQL SECURITY option.

IF ...	THEN ...
no such table exists in the compile-time default database	<p>the system looks for a volatile table with the same name in the login database for the user.</p> <ul style="list-style-type: none"> If the volatile table exists, then it 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 <i>also exists</i> 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.

Related Information

For more information about SQL SECURITY, see [Checking Privileges for Stored Procedures](#).

Executing a Stored Procedure

To execute a stored procedure, use the SQL CALL statement.

The 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 (BEGIN TRANSACTION) and ET (END TRANSACTION) 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.

Data Type Codes

The database returns a specific set of CLIV2 data type codes to the calling application when the CALL statement is submitted.

Stored Procedure Parameters

The data type codes returned when the CALL statement is submitted include a 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.

Related Information

- Stored procedure execution, see the description of the CALL statement in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- Data type codes possible with a CALL statement, see the information about the DataInfo parcel in *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418 or *Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems*, B035-2417.
- Memory considerations for INOUT parameters, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- Executing a stored procedure from embedded SQL, see the CALL statement in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Recompiling a Stored Procedure

You must recompile your stored procedures whenever you upgrade or migrate to or across a major release. Use the ALTER PROCEDURE statement to recompile a stored procedure.

AT TIME ZONE Option for SQL Procedures

When you create an SQL procedure, Vantage stores the current session time zone for the procedure along with its definition to enable the SQL control language elements and the SQL statements in the procedure to execute in a consistent time zone and produce consistent results. However, time or timestamp data passed as an input parameter to the procedure still use the runtime session time zone rather than the creation time zone for the procedure.

The AT TIME ZONE option enables you to reset the time zone for all of the SQL elements of SQL procedures when you recompile a procedure. Vantage then stores the newly specified time zone as the creation time zone for the procedure.

You can only specify AT TIME ZONE with the COMPILE [ONLY] option, and it must follow the COMPILE [ONLY] specification. If it does not, Vantage aborts the request and returns an error to the requestor.

Limitation on the Use of TimeZone Strings

When using standard Teradata system time zone strings, no time zone rules are enforced for the years 1986 and before, for example, DST shifts. Valid years for Teradata standard time zone strings are 1987 through 9999.

If Teradata standard time zone strings do not meet your requirements, you can add a new custom time zone string or modify an existing string by modifying or adding new rules to GetTimeZoneDisplacement.

For more information about the SQL form of ALTER PROCEDURE, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Restrictions on 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.

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 executed in Teradata transaction mode, and vice versa. You can, however, execute the stored procedure after recreating it in the new session mode using REPLACE PROCEDURE.

A stored procedure created on one platform cannot be executed on another platform. But this limitation can be overcome by recompiling a stored procedure using the ALTER PROCEDURE statement.

If a stored procedure is the only statement in a macro, you can execute a procedure from the macro.

Stored procedures do not support the following:

- EXPLAIN and USING request modifiers within a stored procedure
- EXECUTE macro statement

- WITH clause within a stored procedure.

Stored procedures, as well as macros, do not support the following query logging statements:

- BEGIN QUERY LOGGING
- END QUERY LOGGING
- FLUSH QUERY LOGGING
- REPLACE QUERY LOGGING

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.

The queue table form of CREATE TABLE cannot be executed in a stored procedure. All other forms of the CREATE TABLE statement are valid.

DDL and DCL statements for Row Level Security (RLS) administration is not be allowed to be defined in a stored procedure. RLS provides the capability to control data access based on security policies, data sensitivity classifications, and security credentials assigned to users. Supported RLS constraint checks will be executed if a user does not have override privileges.

When you use CURRENT_TIME or CURRENT_TIMESTAMP in a stored procedure and you have manually changed DBS Control General Fields 16 (System TimeZone Hour) and 17 (System TimeZone Minute), the stored procedure must be recompiled. It is not necessary to set the TimeZoneString in tdlocaledef to change the time zone automatically.

Related Information

- REPLACE PROCEDURE or ALTER PROCEDURE, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- CREATE TABLE (Queue Table Form), see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146

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.

The following rules apply to specific names in stored procedures:

This name ...	Must be unique in ...
correlation or column	<p>a FOR iteration statement, or a DECLARE CURSOR statement.</p> <p>The same correlation or column name can be reused in nested or non-nested FOR statements within a stored procedure.</p> <p>A correlation or column name can be the same as the for-loop variable and cursor names in a FOR statement.</p>

This name ...	Must be unique in ...
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.

Literals

All Vantage-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 Vantage-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.

A local variable can have any valid data type.

If a local variable is specified in an SQL statement other than a control statement, it need not be prefixed with a colon character (:). The colon prefixed to a local variable is still supported, but is not recommended.

If a local variable is not prefixed with a colon character, the variable name should not be the same as a column name.

If an SQL statement contains an identifier that is the same as an SQL variable name and a column name, Teradata interprets the identifier as a column name. To prevent this, the SQL variable identifier that is a column name should be qualified with the compound statement 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 can be qualified with the label of the corresponding compound statement in which the variable is declared. This helps avoid conflicts that might be caused by reused local variables in nested compound statements.

Parameters

A stored procedure can have up to 256 parameters of any Vantage-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 need not be prefixed with a colon character (:). The colon character prefix to a parameter is supported, but not recommended.

If a parameter is not prefixed with a colon character, it should not be the same as a column name.

If an SQL statement contains an identifier that is the same as an SQL parameter and an column name, the database interprets it as a column name. To prevent this, you should qualify the column name with the compound statement 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

Rules for IN, OUT, and INOUT Parameters

Parameter	Allowed In	Not Allowed In
IN	Source specification of an SQL statement	Target specification of an SQL statement.

Parameter	Allowed In	Not Allowed In
OUT	Target specification of an SQL statement	Source specification of an SQL statement.
INOUT	Source and target specifications of an SQL statement.	—

Parameters can have any valid data type.

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 executing the stored procedure.
- You can read the output value from the same parameter after execution of the procedure.

The maximum data size of all input and all output parameters in a CREATE/REPLACE PROCEDURE is 1 MB.

When you invoke a stored procedure, the IN constants assume the data type of the value specified unless overridden in the CALL statement.

The data type for an INOUT constant argument is governed by the data type of the value passed in, not what is defined. If the data type of the value passed in is smaller than the data type defined in the CREATE/REPLACE PROCEDURE statement, and the stored procedure returns a value larger than the maximum value of the data type for the value passed in, the system returns an overflow error.

For example, consider a stored procedure that defines an INTEGER INOUT parameter. If you call the procedure with a constant input value that fits into a SMALLINT, the system returns an overflow error if the output value is larger than 32767, the maximum value of a SMALLINT.

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 correlation or column names. If not qualified with the for-loop variable name, the correlation or column 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

See [SQL Cursors](#) for rules and guidelines governing the use of cursors in stored procedures.

See [DECLARE CURSOR \(Stored Procedures Form\)](#) and [FOR](#) for more details and examples of cursors use 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/ISO 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 clause) 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 inner FOR statements.

- 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.

Supported Data Types

All data types supported that Vantage supports can be used for stored procedure parameters and local variables, including the JSON data type, UDTs (except VARIANT_TYPE UDTs), BLOBs, and CLOBs.

Note:

For correct operation of a UDT within a stored procedure, the UDT must have the mandatory ordering and transform functionality defined. In addition, the `tosql` and `fromsql` transform routines must be backed up with an equivalent set of predefined-to-UDT and UDT-to-predefined implicit cast definitions. The easiest way to do this is to reference the same routines in both the `CREATE TRANSFORM` and `CREATE CAST` statements.

For a distinct UDT, you can use its system-generated default transform and implicit casting functionality.

For a structured UDT, however, you must explicitly define the functionality using `CREATE TRANSFORM` and `CREATE CAST` statements.

NOTICE

KANJI1 support is deprecated. KANJI1 is not allowed as a default character set. The system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

TD_ANYTYPE parameter data type cannot be used with SQL stored procedures.

Users may create a Teradata stored procedure containing one or more parameters/variables that are a NUMBER data type. A NUMBER data type may be used to define IN, OUT, or INOUT parameters. A NUMBER data type may also be used to define local variables in the body of the stored procedure.

User-Defined Functions

You can invoke a UDF from a stored procedure control statement, as well as from SQL statements in a stored procedure that are not control statements. This includes UDFs which have VARIANT_TYPE input parameters.

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 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	

Other delimiters like the comma character (,), the full stop 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.

Result Code Variables

For the definition and details of result code variables, see [Result Code Variables](#), [SQL Communications Area \(SQLCA\)](#) and [SQLSTATE Mappings](#).

For a complete listing of the return codes mapped to their corresponding SQLSTATE codes, see [SQLSTATE Mappings](#).

For information on mapping SQLCODEs to the system error codes, see [SQL Communications Area \(SQLCA\)](#).

Triggers

Triggers can call stored procedures, though the following restrictions apply:

The following statements are not allowed inside a stored procedure called from a trigger:

- DDL statements
- DCL statements
- BEGIN TRANSACTION or BT
- END TRANSACTION or ET
- COMMIT
- Exception handling statements

INOUT and OUT parameters are not allowed in a stored procedure called from a trigger.

A row can be passed to a stored procedure, but a table cannot.

In the following valid example, a row is passed to the stored procedure named *Sp1*:

```
CREATE TRIGGER Trig1 AFTER INSERT ON Tab1
REFERENCING NEW AS NewRow
```

```
FOR EACH ROW
(CALL Sp1(NewRow.C1,NewRow.C2);)
```

In the following example, a table is passed to a the stored procedure named *Sp1*. This operation is not valid, and it returns an error to the requestor.

```
CREATE TRIGGER Trig1 AFTER INSERT ON Tab1
REFERENCING NEW_TABLE AS NewTable
FOR EACH STATEMENT
(CALL Sp1(NewTable.c1,NewTable.C2);)
```

Queue Tables

Stored procedures support queue tables.

Multistatement Requests

Stored procedures support multistatement requests.

Comments

Comments, with the exception of nested bracketed comments, can be used in a stored procedure.

The ANSI/ISO SQL:2011 definition of comments includes what are sometimes described as Teradata-style comments. The standard discriminates between the comment types as follows:

Comment Structure	ANSI Name
--	Simple comment
/* ... */	Bracketed comment

Bracketed comments are sometimes called Teradata-style comments, though they are also defined by the ANSI/ISO SQL:2011 standard.

Related Information

- Rules applying to naming a stored procedure, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.
- Keywords, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.
- Local variables, see [DECLARE](#) and [Supported Data Types](#).
- Stored procedure parameters, see [Supported Data Types](#).
- [Host Variables](#) and “USING Row Descriptor,” see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for a description of the same concept as a parameter, but going by a different name.
- Rules, details and examples of the use of parameters in stored procedures, see the description of the CALL statement in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 and the

description of the CREATE PROCEDURE statement in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

- Details on data types and usage considerations, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141 and *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- Guidelines for manipulating LOBS in a stored procedure, see the information about CREATE PROCEDURE (Internal form) in *Teradata Vantage™ - SQL Fundamentals*, B035-1141.
- The JSON data type, see *Teradata Vantage™ - JSON Data Type*, B035-1150.
- Correct operation of a UDT within a stored procedure, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for more information about these statements.
- Stored procedures support multistatement requests—see the information about CREATE PROCEDURE (Internal Form) in *Teradata Vantage™ - SQL Fundamentals*, B035-1141.
- Queue tables, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

DDL Statements in Stored Procedures

Supported DDL Statements

You can use the following SQL DDL statements in a stored procedure:

- ALTER FUNCTION
- ALTER TABLE
- ALTER TRIGGER
- BEGIN LOGGING
- COLLECT STATISTICS (Optimizer Form)
- COMMENT
- CREATE CAST
- CREATE DATABASE
- CREATE ERROR TABLE
- CREATE HASH INDEX
- CREATE INDEX
- CREATE JOIN INDEX
- CREATE MACRO
- CREATE ORDERING
- CREATE PROFILE
- CREATE RECURSIVE VIEW
- CREATE ROLE
- CREATE TABLE
- CREATE TRANSFORM
- CREATE TRIGGER
- CREATE USER
- CREATE VIEW

- DELETE DATABASE
- DELETE USER
- DROP CAST
- DROP DATABASE
- DROP ERROR TABLE
- DROP HASH INDEX
- DROP INDEX
- DROP JOIN INDEX
- DROP MACRO
- DROP ORDERING
- DROP PROCEDURE
- DROP PROFILE
- DROP ROLE
- DROP STATISTICS (Optimizer Form)
- DROP TABLE
- DROP TRANSFORM
- 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 CAST
- REPLACE FUNCTION
- REPLACE MACRO
- REPLACE ORDERING
- REPLACE TRANSFORM
- REPLACE TRIGGER
- REPLACE VIEW
- SET QUERY_BAND FOR TRANSACTION

Usage Notes

Statement	Notes
COMMENT	You can use only DDL COMMENT statements in a stored procedure. You <i>cannot</i> specify DML COMMENT statements, which are restricted to embedded SQL applications, to fetch the comments for database objects, columns of a table, and parameters.
CREATE TABLE	All variations of CREATE TABLE statement are valid.
CREATE VOLATILE TABLE	If you include a CREATE VOLATILE TABLE statement in a stored procedure, the volatile table is created in your login 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.
CREATE DATABASE/ CREATE USER	A CREATE DATABASE or CREATE USER statement in a stored procedure must contain a FROM clause. The specified database is the immediate owner of the USER or DATABASE created. If the CREATE DATABASE or CREATE USER omits the FROM clause, a 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 a runtime exception during stored procedure execution.
SET QUERY_BAND	A SET QUERY_BAND statement in a stored procedure must specify the FOR TRANSACTION clause. You cannot set the query band for a session within a stored procedure. You can pass the query band specification string of a SET QUERY_BAND statement into a stored procedure as an IN or INOUT argument.

Unsupported DDL Statements

You cannot use the following SQL DDL statements in a stored procedure:

- ALTER METHOD
- ALTER PROCEDURE
- ALTER TYPE
- CREATE FUNCTION
- CREATE METHOD
- CREATE PROCEDURE
- CREATE TABLE (queue and trace table forms)
- CREATE TYPE (all forms)
- DATABASE
- DROP TYPE
- EXPLAIN
- HELP (all forms)

- REPLACE METHOD
- REPLACE PROCEDURE
- REPLACE TYPE
- SET QUERY_BAND FOR SESSION
- SET ROLE
- SET SESSION (all forms)
- SET TIME ZONE
- SHOW (all forms)

Transaction Mode Impact on DDL Statements

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 execute 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.

Related Information

- Supported DDL statements, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.
- Privilege checking performed for a SET QUERY_BAND statement, see [Checking Privileges for Stored Procedures](#).

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
- COLLECT STATISTICS (QCD Form)
- COMMIT
- DECLARE CURSOR (selection form)
- DELETE (all forms)
- DROP STATISTICS (QCD Form)

- FETCH
- INSERT
- MERGE
- OPEN
- ROLLBACK
- SELECT (only in cursors)
- SELECT AND CONSUME TOP 1 (only in positioned cursors)
- SELECT INTO
- SELECT AND CONSUME TOP 1 INTO
- UPDATE, including searched, positioned, and upsert form

Unsupported DML Statements

You cannot use the following SQL DML statements in a stored procedure:

- CHECKPOINT
- COLLECT DEMOGRAPHICS

Restricting SQL Statement Execution

The `SQL_data_access` clause in the CREATE/REPLACE PROCEDURE statement indicates whether or not the stored procedure can issue any SQL statements and, if so, what type. The `SQL_data_access` clause includes the following options:

This option...	Indicates that the stored procedure can execute...
CONTAINS SQL	SQL control statements. The stored procedure cannot read or modify SQL data.
READS SQL DATA	statements that read SQL data, such as a FETCH statement. The stored procedure cannot execute statements that modify SQL data.
MODIFIES SQL DATA	all SQL statements that can be called from a stored procedure, such as UPDATE, INSERT, or DELETE statements. This is the default when the clause is not included in the CREATE/REPLACE PROCEDURE statement.

The system returns an exception as follows:

If this option...	Attempts to...	This message returns...
CONTAINS SQL	read or modify SQL data or calls a procedure that attempts to read or modify SQL data	'2F004' reading SQL-data not permitted.
READS SQL DATA	modify SQL data or calls a procedure that attempts to modify SQL data	'2F002' modifying SQL-data not permitted.

Related Information

- DML statements, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

- `SQL_data_access` clause, see *CREATE PROCEDURE (SQL form)/REPLACE PROCEDURE*, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- `SQL_data_access` exceptions, see [SQL Communications Area \(SQLCA\)](#) for `SQLCODE` to `SQLSTATE` mappings.
- Supported DML statements, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

DCL Statements in Stored Procedures

Supported DCL Statements

You can use the following SQL DCL statements in a stored procedure.

- GIVE
- GRANT (all forms)
- GRANT CONNECT THROUGH
- GRANT LOGON
- REVOKE (all forms)
- REVOKE CONNECT THROUGH
- REVOKE LOGON

Transaction Mode Impact on DCL Statements

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.

Diagnostics Statements in Stored Procedures

You can use the following diagnostics statements in a stored procedure:

- GET DIAGNOSTICS
- SIGNAL
- RESIGNAL

Related Information

- DCL statements, see *Teradata Vantage™ - SQL Data Control Language*, B035-1149.
- Diagnostics statements in stored procedures, see [GET DIAGNOSTICS](#), [SQL Communications Area \(SQLCA\)](#), [SIGNAL](#) and [RESIGNAL](#).

SQL Operations on Stored Procedures

The following SQL statements execute 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 from BTEQ, ODBC, JDBC, and CLIV2 applications.

From the BTEQ and TeqTalk utilities, you must submit CREATE/REPLACE PROCEDURE statements in the file referenced by the COMPILE command.

Using Stored Procedures with Unicode Pass Through

Unicode Pass Through (UPT) is a Unicode error handling feature. The feature gives users the ability to allow Pass Through Characters (PTCs) to be imported into and exported from Teradata.

In a session where UPT is disabled, you cannot create a stored procedure that references a PTC inside the stored procedure body. Such a procedure can only be created within a Pass Through session.

When calling a stored procedure that is defined with a PTC parameter or references a PTC in the stored procedure body, the following applies:

- In a session where UPT is disabled, you cannot call a stored procedure that is defined with a PTC parameter.
- In a Pass Through session, you can successfully call a stored procedure that is defined with a PTC parameter and pass a PTC literal to it as an argument.
- In all sessions (UPT enabled or disabled), you can call a stored procedure that references a PTC in the body of the the stored procedure.

The following table summarizes the usage for stored procedures with PTCs.

Stored Procedure Creation or Execution	Current Session Type	Expected Result
Creating a stored procedure that references a PTC in the procedure body	Pass Through session	Successful

Stored Procedure Creation or Execution	Current Session Type	Expected Result
	UPT is disabled	Error
Calling a stored procedure that is defined with a PTC parameter	Pass Through session	Successful
	UPT is disabled	Error
Calling a stored procedure that references a PTC in the procedure body	Pass Through session	Successful
	UPT is disabled	Successful
Calling a stored procedure that is defined with a PTC parameter and also references a PTC in the procedure body	Pass Through session	Successful
	UPT is disabled	Error

For more information, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

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.

Related Information

For more information about the list of control statements and control declarations used to write a stored procedure, see [SQL Control Statements](#).

Completion, Exception, and User-defined Condition Handlers

Stored procedures support completion condition, exception condition, and user-defined condition handlers of the CONTINUE and EXIT types, including:

- SQLSTATE-based condition handlers
- Generic exception condition handler for SQLEXCEPTION conditions
- Generic completion condition handlers for SQLWARNING and NOT FOUND conditions
- Condition handlers for user-defined conditions

Related Information

For more information about condition handling in stored procedures, see [Condition Handling](#).

Cursor Declarations

See [Cursors and Stored Procedures](#), [DECLARE CURSOR \(Stored Procedures Form\)](#) and [FOR](#) for details.

Returning Result Sets from a Stored Procedure

You can use the DYNAMIC RESULT SETS clause in the CREATE/REPLACE PROCEDURE statement to return up to 15 result sets to the caller (an external stored procedure) or client (an application such as BTEQ) of the stored procedure. The stored procedure returns result sets in the form of a multistatement response spool.

The stored procedure assumes zero result sets if the clause is absent. The stored procedure may return no result sets, or fewer result sets than specified by the DYNAMIC RESULT SETS clause.

This clause is optional. Do not use it if you do not want the stored procedure to return result sets.

Creating a Stored Procedure and Returning Result Sets to the Caller or Client

To create a stored procedure that returns result sets:

1. Use the DYNAMIC RESULT SETS clause in the CREATE/REPLACE PROCEDURE statement to specify the number of result sets the stored procedure returns. For example, the following statement defines a stored procedure that returns one result set:

```
CREATE PROCEDURE sp1 (IN SqlStr VARCHAR(50), IN a INT)
    DYNAMIC RESULT SETS 1
```

2. Use a DECLARE CURSOR statement to declare a result set cursor for each result set the stored procedure returns.
 - Specify WITH RETURN ONLY TO CALLER or WITH RETURN ONLY for the stored procedure to return the result set(s) only to the caller of the target procedure. If the client called the procedure, the stored procedure returns the result set(s) to the client application. If a stored procedure called the target procedure, the stored procedure returns the result set(s) to the calling procedure, which can also be an external stored procedure that allows SQL.
 - Specify WITH RETURN TO CALLER or WITH RETURN for the stored procedure to return the result set(s) to both the caller of the target procedure and to the target procedure (the procedure that opened the cursor).
 - Specify WITH RETURN TO CLIENT for the stored procedure to return the result set(s) to both the client (application such as BTEQ) and to the target procedure (the procedure that opened the cursor).
 - Specify WITH RETURN ONLY TO CLIENT for the stored procedure to return the result set(s) only to the application.

- If the SELECT statement that produces the result set is static, include it as the SELECT portion of the DECLARE CURSOR statement. For example:

```
DECLARE cur1 CURSOR WITH RETURN ONLY FOR
  SELECT * FROM m1;
```

- To use the dynamic form of the DECLARE CURSOR statement to return a result set, use a statement name instead of SELECT. For example:

```
DECLARE c1 CURSOR WITH RETURN ONLY FOR s1
```

3. Use a PREPARE statement for each dynamic form of DECLARE CURSOR to prepare the statement name that was specified. For example:

```
PREPARE s1 FROM SqlStr;
```

4. Use an OPEN statement to open each result set cursor to execute the static or dynamic SELECT statement. If the dynamic select statement uses parameter markers (the question mark character), specify a USING clause to identify the variables to use as input. For example:

```
OPEN c1 USING a;
```

Note that you must specify the same number of variables the USING clause as the number of parameter markers.

5. Use the FETCH statement to position the result set cursor to read from the result set.
6. Leave the result set cursors open to return the result sets to the caller or client. If the stored procedure closes the result set cursor, the result set is deleted and not returned. The result sets are returned in the order they were opened.

Example: Static Form of the DECLARE CURSOR Statement

Following is an example of using the DYNAMIC RESULT SETS clause and the static form of the DECLARE CURSOR statement to create a stored procedure that returns result sets.

```
CREATE PROCEDURE Sample_p (INOUT c INTEGER)
  DYNAMIC RESULT SETS 2
BEGIN
  DECLARE cur1 CURSOR WITH RETURN ONLY FOR
    SELECT * FROM m1;
  DECLARE cur2 CURSOR WITH RETURN ONLY FOR
    SELECT * FROM m2 WHERE m2.a > c;
  SET c = c + 1;
  OPEN cur1;
  OPEN cur2;
END;
```

Following is an example of how BTEQ (the client) would read the result sets for the sample stored procedure:

```
BTEQ -- Enter your DBC/SQL request or BTEQ command:
CALL sample_p(1);
*** Procedure has been executed.
*** Warning: 3212 The stored procedure returned one or more result sets.
*** Total elapsed time was 1 second.
      1
-----
      2
*** Procedure dynamic result set. One row found. 2 columns returned.
*** Starting Row Number: 1
*** Database Name: FSK
*** Procedure Name: SAMPLE_P
      a                      b
-----
      1  2.00000000000000E 000
*** Procedure dynamic result set. One row found. 2 columns returned.
*** Starting Row Number: 1
*** Database Name: FSK
*** Procedure Name: SAMPLE_P
      a                      b
-----
      2  4.00000000000000E 000
```

Example: Dynamic Form of the DECLARE Statement

In the following example, the DECLARE CURSOR statement includes statement name s1. The PREPARE statement references s1 to prepare the dynamic select statement contained in SqlStr of the CREATE PROCEDURE statement. The OPEN statement opens the result set cursor c1 specified in the DECLARE CURSOR statement, USING the parameter a specified in the CREATE PROCEDURE statement.

```
CREATE PROCEDURE sp1 (IN SqlStr VARCHAR(50), IN a INT)
DYNAMIC RESULT SETS 1
BEGIN
  DECLARE c1 CURSOR WITH RETURN ONLY FOR s1;
  PREPARE s1 FROM SqlStr;
  OPEN c1 USING a;
END;
```

Following is an example of a dynamic select statement you might input in BTEQ. Note that the CALL statement includes the same number of parameter markers as the USING clause variables.

```
CALL sp1('sel * from tab1 where a = ? order by 1;',1);
```

Related Information

- Reading result sets in an external stored procedure, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- DECLARE CURSOR statement, see [DECLARE CURSOR](#).

Using Dynamic SQL in Stored Procedures

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 Statements

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 that Does Not Return a Results Set

You can invoke dynamic SQL in a stored procedure that does not return a results set with either of these statements:

- An EXECUTE or EXECUTE IMMEDIATE statement.
- The following CALL statement.

CALL Statement Syntax

```
CALL dbc.SysExecSQL ( string_expression ) [;]
```

Syntax Elements

dbc

The qualifying database name DBC must be specified, unless the current default database is DBC.

SysExecSQL

The string used for:

- Invoking dynamic SQL
- Validating the user rights.

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

Example: Using Dynamic SQL Statements With an EXECUTE IMMEDIATE Statement

The following example illustrates the use of dynamic SQL statements within a stored procedure that does not return a results set and that uses an EXECUTE IMMEDIATE statement.

```
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)' ;
  DECLARE sqlstr VARCHAR(500);
  SET sqlstr = 'CREATE TABLE ' || my_database ||
    '.' || my_table || sales_columns ;
  EXECUTE IMMEDIATE sqlstr;
END;
```

Example: Using Dynamic SQL Statements With a CALL Statement

The following example illustrates the use of dynamic SQL statements within a stored procedure that does not return a results set and that uses a CALL statement:

```
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;
```

Invoking Dynamic SQL that Returns a Results Set

You can invoke dynamic SQL in a stored procedure that returns a results set using the dynamic form of the DECLARE CURSOR (Stored Procedures Form) statement.

Example: Using Dynamic SQL Statements Within a Stored Procedure that Returns a Results Set

The following example illustrates the use of dynamic SQL statements within a stored procedure that returns a results set. Note that the procedure does not use a WITH RETURN statement.

```
CREATE PROCEDURE GetEmployeeSalary
(IN EmpName VARCHAR(100), OUT Salary DEC(10,2))
BEGIN
  DECLARE SqlStr VARCHAR(1000);
  DECLARE C1 CURSOR FOR S1;
  SET SqlStr = 'SELECT Salary FROM EmployeeTable WHERE EmpName = ?';
  PREPARE S1 FROM SqlStr;
  OPEN C1 USING EmpName;
  FETCH C1 INTO Salary;
  CLOSE C1;
END;
```

SQL Statements that Cannot Be Used Dynamically

The following SQL statements cannot be used dynamically when you write a stored procedure that does not return a results set:

- ALTER PROCEDURE
- CALL
- CREATE PROCEDURE
- DATABASE
- EXPLAIN
- HELP (all forms)
- OPEN
- PREPARE
- REPLACE PROCEDURE
- SELECT
- SELECT INTO
- SET ROLE
- SET SESSION ACCOUNT
- SET SESSION COLLATION
- SET SESSION DATEFORM
- SET TIME ZONE
- SHOW
- Cursor statements, including CLOSE, FETCH, and OPEN

Note:

The only SQL statement that is supported when you use of dynamic SQL statements in a stored procedure that returns a results set is SELECT.

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.

Rules for Dynamic SQL Statements

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.

Note:

If the creator of the stored procedure is not the immediate owner, and the OWNER SQL SECURITY option is specified, the system verifies that the user has the CREATE OWNER PROCEDURE privilege or else a compilation error is reported, and the procedure is not created.

You can specify multistatement requests in dynamic SQL requests within a BEGIN REQUEST ... END REQUEST block. Otherwise, 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 a DDL COMMENT statement as dynamic SQL in a stored procedure. You cannot specify a 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.

No specific privilege is required to use the CALL DBC.SysExecSQL statement.

Related Information

For more information about objects referenced in, or created through the dynamic SQL statements, see [Ownership of Objects Created by Stored Procedures](#).

Recursive Stored Procedures

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.

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. An SPL compilation warning is reported when an attempt is made to create either of the procedures because the referenced procedure does not exist.

This warning 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.

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 example illustrates the creation of a stored procedure that directly references itself. The stored procedure is created with a compilation warning because the procedure being invoked by the CALL statement does not exist at compilation time.

The second example illustrates the creation of a stored procedure that entails mutual recursion and avoids any compilation warnings. This is useful for creating a new stored procedure or for changing the parameters of an existing procedure.

Example: Recursion

Assume that the table named Employee exists.

```
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: 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.

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;

```

```
END IF;
END;
```

Related Information

For more information about mutual recursion, see [Example: Mutual Recursion](#).

Stored Procedures and Tactical Queries

Stored procedures can be of great benefit for some tactical query applications. This section provides:

- Some examples of using stored procedures to process complex updates, reduce workload overhead, and maintain security audits
- A comparison of the relative efficiency of stored procedures and macros for different tactical query applications

Using Stored Procedures to Execute 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 execute 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 execute the following complex update, a stored procedure would execute 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 execute 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.

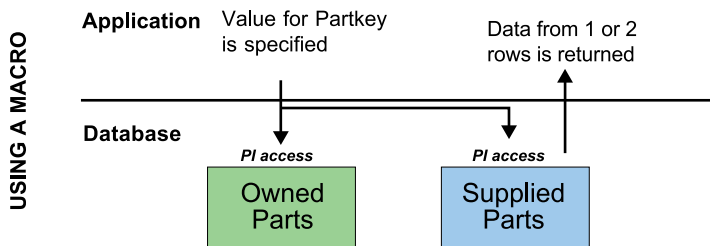
Because stored procedures cannot return multirow result sets, macros are usually a better approach to use when you need to return more than one row from a table.

Using Stored Procedures to Eliminate Unnecessary Database Work

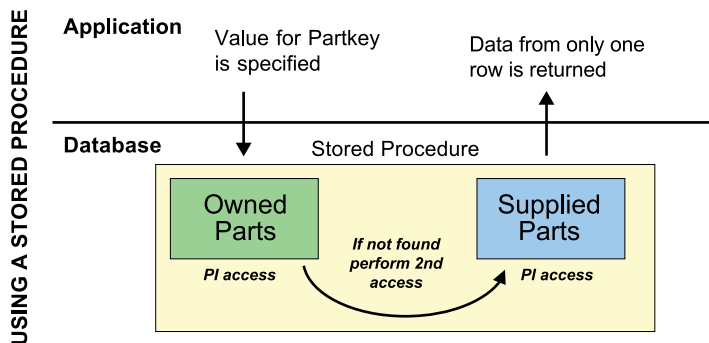
Suppose you have an application that uses data from only one of two possible tables, but you cannot know which of the two tables is required until you attempt to access the first. A stored procedure can eliminate processing that would otherwise be necessary to solve the problem.

To illustrate, consider a database with two tables that contain owned_parts and supplied_parts parts information. Assume you have a business rule that says that you return owned_parts data if it exists, and only return supplied_parts data if owned_parts data does not have the specified prime key value. Also assume that you have a tactical query that probes for parts data by specifying a part key.

A macro 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.

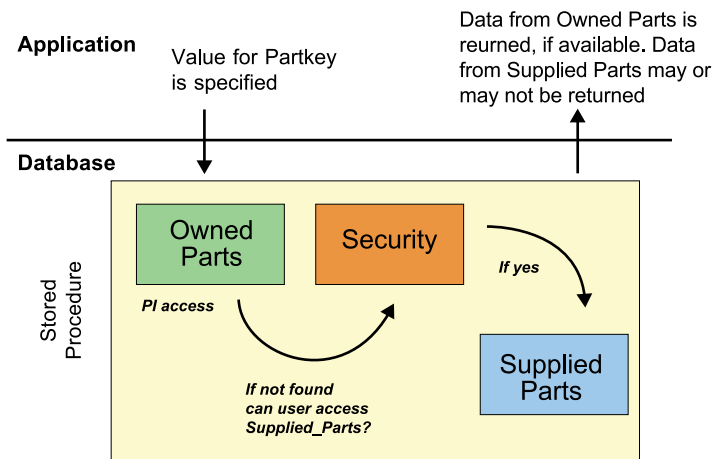


This extra work is unnecessary if you write a stored procedure to solve the problem. You can code logic in a stored procedure to determine if a row has been found in the owned_parts table, and, if so, to then return to the application without attempting to access the supplied_parts table. The following 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](#), 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.

Macros or Stored Procedures for Tactical Queries

Macros were once a better choice than stored procedures for simple requests, multistatement requests (statements executed in parallel), and statements returning multiple rows because performance was almost always better. Stored procedures now support multistatement requests and result sets, and with their conditional logic, are a better choice than macros for running tactical queries.

Simple Requests

Stored procedures may perform better than macros for simple requests. You can use either macros or stored procedures to run simple requests.

Multistatement Requests

Both macros and stored procedures support multistatement requests. Multistatement request performance for stored procedures is the same, if not better than, macro performance.

Statements Returning Multiple Rows

Stored procedures now support result sets, which means that a stored procedure can now return multiple rows. Macros have no advantage over stored procedures in returning multiple rows.

Differences Between Macros and Stored Procedures

The following table summarizes the differences between macros and stored procedures.

Macro	Stored Procedure
Limited procedural logic.	Sophisticated procedural logic.
Can return multirow result sets for the same request.	DYNAMIC RESULT SETS allows the stored procedure to return up to 15 result sets.

Macro	Stored Procedure
Multistatement request parallelizes multiple single row statements.	Multistatement request using the BEGIN REQUEST – END REQUEST statements parallelizes multiple single row DML statements.
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.	Can be invoked by a trigger.

Debugging Stored Procedures

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.

Comparing Debugging Methods

The following table describes the advantages and disadvantages of these two methods. Evaluate the methods based on your requirements.

Method	Advantages	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 must 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 must disable or remove the CALL statement(s) from the stored procedure body after debugging, and recompile the procedure.

Using INSERT Statements for Debugging

You can use the INSERT statements in the stored procedure to insert any values or text in any table. 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 executed, the INSERT statement specified in the FOR statement is executed 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 executed. It inserts the stored procedure name, and values of SQLCODE and SQLSTATE status variables in the ErrorLogTable. You can query this table 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. 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 rows 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 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

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.

Assumptions

- 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.
- 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 1000 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 following CREATE PROCEDURE statement creates the sample stored procedure. 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:

- Capture and add 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.

Compilation results in errors and the stored procedure is not created if any database object referenced in the cursor SELECT statement is missing.

When CREATE PROCEDURE is executed from CLIV2, ODBC, or JDBC, an SPL compiler in the database compiles the stored procedure.

Condition Handling

This section describes the SQL stored procedure statements that enable stored procedures to handle completion, exception, and warning conditions gracefully. The section begins with a description of stored procedure condition handling and then describes the different types of condition handling statements individually.

Benefits of Condition Handling

- 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), thereby allowing the stored procedure to continue with its execution.
- Providing a different handling mechanism for different conditions.

Condition Handling Terms

Term	Definition
Completion condition	<p>When the execution of an SQL statement, including a control statement, is completed without any fatal event, and the database 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 nonzero value depending on the SQL statement.</p> <p>Examples of completion condition:</p> <ul style="list-style-type: none"> • An SQL statement, including a control statement, is executed 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	<p>Represents an error or informational state caused by execution of an SQL statement, including a control statement.</p> <p>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.</p>
Condition handler	<p>A construct defined to execute one or more actions depending on the SQLSTATE value returned to an application or on the condition specified by <i>condition_name</i> in the handler declaration.</p> <p>The handler first defines one or more conditions to be handled and then the associated actions. The actions are executed when the corresponding condition occurs during stored procedure execution.</p>

Term	Definition
	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 <code>SQLEXCEPTION</code> instead of one or more specific SQLSTATE codes. <code>SQLEXCEPTION</code> is treated as a generic exception condition handler.
Condition name	A mnemonic name that can be associated with an SQLSTATE code in a <code>DECLARE CONDITION</code> statement. If you do not associate an SQLSTATE value with the condition name, then it is used to represent a user-defined condition. A condition name identifies the condition to be handled and can be used only in condition declarations, handler declarations, <code>SIGNAL</code> statements and <code>RESIGNAL</code> statements.
Condition value	An SQLSTATE value which is a 5-character string literal.
Exception condition	When the execution of an SQL statement, including a control statement, is unsuccessful. The database response indicates an <code>ERROR</code> or <code>FAILURE</code> . After an exception condition is handled, the <code>SQLCODE</code> reflects the return code, the <code>SQLSTATE</code> is set to a value other than '00000' representing the exception condition, and the <code>ACTIVITY_COUNT</code> is set to "0". Examples of exception conditions include: <ul style="list-style-type: none"> • Invalid cursor state • Divide-by-zero violation • String truncation (only in ANSI session mode) • Cardinality violation
Generic condition handler	Handler declared to handle generic conditions, represented by the keywords <code>SQLEXCEPTION</code> , <code>SQLWARNING</code> , or <code>NOT FOUND</code> . These keywords are declared instead of one or more specific SQLSTATE codes. <code>SQLEXCEPTION</code> represents all exception conditions. <code>SQLWARNING</code> represents all completion conditions except successful completion and "no data found" completion conditions. <code>NOT FOUND</code> represents all "no data found" completion conditions.
Successful completion	When the database 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 <code>SQLSTATE</code> is set to '00000', <code>SQLCODE</code> is set to "0" and the <code>ACTIVITY_COUNT</code> is set to either "0" or a nonzero value depending on the SQL statement. The status variable values are unchanged for a control statement.
User-defined condition	A condition defined by the user for handling situations that are specific to a stored procedure and that are not represented by any SQLSTATE value.

Related Information

- SQLSTATE variable, see [SQLSTATE](#).
- Declaring a user-defined condition, see [DECLARE CONDITION](#).

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.

Related Information

- SQLSTATE, see [SQLSTATE](#).
- SQLSTATE codes and their mappings to database error codes, see [SQLSTATE Mappings](#).

Diagnostics Area

The Diagnostics Area contains detailed information about the execution status of the statements within a stored procedure. It is divided into a Statement Area and zero or more Condition Areas. The Statement Area contains information about the execution of a statement in a stored procedure. A Condition Area contains information about the successful, completion or exception condition that resulted from the execution of a statement.

You can use the GET DIAGNOSTICS statement to retrieve information about successful, exception, or completion conditions from the Diagnostics Area.

Related Information

For more information about the Diagnostics Area, see [Diagnostics Area](#) and [GET DIAGNOSTICS](#).

Conditions and Condition Handlers

Condition handlers can be either SQLSTATE-based, generic, or associated with an user-defined condition.

SQLSTATE-based Condition Handlers

Execution of the SQL statements within a stored procedure may result in certain completion, exception, or warning conditions. These conditions are posted to the SQLSTATE status variable. You can declare a condition handler and associate it with one or more SQLSTATE values. The condition handler will execute its actions when the conditions represented by the specified SQLSTATE values occur during stored procedure execution.

Generic Conditions and Handlers

Generic conditions are represented by the keywords SQLEXCEPTION, SQLWARNING, or NOT FOUND. You can declare a condition handler and associate it with one or more generic conditions instead of specific SQLSTATE values. The condition handler will execute its actions when the specified generic conditions occur during stored procedure execution.

Condition Handlers for Condition Names

SQLSTATE is a 5-character string value. You can declare a mnemonic name and associate it with an SQLSTATE value to make it easier to remember what condition the SQLSTATE value represents. You can declare a condition handler and associate it with one or more condition names. The condition handler will execute its actions when the conditions identified by the condition names or the associated SQLSTATE values occur during stored procedure execution.

User-Defined Conditions and Handlers

You can define custom conditions by declaring a condition name without associating it with an SQLSTATE value. This is useful if the conditions represented by the SQLSTATE values do not meet your needs. You can declare a condition handler and associate it with one or more user-defined conditions. The condition handler will execute its actions when the user-defined conditions occur during stored procedure execution. You can use the SIGNAL statement to explicitly raise a user-defined condition.

Related Information

- Execution of the SQL statements within a stored procedure, see [DECLARE HANDLER \(Basic Syntax\)](#).
- Behavior of the generic condition handlers, see [DECLARE HANDLER \(SQLEXCEPTION Type\)](#), [DECLARE HANDLER \(SQLWARNING Type\)](#) and [DECLARE HANDLER \(NOT FOUND Type\)](#).
- Declaring a condition name, see [DECLARE CONDITION](#).
- Declaring a user-defined condition and associating it with a condition handler, see [DECLARE CONDITION](#) and [DECLARE HANDLER \(Basic Syntax\)](#).
- SIGNAL statement, see [SIGNAL](#).

Condition Handler Types

You must specify one of the following handler types in a handler declaration:

- CONTINUE
- EXIT

When a stored procedure encounters a condition, the specified handler action is executed regardless of the handler type specified.

The difference is that CONTINUE passes control to the next statement within the compound statement, and EXIT passes control to the next statement outside the compound statement that contains the handler.

The behavior of CONTINUE and EXIT handlers is described in [DECLARE HANDLER \(CONTINUE Type\)](#) and [DECLARE HANDLER \(EXIT Type\)](#).

Raising Conditions

During execution of SQL statements within a stored procedure, conditions are raised when the SQL statements complete execution or result in an error or warning.

You can also explicitly raise an exception condition, a completion condition (other than successful condition), or a user-defined condition using the `SIGNAL` statement.

You can use the `RESIGNAL` statement to resignal or invoke a condition from a handler declaration. You can explicitly specify the `RESIGNAL` statement only in a handler declaration. `RESIGNAL` always propagates the condition outward. When a `RESIGNAL` statement is submitted from a handler action, the outer containing compound statements are searched to find the appropriate handler to handle the condition raised by the `RESIGNAL` statement.

Related Information

For more information about, see `SIGNAL` and `RESIGNAL` statements, see [SIGNAL](#) and [RESIGNAL](#).

Control Statement Handling

The handling of conditions raised by SQL statements, including control statements, within a stored procedure is described in [Statement-Specific Condition Handling](#).

Condition Handler Rules

- You can declare condition handlers for exception conditions and completion conditions other than successful completion. No condition handler can be defined for successful completion (`SQLSTATE = '00000'`).
- 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.

- You cannot repeat the same `SQLSTATE` code in a `DECLARE HANDLER` statement.
- You cannot declare the same `SQLSTATE` code for multiple condition handlers in the same 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.

- You can specify `SQLEXCEPTION`, `SQLWARNING`, `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.

The following additional rules apply when declaring a handler which specifies a condition name.

- You can specify more than one condition name in a handler declaration as long as the condition names are not identical. The handler action is associated with every condition name in the DECLARE HANDLER statement.
- You cannot repeat the same condition name within a handler declaration. Otherwise, error SPL1052 is reported during stored procedure compilation, and the stored procedure is not created.
- You cannot specify a condition name and a generic condition in the same handler declaration. Otherwise, error SPL1082 is reported during stored procedure compilation, and the stored procedure is not created.
- You cannot specify a condition name and the SQLSTATE value associated with the condition name in the same handler declaration. Otherwise, error SPL1054 is reported during stored procedure compilation, and the stored procedure is not created.
- You cannot declare multiple handler declarations which specify the same condition name within the same compound statement. Otherwise, error SPL1052 is reported during stored procedure compilation, and the stored procedure is not created.
- If you declare a handler for a condition name, you cannot declare another handler in the same compound statement to handle the SQLSTATE value associated with that condition name. Otherwise, error SPL1054 is reported during stored procedure compilation, and the stored procedure is not created.
- If you declare a handler for a condition name that has an SQLSTATE value associated with it, the same handler is also used for handling conditions with that SQLSTATE value.

Example: Using a Condition Name and Its Associated SQLSTATE Value in a Handler

The following example illustrates the usage of a condition name and its associated SQLSTATE value in a handler. The condition declaration defines condition name *divide_by_zero* and associates it with SQLSTATE '22012'. The EXIT handler is defined to handle *divide_by_zero*. During stored procedure execution, the divide-by-zero exception with SQLSTATE '22012' is raised and is handled by the EXIT handler. After successful completion of the EXIT handler statements, control exits the compound statement cs1, and the stored procedure completes successfully.

```
CREATE PROCEDURE condsp1 (INOUT IOParm2 INTEGER,
                        OUT OParam3 INTEGER)
cs1: BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
  DECLARE EXIT HANDLER
    FOR divide_by_zero, SQLSTATE '42000'
    SET OParam3 = 0;
  SET IOParm2 = 0;
  SET OParam3 = 20/IOParm2;    /* raises exception 22012 */
END cs1;
```

Example: Associating a Handler Action with Multiple Condition Names

The following example illustrates the association of the same handler action with multiple condition names. A CONTINUE handler is defined for condition names *divide_by_zero* and *table_does_not_exist*. During stored procedure execution, the CONTINUE handler can handle both exceptions ERRAMPEZERODIV (SQLCODE 2802 and SQLSTATE '22012') and ER RTEQTVNOEXIST (SQLCODE 3807 and SQLSTATE '42000').

```
CREATE PROCEDURE condsp2 (INOUT IOParm2 INTEGER,
                        OUT OParam3 CHAR(30))
Cs1: BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
  DECLARE table_does_not_exist CONDITION FOR SQLSTATE '42000';
  DECLARE CONTINUE HANDLER
    FOR divide_by_zero, table_does_not_exist
    SET OParam3 = 0;
  SET IOParm2=0;
  SET OParam3 = 20/IOParm2; /* raises exception 22012 */
  INSERT notab VALUES (IOParm2 + 20); /* raises exception 42000 */
END Cs1;
BTEQ> DROP TABLE notab;
BTEQ> CALL condsp2(IOParm2, OParam3);
```

Example: Using Handlers for Generic Conditions and Explicitly Declared Conditions

The following example illustrates using different handlers to handle generic conditions and explicitly declared conditions. The second declared handler handles divide-by-zero exceptions. The first handler declared for SQLEXCEPTION handles all other exception conditions.

```
CREATE PROCEDURE condsp3 (OUT OParam3 INTEGER)
cs1: BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
  DECLARE EXIT HANDLER
    FOR SQLEXCEPTION
    SET OParam3 = 0;
  DECLARE EXIT HANDLER
    FOR divide_by_zero
    SET OParam3 = 1;
  ...
END cs1;
```

Example: Results from Using the Same Handler for Both Declared and Generic Conditions

You cannot use the same handler to handle both declared conditions and generic conditions. The handler in this example is defined for both `SQLEXCEPTION` and condition name `divide_by_zero`. During stored procedure compilation, error SPL1082 will be reported, and the stored procedure will not be created.

```
CREATE PROCEDURE condsp4 (OUT OParam3 INTEGER)
cs1: BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
  DECLARE EXIT HANDLER
    FOR SQLEXCEPTION, divide_by_zero
    SET OParam3 = 0;
  ...
END cs1;
```

Example: Defining the Handle Condition and the SQLSTATE

You cannot declare a handler for a condition name and the `SQLSTATE` value associated with the condition name in the same handler. In this example, the handler is defined to handle condition name `divide_by_zero` and the `SQLSTATE` value '22012' associated with `divide_by_zero`. During stored procedure compilation, error SPL1054 will be reported, and the stored procedure will not be created.

```
CREATE PROCEDURE condsp5 (OUT OParam3 INTEGER)
cs1: BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
  DECLARE EXIT HANDLER
    FOR divide_by_zero, SQLSTATE '22012'
    SET OParam3 = 0;
  ...
END cs1;
```

Example: Results When Declaring Two Handlers for the Same Condition Name

You cannot declare multiple handlers for the same condition name within the same compound statement. In this example, two handlers are defined to handle the same condition name, `divide_by_zero`, in the compound statement `cs1`. During stored procedure compilation, error SPL1052 will be reported, and the stored procedure will not be created.

```
CREATE PROCEDURE condsp6 (OUT OParam3 INTEGER)
cs1: BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
  DECLARE EXIT HANDLER
    FOR divide_by_zero
    SET OParam3 = 0;
```

```

DECLARE EXIT HANDLER
  FOR divide_by_zero
    SET OParam3 = 1;
...
END cs1;

```

Example: Results When Declaring Handlers for a Condition Name and SQLSTATE Value

You cannot declare a handler for a condition name and another handler for the SQLSTATE value associated with the condition name within the same compound statement. In this example, the first handler is defined to handle the condition name *divide_by_zero*. The second handler is defined for SQLSTATE '22012' which is associated with *divide_by_zero*. Both handlers are defined within the compound statement *cs1*. Therefore, during stored procedure compilation, error SPL1054 will be reported, and the stored procedure will not be created.

```

CREATE PROCEDURE condsp7 (OUT OParam3 INTEGER)
cs1: BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
  DECLARE EXIT HANDLER
    FOR divide_by_zero
      SET OParam3 = 0;
  DECLARE EXIT HANDLER
    FOR SQLSTATE '22012'
      SET OParam3 = 1;
  ...
END cs1;

```

Related Information

- Specifying a condition name and a generic condition in the same handler declaration, see [Example: Using Handlers for Generic Conditions and Explicitly Declared Conditions](#) and [Example: Results from Using the Same Handler for Both Declared and Generic Conditions](#).
- Specifying a condition name and the SQLSTATE value associated with the condition name in the same handler declaration, see [Example: Defining the Handle Condition and the SQLSTATE](#).
- Declaring multiple handler declarations, see [Example: Results When Declaring Two Handlers for the Same Condition Name](#).
- Declaring a handler for a condition name, see [Example: Results When Declaring Handlers for a Condition Name and SQLSTATE Value](#).

Rules for Condition Handlers in Nested Compound Statements

- Exceptions, completion, and user-defined 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:

IF a condition is raised ...	AND an appropriate handler...	THEN ...
in a non-nested compound statement	exists in that statement	the condition is handled and the stored procedure execution either continues or terminates based on the type of handler.
	does not exist and the condition is: <ul style="list-style-type: none"> an exception or unhandled user-defined condition a completion condition 	<ul style="list-style-type: none"> the stored procedure terminates. the stored procedure execution continues.
<ul style="list-style-type: none"> in a nested compound statement, or by a statement other than a handler action statement 	exists within that statement	the condition is handled.
	does not exist within that statement, then the immediate outer statement is searched for a suitable handler. <ul style="list-style-type: none"> 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 and the condition is: <ul style="list-style-type: none"> an exception or unhandled user-defined condition, the stored procedure terminates. a completion condition, the stored procedure execution continues with the statement following the statement that caused the condition. 	

- The rules for propagation and handling of exception, completion and user-defined conditions raised by any handler action statement are different from the above.

Related Information

- Reusing SQLSTATE code for condition handlers in other nested or non-nested compound statements within a stored procedure, see [Example: Reusing the SQLSTATE Code](#).
- Rules for propagation and handling of exception, completion and user-defined conditions raised by any handler action statement, see [Conditions Raised by a Handler Action](#).

Status Variable Values

On completion of a handler action, status variables are set to the following values:

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 that the system returned. An example is a divide-by-zero condition.

Successful evaluation of an expression does not raise a completion condition.

Precedence of Specific Condition Handlers

A stored procedure can contain generic condition handlers and specific condition handlers which handle specific SQLSTATE codes or specific conditions defined by condition names). If a stored procedure contains both specific 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 handlers for exception conditions are specified in a stored procedure.

IF a raised exception ...	THEN this handler action executes ...
matches any of the SQLSTATE values specified for a handler	the action defined for the specific condition handler.
matches any of the condition names specified for a handler	the action defined for the specific condition handler.
does not match a specific SQLSTATE code or condition namespecified for any handler	the action defined for the generic exception condition handler.

- When both SQLWARNING and specific handlers for completion conditions are specified.

IF a raised condition ...	THEN this handler action executes ...
matches any of the SQLSTATE values specified for a handler	the action defined for the specific condition handler.

IF a raised condition ...	THEN this handler action executes ...
matches any of the condition names specified for a handler	the action defined for the specific condition handler.
does not match a specific SQLSTATE code or condition name specified for any handler	the action defined for the generic completion condition handler.

- When both NOT FOUND and specific handlers for “no data found” completion conditions are specified.

IF a “no data found” condition occurs and if the SQLSTATE value...	THEN this handler action executes ...
matches any of the specific completion condition handlers	the action defined for the specific completion condition handler.
does not match any specific condition handler	the action defined for the generic NOT FOUND condition handler.

- The completion condition “no data found” has precedence over the other generic completion conditions. Only a generic NOT FOUND handler or a specific condition 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 session modes in general.

The following table describes the transaction semantics for error and failure conditions, respectively.

FOR this condition type ...	AND this session 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	<ul style="list-style-type: none"> ANSI Teradata 	a transaction-level rollback. All updates executed within the transaction are rolled back.

Example: Using a CONTINUE Handler Associated with An Outer Compound Statement

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 CHARACTER(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 CHARACTER(30), c2 INTEGER);
```

Now execute 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: Exceptions Raised in An Inner Compound Statement

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 CHARACTER(30), IN pAmt INTEGER)
L1: BEGIN
...
L2: BEGIN
  DECLARE vName CHARACTER(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 execute 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: Reusing the SQLSTATE Code

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.

```

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, completion or user-defined conditions raised 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 only exception is the RESIGNAL statement, whose condition is propagated outside the compound statement action clause in a handler.

The following table compares unhandled exception, completion, and user-defined conditions.

IF the unhandled condition is ...	THEN ...
an exception or a user-defined condition	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	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 then handled.

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. Then 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 then not handled.

1. The stored procedure raises an exception with the SQLSTATE code '42000', which means the referenced database object does not exist.
2. Then 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, 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 an 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.

1. The stored procedure raises a completion condition (a warning) with the SQLSTATE code 'T7473', which means the requested sample size is larger than the table rows.
2. Then the handler action raises an exception condition '23505' for attempting to insert a duplicate row in the table.
3. If a suitable handler for '23505' exists within the handler action, 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 suitable handler to handle the condition.
5. If the original completion condition is handled and
 - if the handler is a CONTINUE type, the stored procedure execution continues with the statement that raised the completion condition.
 - if the handler is an 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.

1. The stored procedure raises a completion condition 'T7473', which means the requested sample size is larger than the table rows.
2. The completion condition is handled by a generic condition handler.
3. Then 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: An Exception Raised By a Handler

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 CHARACTER(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 CHARACTER(30), c2 INTEGER);
```

Drop the table *Proc_Error_Tbl*:

```
DROP TABLE Proc_Error_Tbl;
```

Now execute 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'.

Because there is no handler within this handler to handle SQLSTATE '42000', the original condition that invoked the handler, SQLSTATE '23505', is propagated outwards. The outer compound statement has an EXIT handler defined for SQLSTATE '23505'. This handler handles the exception and control exits the compound statement. Because the procedure contains no other statement, the procedure terminates.

Example: Ignoring a Completion Condition Raised Within a Handler

In this example, a completion condition raised within a handler is ignored.

```

CREATE PROCEDURE spSample1(IN pName CHARACTER(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 CHARACTER(30));
CREATE SET TABLE tab1(c1 CHARACTER(30), c2 DECIMAL(18,2));
```

Now execute 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: Combining Conditions Raised by Statements Within and Outside a Handler Action Clause

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 IOParm1 INTEGER,
                       INOUT IOParm2 CHARACTER(20))
Loutermost: BEGIN
  DECLARE Var1 INTEGER DEFAULT 10;
  L1: BEGIN
    DECLARE EXIT HANDLER FOR SQLSTATE '42000'
      -- Statement 3_1a
      SET IOParm2 = '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 IOParm2 = ' Duplicate row error ';
        DECLARE EXIT HANDLER
          FOR SQLSTATE '42000'
          BEGIN

            -- Statement 3_3a
            SET IOParm2 = 'Nonexistent table in inner block ';
            -- Statement 3_3b
            INSERT INTO tab1 VALUES (IOParm1);
```

```

        -- 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: 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 preceding 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: ANSI Session Mode for Reporting Handler Action-Raised Conditions

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI session mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
  WHERE Employee_Number = 1000;
```

```
CALL spSample7b();
```

When the preceding three SQL statements are invoked in ANSI session mode, the following sequence of events occurs:

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 executed.
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*.
7. The following items are not executed:
 - `Statement_7a_3` from *spSample7a*
 - `Statement_7b_3` from *spSample7b*
8. End of process.

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: ANSI Session Mode for Multiple Condition Handlers

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI session mode:

```
INSERT INTO Department VALUES ('10', 'Development');  
  
UPDATE Employee  
  SET Salary_Amount = 10000  
  WHERE Employee_Number = 1000;  
  
CALL spSample10();
```

When the preceding three SQL statements are invoked in ANSI session mode, the following sequence of events occurs:

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 executed:

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

6. End of process.

Statement-Specific Condition Handling

This section describes the behavior of various SQL control 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 executed only after all the open cursors have been closed.

Example: WHILE Loop Exceptions

The following example 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: ANSI Session Mode for Statement-Specific Condition Handling

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI session mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee
  SET Salary_Amount = 10000
  WHERE Employee_Number = 1000;

CALL spSample8();
```

When the preceding three SQL statements are invoked in ANSI session mode, the following sequence of events occurs:

When the preceding three SQL statements are invoked in ANSI session mode, the following sequence of events occurs:

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, execution 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 executes 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
6. End of process.

When stored procedure *spSample8* is created in a session in Teradata session mode, the process described above applies with one difference: because every request is an implicit transaction in Teradata session 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: Exceptions Raised By an IF Statement

The following example 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;
```

The preceding example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI session mode:

```

INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee
SET Salary_Amount = 10000
WHERE Employee_Number = 1000;

CALL spSample9();

```

Consider the following sequence of events with respect to the preceding stored procedure:

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, execution 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 executed.
4. The updates made by the following items remain intact in a transaction that is uncommitted:
 - Statement_9_1
 - Statement_9_3
5. End of process.

Example: Exception Raised by a Condition in WHILE Loop

The following example illustrates the behavior of a 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: ANSI Session Mode

The preceding example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI session mode:

```

INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee
SET Salary_Amount = 10000
WHERE Employee_Number = 1000;

CALL spSample8();

```

When the preceding three SQL statements are invoked in ANSI session mode, the following sequence of events occurs:

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 (statement_8_3), and execution of the stored procedure continues from statement_8_3.

3. Statement_8_2 and the SET statement in the WHILE loop are not executed.
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

5. When stored procedure *spSample8* is created in a session in Teradata session mode, the process described above applies with one difference: because every request is an implicit transaction in Teradata session mode, the following statements are committed:
 - The first two interactive SQL statements
 - Statement_8_1
 - Action statement for the condition handler
 - Statement_8_3
6. End of process.

DECLARE CONDITION

Assign a mnemonic name to an SQLSTATE code, or declare a user-defined condition.

ANSI Compliance

DECLARE CONDITION is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable control declaration.

Stored procedures only.

Syntax

```
DECLARE condition_name CONDITION
  [ FOR SQLSTATE [VALUE] sqlstate_code ] ;
```

Syntax Elements

condition_name

The mnemonic name to be associated with an SQLSTATE code. If you do not specify an SQLSTATE value, the condition name is used to declare a user-defined condition.

sqlstate_code

The five-character literal SQLSTATE code to be handled.

You cannot specify '00000' which represents successful completion of statements.

Usage Notes

The DECLARE CONDITION statement allows you to declare a symbolic name for a condition and optionally associate it with a particular SQLSTATE value. This makes it easier to remember what condition the SQLSTATE value represents.

If the conditions represented by the available SQLSTATE values do not meet your needs, you can also define custom conditions that are specific to your stored procedure.

To declare a user-defined condition, simply declare a condition name without associating it with any SQLSTATE value in the condition declaration. The database will treat *condition_name* as a user-defined condition in this case. You can use the SIGNAL statement with the condition name to explicitly raise the user-defined condition.

The following rules apply to condition declarations:

- A condition name must be a valid identifier. The name can be the same as that of a local variable, parameter, FOR loop alias/column, or label name declared in the stored procedure.
- You can optionally associate a condition name with an SQLSTATE value. For example, the condition name *divide_by_zero* can be associated with the SQLSTATE value '22012'.
- You can declare the same condition name in different nested or non-nested compound statements. You can associate that condition name with the same SQLSTATE value or with a different SQLSTATE value in each of the compound statements. The scoping rules of compound statements allow each declaration to define a different condition.
- You cannot declare the same condition name more than once in the same compound statement. Otherwise, error SPL1080 is reported during stored procedure compilation, and the stored procedure is not created.
- You must declare condition and variable declarations before any other type of declaration in a compound statement.
- You cannot declare more than one condition name to be associated with the same SQLSTATE value in the same compound statement. Otherwise, error SPL1081 is reported during stored procedure compilation, and the stored procedure is not created.
- A handler defined for an SQLSTATE value can also handle any explicitly declared condition associated with that SQLSTATE value.

Example: Using the Same Name for Local Variables, Conditions, and Aliases

The following example illustrates the usage of the same name for local variables, conditions, and aliases as follows:

- *divide_by_zero* is used as a variable and a condition name
- *IOParm1* is used as a parameter and a condition name
- *cs1* is used as a label name and a condition name

```
CREATE PROCEDURE dec1(INOUT IOParm1 INTEGER)
cs1: BEGIN
    DECLARE divide_by_zero INTEGER;
```

```

DECLARE divide_by_zero CONDITION FOR SQLSTATE VALUE '22012';
DECLARE IOParam1 CONDITION;
DECLARE cs1 CONDITION;
DECLARE alias1 CONDITION;
FOR rp1 AS c_rp1
    CURSOR FOR
        SELECT c1 AS alias1 FROM tab1
DO
    SET IOParam1 = rp1.alias1;
END FOR;
...
END cs1;

```

Example: Using Condition Names in Nested Compound Statements

The following example illustrates the usage of condition names in nested compound statements. The condition declarations define the same condition name, *divide_by_zero*, but the declarations are made within different compound statements, and they define the same or different conditions in each compound statement.

```

CREATE PROCEDURE dec2()
cs1: BEGIN
    DECLARE divide_by_zero CONDITION FOR SQLSTATE VALUE '22012';
cs2: BEGIN
    DECLARE divide_by_zero CONDITION;
cs3: BEGIN
    DECLARE divide_by_zero CONDITION FOR SQLSTATE VALUE '22012';
    ...
END cs3;
cs4: BEGIN
    DECLARE divide_by_zero CONDITION FOR SQLSTATE VALUE '42000';
    ...
END cs4;
    ...
END cs2;
    ...
END cs1;

```

Example: Condition Names in a Compound Statement

The following example illustrates the scope of a condition name in a compound statement.

During stored procedure execution, the scope of condition name *exception_cond* declared in line 3 is compound statements cs1 and cs2. The scope of *exception_cond* declared in line 7 is cs2. The definition of *exception_cond* in cs2 overwrites the definition of *exception_cond* in cs1.

The INSERT statement in line 10 raises exception ERRTEQTVNOEXIST (SQLCODE 3807 and SQLSTATE '42000'). Since a CONTINUE handler was defined to handle *exception_cond*, which is associated with SQLSTATE '42000' in cs2, this handler (declared in line 8) is invoked.

The INSERT statement in line 13 also raises exception ERRTEQTVNOEXIST. However, since there is no handler declared to handle SQLSTATE '42000' in cs1 (the containing outer compound statement), the stored procedure terminates with exception ERRTEQTVNOEXIST (SQLCODE 3807 and SQLSTATE '42000').

```

1. CREATE PROCEDURE dec3()
2. cs1: BEGIN
3.     DECLARE exception_cond CONDITION FOR SQLSTATE VALUE '22012';
4.     DECLARE CONTINUE HANDLER FOR exception_cond
5.     ...
6.     cs2: BEGIN
7.         DECLARE exception_cond CONDITION FOR SQLSTATE VALUE '42000';
8.         DECLARE CONTINUE HANDLER FOR exception_cond
9.         ...
10.        INSERT INTO Tab1 VALUES (10); -- Raises exception '42000'
11.        ...
12.    END cs2;
13.    INSERT INTO Tab1 VALUES (10); -- Unhandled exception '42000'
14.    ...
15. END cs1;

```

Example: Using Different Condition Names in a Compound Statement

The following example illustrates the usage of different condition names in a compound statement. The condition names *divide_by_zero* and *balance_too_low* are both declared within the same compound statement.

```

CREATE PROCEDURE dec4()
cs1: BEGIN
    DECLARE divide_by_zero CONDITION FOR SQLSTATE VALUE '22012';
    DECLARE balance_too_low CONDITION;
    ...
END cs1;

```

Example: Exceptions for an SQLSTATE Value

The following example illustrates that a handler defined to handle exceptions related to an SQLSTATE value can also handle explicit conditions associated with that SQLSTATE value. During stored procedure execution, the CONTINUE handler can handle the divide-by-zero condition and any exception with SQLSTATE '22012'.

```

CREATE PROCEDURE dec5()
cs1: BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE VALUE '22012';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '22012'
  ...
  SET IOPar1 = 10/0;
  ...
  SIGNAL divide_by_zero;
  ...
END cs1;

```

Example: Results When Condition Names Are Declared Twice Within the Compound Statement

You cannot use the same condition name more than once in the same compound statement. In the following example, the condition name *divide_by_zero* is declared twice within the compound statement *cs1*. During stored procedure compilation, error SPL1080 will be reported, and the stored procedure will not be created.

```

CREATE PROCEDURE dec6()
cs1: BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE VALUE '22012';
  DECLARE divide_by_zero CONDITION FOR SQLSTATE VALUE '42000';
  ...
END cs1;

```

Example: Results When the Same SQLSTATE Value Is Associated with Different Condition Names

You cannot associate the same SQLSTATE value with different condition names in the same compound statement. In the following example, the SQLSTATE value '22012' is associated with both *divide_by_zero* and *zero_division* condition names in the same compound statement. During stored procedure compilation, error SPL1081 will be reported, and the stored procedure will not be created.

```

CREATE PROCEDURE dec7()
BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
  DECLARE zero_division CONDITION FOR SQLSTATE '22012';
  ...
END;

```

Related Information

- SQLSTATE codes and their meanings, see [SQLSTATE Mappings](#).
- SIGNAL statement, see [SIGNAL](#) Condition name must be a valid identifier, see [Example: Using the Same Name for Local Variables, Conditions, and Aliases](#).

- Associate a condition name with an SQLSTATE value, see [Example: Using the Same Name for Local Variables, Conditions, and Aliases](#).
- Scoping rules of compound statements that allow each declaration to define a different condition, see [Example: Using Condition Names in Nested Compound Statements](#) and [Example: Condition Names in a Compound Statement](#).
- Declaring the same condition name more than once in the same compound statement, see [Example: Results When Condition Names Are Declared Twice Within the Compound Statement](#)
- Declaring the condition and variable declarations before any other type of declaration in a compound statement, see [Example: Using the Same Name for Local Variables, Conditions, and Aliases](#).
- Declaring more than one condition name to be associated with the same SQLSTATE value in the same compound statement, see [Example: Results When the Same SQLSTATE Value Is Associated with Different Condition Names](#).
- A handler defined for an SQLSTATE value that also handles any explicitly declared condition associated with that SQLSTATE value, see [Example: Exceptions for an SQLSTATE Value](#).

DECLARE HANDLER (Basic Syntax)

Associates a condition handler with one or more exception, completion, or user-defined conditions to be handled in a stored procedure.

ANSI Compliance

DECLARE HANDLER is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable control declaration.

Stored procedures only.

Syntax

```
DECLARE { CONTINUE | EXIT } HANDLER FOR
{
  { sqlstate_state_spec | condition_name } [,...] |

  { SQLEXCEPTION | SQLWARNING | NOT FOUND } [,...]

} handler_action_statement ;
```

Syntax Elements

sqlstate_state_spec

```
SQLSTATE [VALUE] sqlstate_code
```

CONTINUE

The type of handler action to be executed.

EXIT

The type of handler action to be executed.

sqlstate_code

The five-character literal SQLSTATE code to be handled.

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.

SQLEXCEPTION

Generic condition to be handled.

You can specify one or any combination of the generic conditions in a comma-separated list.

SQLWARNING

Generic condition to be handled.

You can specify one or any combination of the generic conditions in a comma-separated list.

NOT FOUND

Generic condition to be handled.

You can specify one or any combination of the generic conditions in a comma-separated list.

condition_name

the name of the condition to be handled.

handler_action_statement

either a single statement or multiple statements enclosed in a compound statement that define the handler action.

The handler action is executed when a particular exception or completion condition is returned to the application, or when a user-defined condition is encountered.

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, condition, cursor, or handler) statements are not allowed as a single statement for handler action. These can be submitted from within a compound statement.

Usage Notes

You can specify one of the following in a handler declaration, but not both.

- a list of SQLSTATE values and/or condition names
- a list of generic conditions

You cannot repeat the same condition name, SQLSTATE code, or generic condition in handler declarations within the same compound statement.

You cannot specify both the condition name and the SQLSTATE value associated with the condition name in handler declarations within the same compound statement.

Related Information

- SQLSTATE codes and their meanings, see [SQLSTATE Mappings](#).
- *condition_name*, see [DECLARE CONDITION](#).
- Condition name and the SQLSTATE value associated with the condition name, see [Condition Handler Rules](#).

DECLARE HANDLER (CONTINUE Type)

CONTINUE handlers are useful for handling completion conditions and exception conditions not severe enough to affect the flow of control.

CONTINUE Handler Actions

When a condition is raised, a CONTINUE handler does the following:

1. Executes the handler action.
2. Passes control to the next statement following the statement that invoked it.
3. Executes all remaining SQL statements following the statement that raised the condition.
4. The following table describes the detailed flow of control for a CONTINUE handler when it is activated by a raised exception.

IF ...	THEN in the next stage, control ...
the handler action completes successfully	returns to the statement following the statement that raised the condition.

IF ...	THEN in the next stage, control ...
the exception was raised by a statement embedded within a control statement such as FOR, IF, LOOP, or WHILE	
a control statement raises an exception (for example, while evaluating a conditional expression)	passes to the statement following the control statement that raised the condition.

5. 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.

6. End of process.

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: ANSI Session Mode for DECLARE HANDLER

The following example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI session 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 session mode, the condition is handled using a condition handler.

When the preceding three SQL statements are invoked in ANSI session mode, the following sequence of events describes the impact of an error condition with respect to them:

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.
5. End of process.

When the preceding three SQL statements are invoked in ANSI session mode, the following sequence of events describes the impact of a failure condition with respect to them:

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 session 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 execution continues with the next statement after the handler action completes.
5. End of process.

Example: Teradata Session Mode

The following example assumes that the following three SQL statements are invoked interactively from BTEQ in Teradata session mode. Because the statements are invoked in Teradata session 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 preceding three SQL statements are invoked in Teradata session mode, the following sequence of events occurs:

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.
4. Because the first two BTEQ requests are implicit transactions, their updates are not rolled back.
5. Control passes to Statement_4_3 after the handler action completes.
6. End of process.

Example: Teradata Session Mode Using a BT Statement

This example assumes that the following three SQL statements are invoked interactively from BTEQ in Teradata session mode. 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 preceding three SQL statements are invoked in Teradata session mode, the following sequence of events occurs:

1. The updates made by Statement_4_1, Statement_4_2, and the first three BTEQ requests are all rolled back.

2. The stored procedure statement marked as Statement_4_2 raises an exception with the SQLSTATE indicating a failure condition.
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 executed after the handler action completes.

Note:

Both the handler action and Statement_4_3 are executed as implicit transactions because the effect of the initial BT was revoked when it was rolled back in Stage 2.

5. End of process.

DECLARE HANDLER (EXIT Type)

EXIT handlers deal with conditions that are serious enough to terminate the procedure.

EXIT Handler Actions

When a condition is raised, an EXIT handler does the following:

1. Executes the handler action.
2. Implicitly exits the BEGIN END compound statement in which the handler is declared.
3. 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.
4. The following table describes the detailed flow of control for an EXIT handler when it is activated by a raised exception or completion condition:

IF ...	THEN the next stage in the process is this ...
the handler action completes successfully	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.
the CREATE PROCEDURE statement for this procedure defines INOUT or OUT parameters	the value for ACTIVITY_COUNT in the SUCCESS response is set to 1.
no INOUT or OUT parameters are defined for the procedure	then the value for ACTIVITY_COUNT in the SUCCESS response is set to 0.
the caller is a stored procedure	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.

IF ...	THEN the next stage in the process is this ...
a control statement raises an exception	control exits the compound statement that contains the invoked EXIT handler.

5. If the handler action raises a condition, it is handled if a handler has been defined within the handler action clause.
6. End of process.

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.

Note:

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: ANSI Session Mode for EXIT Handler

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI session 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 sequence of events occurs:

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.

4. End of process.

If an exception condition is reported (that is, a failure condition), the following occurs:

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 exits 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 executed in the EXIT handler that have not been committed.
6. End of process.

Example: Teradata Session Mode for DECLARE HANDLER (EXIT Type)

This example assumes that the following three SQL statements are invoked interactively from BTEQ in Teradata session mode. Because the statements are invoked in Teradata session 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 preceding three SQL statements are invoked in Teradata session mode, the following sequence of events occurs:

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 executed in an implicit transaction, the update made by that statement is not affected.
4. Because the first two BTEQ requests are implicit transactions, their updates are not rolled back.
5. Control exits the calling compound statement and passes to the next statement, if any, after the handler action completes.
6. End of process.

Example: Teradata Session Mode for BT Statement

This example assumes that the following three SQL statements are invoked interactively from BTEQ in Teradata session 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 preceding three SQL statements are invoked in Teradata session mode, the following sequence of events occurs:

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:

The handler action is executed as an implicit transaction because the effect of the initial BT was revoked when it was rolled back in Stage 2.

5. End of process.

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: ANSI Session Mode for EXIT Handler That Contains a COMMIT Statement

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI session 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 sequence of events occurs:

1. The first two BTEQ requests and Statement_6_1 and Statement_6_2 from the stored procedure execute 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 executes within the transaction begun by Statement_6_3.

7. End of process.

If an exception is reported (that is a failure condition), the following occurs:

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 handler action statements initiate a new transaction.
4. The failure condition invokes the EXIT handler defined to handle that specific condition.

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.

5. End of process.

Related Information

[Conditions Raised by a Handler Action](#)

DECLARE HANDLER (SQLEXCEPTION Type)

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.

SQLEXCEPTION Handler Actions

The following table describes the flow of control for an SQLEXCEPTION handler when it is activated by a raised exception:

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 executes 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.
6. End of process.

Example: 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: ANSI Session Mode for DECLARE HANDLER (SQLEXCEPTION Type)

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI session mode:

```

INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
    WHERE Employee_Number = 1000;

CALL spSample11();

```

When the preceding three SQL statements are invoked in ANSI session mode, the following sequence of events occurs:

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

5. End of process.

Example: Behavior of an SQLEXCEPTION and Specific Condition Handlers when Both DECLARE HANDLER Forms Are Combined

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: ANSI Session Mode for Specific and Generic Condition Handlers

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI session mode:

```

INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
WHERE Employee_Number = 1000;

CALL spSample12();
```

When the preceding three SQL statements are invoked in ANSI session mode, the following sequence of events occurs:

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 executes 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.
6. Handler_2 executes 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
8. End of process.

Related Information

[Example: Behavior of an SQLEXCEPTION and Specific Condition Handlers when Both DECLARE HANDLER Forms Are Combined.](#)

DECLARE HANDLER (SQLWARNING Type)

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.

SQLWARNING Handler Actions

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: Generic Condition Handler for DECLARE HANDLER (SQLWARNING Type)

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: ANSI Session Mode for DECLARE HANDLER (SQLWARNING Type)

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI session mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
  WHERE Employee_Number = 1000;

CALL spSample11();
```

When the preceding three SQL statements are invoked in ANSI session mode, the following sequence of events occurs:

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
4. End of process.

Example: Specific and Generic Condition Handlers

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: ANSI Session Mode Calling spSample12();

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI session mode:

```

INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
  WHERE Employee_Number = 1000;

CALL spSample12();

```

When the preceding three SQL statements are invoked in ANSI session mode, the following sequence of events occurs:

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 executes 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

4. End of process.

DECLARE HANDLER (NOT FOUND Type)

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.

NOT FOUND Handler Actions

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: Behavior of a NOT FOUND 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: ANSI Session Mode for DECLARE HANDLER (NOT FOUND Type)

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI session mode:

```
INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
  WHERE Employee_Number = 1000;

CALL spSample11();
```

When the preceding three SQL statements are invoked in ANSI session mode, the following sequence of events occurs:

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
4. End of process.

Example: Specific and Generic Condition Handlers When Both DECLARE HANDLER Forms Are Combined

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: ANSI Session Mode Where Handler_2 Executes Its Handler Action

This example assumes that the following three SQL statements are invoked interactively from BTEQ in ANSI session mode:

```

INSERT INTO Department VALUES ('10', 'Development');

UPDATE Employee SET Salary_Amount = 10000
    WHERE Employee_Number = 1000;

CALL spSample12();

```

When the preceding three SQL statements are invoked in ANSI session mode, the following sequence of events occurs:

1. Handler_2 executes its handler action. Because Handler_2 is an EXIT handler, control passes to the caller after the handler action completes.
2. 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.
3. The following items remain active, but are not committed:
 - The first two interactive SQL statements
 - Statement_12_1

- Action statement for Handler_2

4. End of process.

Diagnostics Area

The Diagnostics Area is a system-managed data structure that contains information about the execution status of the statements within an SQL stored procedure. You can use the GET DIAGNOSTICS statement to extract the information from the Diagnostics Area.

The Diagnostics Area is divided into two components:

- One Statement Area
- Zero, one, or as many as 16 Condition Areas

The Statement Area, sometimes referred to as the Header, contains information about the last statement within the stored procedure.

The Condition Area, sometimes referred to as the Detail Area, contains information about each error, warning, or success code that resulted from the execution of the statement documented in the Statement Area.

The Diagnostics Area is not affected by the following statements:

- BEGIN END
- DECLARE
- GET DIAGNOSTICS
- ITERATE
- LEAVE
- LOOP

The Diagnostics Area is affected by the following statements only for error and warning conditions. These statements first clear the Diagnostics Area and then insert information into it about the error or warning condition raised during the execution of the statement.

- CASE
- FOR
- IF
- REPEAT
- SET
- WHILE

Rules

The following rules apply to the Diagnostics Area:

- Only one Diagnostics Area is associated with each session.
- The Diagnostics Area is emptied (cleared) before the execution of a client-invoked stored procedure.

- The maximum number of conditions that can be stored in the Condition Area of the Diagnostics Area is 16.
- Vantage empties the Diagnostics Area before the execution of all SQL statements except CALL.

After a statement executes, Vantage populates the Statement Area (and, if any conditions are raised, the first Condition Area) with data about the statement and any conditions raised during its execution, respectively.

- Vantage does *not* empty the Diagnostics Area before the execution of a CALL statement within an SQL stored procedure, nor does it modify the Diagnostics Area after the execution of a CALL statement.

The content of the Diagnostics Area remains as it was at the end of the execution of the invoked SQL stored procedure.

- Vantage empties the Diagnostics Area after the execution of any CASE, FOR, IF, REPEAT, SET, or WHILE statement if an exception or completion condition is raised during the execution of that statement.

The system populates the Statement Area and the first Condition Area with data about that statement and the raised condition. The successful completion of any of these statements does not affect the Diagnostics Area.

- If a statement within a handler other than GET DIAGNOSTICS returns an exception or a user-defined condition, and the condition is not dealt with by the handler, then Vantage implicitly submits a RESIGNAL statement.

This action empties the Diagnostics Area, and Vantage restores the original condition with which the handler was invoked to the Diagnostics Area. The system then propagates the condition in Condition Area 1 outside the compound statement containing the handler.

A RESIGNAL statement can add a maximum of 16 Condition Areas to the Diagnostics Area.

The NUMBER option in a RESIGNAL statement area indicates the number of conditions stored in the Diagnostics Area.

If you attempt to store more than 16 conditions in the Diagnostics Area, the value for NUMBER does not increment, and the MORE field in the Statement Area is set to Y.

Structure of the Diagnostics Area

The following table lists the statement information items.

Field	Description	Data Type	Default	Attribute
COMMAND_ FUNCTION_	An identifying text string for the executed SQL statement.	VARCHAR(128) CHARACTER SET LATIN	null	READ_ ONLY
COMMAND_ FUNCTION_ CODE	A number identifying the SQL statement executed. Positive values are reserved for SQL statements defined by the ISO/IEC 9075 SQL standard, while negative	INTEGER	0	READ_ ONLY

Field	Description	Data Type	Default	Attribute
	values are reserved for Teradata-defined SQL statements.			
MORE	<p>A code to indicate whether all the conditions raised during the execution of the SQL statement are stored in the Diagnostics Area or not.</p> <p>Codes are:</p> <ul style="list-style-type: none"> • N, all the conditions that were raised during execution of the SQL statement have been stored in the Diagnostics Area. • Y, more conditions were raised during execution of the SQL statement than there are Condition Areas in the Diagnostics Area. 	CHARACTER(1) CHARACTER SET LATIN	N	READ_ONLY
NUMBER	The number of exception or completion conditions that has been stored in the Diagnostics Area as a result of executing the previous SQL statement (with the exception of a preceding GET DIAGNOSTICS statement).	INTEGER	0	READ_ONLY
ROW_COUNT	The number of rows affected by executing a searched DELETE request, INSERT request, a MERGE request, a searched UPDATE request, or as a direct result of executing the previous SQL statement.	INTEGER	0	READ_ONLY
TRANSACTION_ACTIVE	<p>A code indicating whether the transaction is currently active or not.</p> <p>Codes are:</p> <ul style="list-style-type: none"> • 0, transaction is not currently active. • 1, transaction is currently active. 	INTEGER	0	READ_ONLY

The following table lists the condition information items.

Field	Description	Data Type	Default	Attribute
CLASS_ORIGIN	<p>An identifier for the naming authority that defined the class value of RETURNED_SQLSTATE.</p> <p>Codes are:</p> <ul style="list-style-type: none"> • ISO-9075 (ANSI-defined) • Teradata (Teradata-defined) 	VARCHAR(128) CHARACTER SET UNICODE	null	<p>Modifiable for condition names that are not associated with an SQLSTATE value.</p> <p>For these cases, the value can be any string except <i>ISO-9075</i> or <i>Teradata</i>.</p>

Field	Description	Data Type	Default	Attribute
CONDITION_IDENTIFIER	The condition name specified in a SIGNAL or RESIGNAL statement.	VARCHAR(128) CHARACTER SET UNICODE	null	READ_ONLY
CONDITION_NUMBER	A sequence number to identify each Condition Information Item (detail) area in the Diagnostics Area.	INTEGER	0	READ_ONLY
MESSAGE_LENGTH	The length in characters of the character string value in MESSAGE_TEXT.	INTEGER	0	READ_ONLY
MESSAGE_TEXT	If the value of RETURNED_SQLSTATE corresponds to any of the bulleted items in the following list, then the value for MESSAGE_TEXT is the message text item of the SQL-invoked routine that raised the exception: <ul style="list-style-type: none"> • External routine invocation exception • External routine exception • SQL routine exception • Warning 	VARCHAR(128) CHARACTER SET UNICODE	null	Modifiable
RETURNED_SQLSTATE	The SQLSTATE parameter that would have been returned if this were the only completion or exception condition possible.	CHARACTER(5) CHARACTER SET LATIN	null	READ_ONLY
SUBCLASS_ORIGIN	An identifier for the naming authority that defined the subclass value of RETURNED_SQLSTATE. Codes are: <ul style="list-style-type: none"> • ISO-9075, (ANSI-defined) • Teradata (Teradata-defined) 	VARCHAR(128) CHARACTER SET UNICODE	null	Modifiable for condition names that are not associated with an SQLSTATE value. For these cases, the value can be any string except ISO-9075 or Teradata.

Related Information

- GET DIAGNOSTICS statement, see [GET DIAGNOSTICS](#).
- RESIGNAL statement, see [RESIGNAL](#).
- Command functions and their associated codes, see [Command Function Code Values](#).

Diagnostic Statements

You can use the following statements, all of which are restricted to SQL stored procedures, to insert information into, or retrieve information from, the Diagnostics Area.

- SIGNAL
- RESIGNAL
- GET DIAGNOSTICS

SIGNAL

SIGNAL explicitly raises an exception, completion condition, not including Success, or user-defined condition in the Diagnostics Area.

Invocation

Executable.

Stored procedures only.

Syntax

```
SIGNAL {
  condition_name |
  SQLSTATE [VALUE] SQLSTATE_code
} [ SET condition_information_item = value ] ;
```

Syntax Elements

condition_name

The name of a variable declared to identify a condition within an SQL stored procedure.

If *condition_name* specifies a condition that corresponds to an SQLSTATE value, the value of that SQLSTATE is assigned to RETURNED_SQLSTATE in the Condition Area.

SQLSTATE [VALUE] *SQLSTATE_code*

The value for an SQLSTATE to be assigned to RETURNED_SQLSTATE in the Condition Area.

value

A text or numeric value to be assigned to the specified condition information name.

condition_information_item

One of the following field names from the Condition Area of the Diagnostics Area as shown in the following table.

CLASS_ORIGIN

The identification of the naming authority that defined the class value of RETURNED_SQLSTATE.

The value must be *ISO 9075* if the class value is defined in the ANSI/ISO SQL:2011 standard or *Teradata* if the class value is a Teradata class extension to the SQL:2011 standard.

Data type: VARCHAR(128) CHARACTER SET UNICODE

Default: null

CONDITION_IDENTIFIER

The condition name specified in a SIGNAL or RESIGNAL statement.

Data type: VARCHAR(128) CHARACTER SET UNICODE

Default: null

CONDITION_NUMBER

Takes values from 1 to 16, where 16 is the maximum number of conditions that can be stored in the diagnostics area.

Data type: INTEGER

Default: 0

MESSAGE_LENGTH

The length of MESSAGE_TEXT in characters.

Data type: INTEGER

Default: 0

MESSAGE_TEXT

The text of the Error or Warning message returned by the previous SQL statement execution or a message specified in a SIGNAL or RESIGNAL statement as signal information.

Data type: VARCHAR(128) CHARACTER SET UNICODE

Default: null

RETURNED_SQLSTATE

The SQLSTATE value returned by the previous SQL statement, the SQLSTATE value specified in a SIGNAL or RESIGNAL statement, or the SQLSTATE value associated with the condition name if a condition name is specified in a SIGNAL or RESIGNAL statement.

Data type: CHARACTER(5) CHARACTER SET LATIN

Default: null

SUBCLASS_ORIGIN

The identification of the naming authority that defined the subclass value of RETURNED_SQLSTATE.

The value must be *ISO 9075* if the class value is defined in the ANSI/ISO SQL:2011 standard or *Teradata* if the class value is a Teradata subclass extension to the SQL:2011 standard.

Data type: VARCHAR(128) CHARACTER SET UNICODE

Default: null

Usage Notes

When a SIGNAL statement is executed, the Diagnostics Area is emptied and the Statement Area is filled in with the details of the SIGNAL statement and a Condition Area with condition number 1 is added in the Diagnostics Area corresponding to the SQLSTATE value or condition name specified in the SIGNAL statement. If signal information is specified in the SIGNAL statement, this added condition area is modified with the details given in the signal information.

The following rules apply to SIGNAL:

- If a condition name is specified in a SIGNAL statement, it must be declared in the scope that applies to the SIGNAL statement. Otherwise, error SPL1079 is reported during stored procedure compilation.
- The usage of a condition name in a SIGNAL statement is equivalent to the usage of the SQLSTATE value to which the condition name corresponds. This is true only if the condition name was declared with an SQLSTATE value. illustrates this rule.
- If a SIGNAL statement specifies an exception or user-defined condition and no handler is defined within the compound statement to handle the condition, the rules for the Diagnostics Area and condition handling are same as listed for RESIGNAL and BEGIN END.
- If a SIGNAL statement specifies a completion condition and no handler is defined within the compound statement to handle the condition, the rules for condition handling are the same as those for BEGIN END.
- If a SIGNAL statement within a nonhandler compound statement specifies a user-defined condition, and no handler is defined to handle it in the compound statement or in any outer containing compound statement, the system returns a warning message during the compilation of the procedure. Then at runtime, the SIGNAL statement raises an exception with SQLCODE 7603 and SQLSTATE '45000'.
- If more than one condition declaration is specified for the same condition name, the condition declaration that is most local to the scope of the compound statement containing the SIGNAL statement is used.

The following rules apply to *signal_information*:

- The left hand side of a signal information specification can only stipulate one of the following Statement Area field names:

- CLASS_ORIGIN
- MESSAGE_TEXT
- SUBCLASS_ORIGIN

If a local variable, parameter, or a FOR loop alias/column is specified, Vantage aborts the request during compilation and returns an error to the requestor.

- The left hand side of signal information cannot specify any of the following Statement Area field names:
 - CONDITION_IDENTIFIER
 - CONDITION_NUMBER
 - MESSAGE_LENGTH
 - RETURNED_SQLSTATE

If you specify any of these, Vantage aborts the request during compilation and returns an error to the requestor.

- The left hand side of signal information can only specify the following Statement Area fields if a condition name is specified in a SIGNAL statement, and the specified condition name is not associated with any SQLSTATE value:
 - CLASS_ORIGIN
 - SUBCLASS_ORIGIN

Otherwise, Vantage aborts the request during compilation and returns an error to the requestor.

- You cannot repeat any of the condition information item names in the signal information specification. Otherwise, Vantage aborts the request during compilation and returns an error to the requestor.
- The data type of the value specified in the condition information item of a signal information specification must be compatible with the data type specified for each column in the Condition Area. Otherwise, Vantage aborts the request during compilation and returns an error to the requestor.
- You cannot specify either *ISO 9075* or *Teradata* for CLASS_ORIGIN or SUBCLASS_ORIGIN in the signal information variable.

If you specify the right hand side of the signal information clause for CLASS_ORIGIN or SUBCLASS_ORIGIN as either *ISO 9075* or *Teradata*, Vantage aborts the request during compilation and returns an error to the requestor.

If the right hand side of the signal information clause for CLASS_ORIGIN or SUBCLASS_ORIGIN becomes either *ISO 9075* or *Teradata* during runtime, Vantage aborts the request, returns an error to the requestor, and sets SQLCODE to 7609 and SQLSTATE to 'T7609'.

Contents of the Statement Area

The following table specifies the contents of the Statement Area after the execution of a SIGNAL statement:

Field Name	Contents
COMMAND_FUNCTION	SIGNAL

Field Name	Contents
COMMAND_FUNCTION_CODE	92
NUMBER	1
MORE	N
ROW_COUNT	0
TRANSACTION_ACTIVE	If no transaction is active, TRANSACTION_ACTIVE contains 0 If a transaction is active, TRANSACTION_ACTIVE contains 1

Contents of Condition Area 1

The following table specifies the contents of Condition Area 1 after the execution of a SIGNAL statement:

Field Name	Contents
CLASS_ORIGIN	<p>If SIGNAL specifies:</p> <ul style="list-style-type: none"> An SQLSTATE value or the condition name associated with an SQLSTATE value, CLASS_ORIGIN contains <i>ISO 9075</i> if the class value is defined in the ANSI/ISO SQL standard, or <i>Teradata</i> if the class value is a Teradata extension to the ANSI/ISO SQL standard. A user-defined condition, CLASS_ORIGIN contains the value specified by the <i>signal_information</i> variable. Neither of these, CLASS_ORIGIN contains a null. <p>Data type: VARCHAR(128) CHARACTER SET UNICODE</p>
CLASS_ORIGIN	<p>If SIGNAL specifies an SQLSTATE value or the condition name associated with an SQLSTATE value, then CLASS_ORIGIN is defined in ANSI/ISO SQL standard or Teradata if the class value is a Teradata extension to the ANSI/ISO SQL standard.</p> <p>If SIGNAL specifies a user-defined condition, then CLASS_ORIGIN contains a value specified by the <i>signal_information</i> variable.</p> <p>If SIGNAL specifies neither of these, then CLASS_ORIGIN contains a null.</p> <p>Data type: VARCHAR(128) CHARACTER SET UNICODE</p>
CONDITION_IDENTIFIER	<p>The condition name specified in the SIGNAL statement.</p> <p>If SIGNAL does not specify a condition name, then this field is set null.</p> <p>Data type: VARCHAR(128) CHARACTER SET UNICODE</p>
CONDITION_NUMBER	<p>1</p> <p>Data type: INTEGER</p>
MESSAGE_TEXT	<p>The value specified for MESSAGE_TEXT by the <i>signal_information</i> variable in the SIGNAL statement.</p> <p>If SIGNAL does not specify a message text value, then this field is set null.</p> <p>Data type: VARCHAR(128) CHARACTER SET UNICODE</p>
MESSAGE_LENGTH	<p>The length of the MESSAGE_TEXT in characters.</p> <p>If SIGNAL does not specify a message text value, then this field is set to 0.</p> <p>Data type: INTEGER</p>

Field Name	Contents
RETURNED_SQLSTATE	<p>One of the following:</p> <ul style="list-style-type: none"> The SQLSTATE value associated with the condition specified in the SIGNAL statement. The SQLSTATE value specified in the SIGNAL statement. Null. <p>The value is NULL if a condition name is specified in the SIGNAL statement and it is not associated with any SQLSTATE value.</p> <p>Data type: CHARACTER(5) CHARACTER SET LATIN</p>
CLASS_ORIGIN	<p>If SIGNAL specifies:</p> <ul style="list-style-type: none"> An SQLSTATE value or the condition name associated with an SQLSTATE value, CLASS_ORIGIN contains <i>ISO 9075</i> if the class value is defined in the ANSI/ISO SQL standard, or <i>Teradata</i> if the class value is a Teradata extension to the ANSI/ISO SQL standard. A user-defined condition, CLASS_ORIGIN contains the value specified by the <i>signal_information</i> variable. Neither of these, CLASS_ORIGIN contains a null. <p>Data type: VARCHAR(128) CHARACTER SET UNICODE</p>
SUBCLASS_ORIGIN	<p>If SIGNAL specifies an SQLSTATE value or the condition name associated with an SQLSTATE value, then SUBCLASS_ORIGIN contains <i>ISO 9075</i> if the class value is defined in the ANSI/ISO SQL standard or <i>Teradata</i> if the class value is a Teradata extension to the ANSI/ISO SQL standard.</p> <p>If SIGNAL specifies a user-defined condition, then SUB_CLASS contains the value specified by the <i>signal_information</i> variable.</p> <p>If SIGNAL specifies neither of these, then SUB_CLASS contains a null.</p> <p>Data type: VARCHAR(128) CHARACTER ST UNICODE</p>

Example: Using a Condition Name In a SIGNAL Statement and a Handler Declaration

The following example illustrates the usage of a condition name in a SIGNAL statement and a handler declaration defined for the SQLSTATE value associated with the condition name.

During stored procedure execution via *Req1*, the value of *InParam2* is zero and the SIGNAL statement is executed.

The SIGNAL statement invokes the EXIT handler defined to handle SQLSTATE '22012'. Note that condition name *divide_by_zero* is associated with SQLSTATE '22012'.

Though the SIGNAL statement uses *divide_by_zero* and the handler is defined to handle the SQLSTATE value associated with *divide_by_zero*, the handler for SQLSTATE '22012' is invoked.

After successfully executing the handler action statement, control exits compound statement *cs1* and the stored procedure terminates.

```
CREATE PROCEDURE signalssp3 (IN  InParam1 INTEGER,
                           IN  InParam2 INTEGER,
                           OUT OParam3  INTEGER)
```

```

cs1:BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
  DECLARE EXIT HANDLER FOR SQLSTATE '22012'
    SET OParam3 = 0;
  IF (InParam2 = 0) THEN
    SIGNAL divide_by_zero;
  ELSE
    SET OParam3 = InParam1 + InParam2;
    ...
  END IF;
  ...
END cs1;
BTEQ> CALL signalsp3(10, 0, OParam3);

```

Example: Using a Condition Name in a SIGNAL Statement

The following example illustrates the usage of a condition name in a SIGNAL statement.

During stored procedure execution via *Req1*, the value of *InParam2* is zero and the SIGNAL statement is executed.

The SIGNAL statement looks for a handler defined for *divide_by_zero* or its associated SQLSTATE '22012'.

Because there is an EXIT handler defined to handle *divide_by_zero*, it is invoked.

After successfully executing the handler action statement, control exits compound statement *cs1* and the stored procedure terminates.

```

CREATE PROCEDURE signalsp4 (IN  InParam1 INTEGER,
                           IN  InParam2 INTEGER,
                           OUT OParam3  INTEGER)

cs1:BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
  DECLARE EXIT HANDLER FOR divide_by_zero
    SET OParam3 = 0;
  IF (InParam2 = 0) THEN
    SIGNAL divide_by_zero;
  ELSE
    SET OParam3 = InParam1 + InParam2;
    ...
  END IF;
  ...
END cs1;
BTEQ> CALL signalsp4(10, 0, OParam3);

```

Example: The CONTINUE handler and Exceptions Raised by the SIGNAL Statement

In the following example, the CONTINUE handler defined in the containing outer compound statement handles the exception raised by the SIGNAL statement.

During the stored procedure execution, the SIGNAL statement signals *divide_by_zero*.

Because there is no handler defined to handle *divide_by_zero* in *cs2*, the SQLSTATE value associated with *divide_by_zero* is propagated to outer compound statement *cs1* and is handled by the CONTINUE handler in *cs1*.

After the successful completion of the handler action statement, control returns to the line following SIGNAL *divide_by_zero*.

```
CREATE PROCEDURE signalsp5 (IN  InParam1 INTEGER,
                           IN  InParam2 INTEGER,
                           OUT OParam3  INTEGER)

cs1: BEGIN
    DECLARE CONTINUE HANDLER FOR SQLSTATE '22012'
        SET OParam3=1;
    DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012'
cs2: BEGIN
    IF (InParam2 = 0) THEN
        SIGNAL divide_by_zero;
        ...
    ELSE
        SET OParam3 = InParam1 + InParam2;
        ...
    END IF;
    ...
END cs2;
...
END cs1;
BTEQ> CALL signalsp5(10, 0, OParam3);
```

Example: Using Signal Information In a SIGNAL Statement

The following example illustrates the use of signal information in a SIGNAL statement.

During stored procedure execution, the SIGNAL statement sets MESSAGE_TEXT as '*balance is too low*' and CLASS_ORIGIN as '*Stored Procedure*'.

MESSAGE_LENGTH is implicitly set to 19.

The EXIT handler handles the condition.

The GET DIAGNOSTICS statement retrieves the MESSAGE_TEXT and CLASS_ORIGIN from the first condition area and assigns them to the output parameters, *Message* and *Class*.

After successful execution of the EXIT handler action statements, control exits compound statement *cs1* and the stored procedure terminates.

```
CREATE PROCEDURE setsignalsp1 (INOUT acno    INTEGER,
                              INOUT amt     FLOAT,
                              OUT  Message VARCHAR(50),
                              OUT  Class   VARCHAR(50))

cs1: BEGIN
  DECLARE balance_too_low CONDITION;
  DECLARE count INTEGER DEFAULT 0;
  DECLARE bal_amt, balance FLOAT;
  DECLARE EXIT HANDLER FOR balance_too_low
  BEGIN
    GET DIAGNOSTICS EXCEPTION 1
      Message = MESSAGE_TEXT, Class = CLASS_ORIGIN;
    SET count = count + 1;
    INSERT INTO errortbl VALUES (acno, count, User,
                                current_timestamp, 'Balance too low for the
                                account');

  END;
  SELECT balamt INTO balance
  FROM Deposit
  WHERE accountno = acno;
  SET bal_amt = balance - amt;
  IF (bal_amt < 1000) THEN
    SIGNAL balance_too_low
      SET MESSAGE_TEXT = 'Balance is too low',
      CLASS_ORIGIN = 'Stored Procedure';
  ELSE
    UPDATE Deposit
    SET balance = bal_amt
    WHERE accountno = acno;
  END IF;
END cs1;
```

Example: The Condition Declaration and the SIGNAL Statement

In the following example, the condition declaration that is more local to the scope of the compound statement containing the SIGNAL statement gets used.

During stored procedure execution, the value of *InParam2* is zero and the SIGNAL statement in compound statement *cs2* raises a user-defined condition *divide_by_zero*.

Because there is a handler for this condition in compound statement *cs2*, it is invoked. Even though compound statement *cs1* has a handler defined to handle *divide_by_zero*, it is not invoked because

the condition declaration is more local to the scope of the compound statement containing the SIGNAL statement.

Stored procedure execution continues after executing the handler action statement.

```
CREATE PROCEDURE signalsp7 (IN  InParam1 INTEGER,
                           IN  InParam2 INTEGER,
                           OUT OParam3  INTEGER)

cs1: BEGIN
  DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
  DECLARE EXIT HANDLER FOR divide_by_zero
    SET OParam3 = 0;
  cs2: BEGIN
    DECLARE divide_by_zero CONDITION FOR SQLSTATE '22012';
    DECLARE EXIT HANDLER FOR divide_by_zero
      SET OParam3 = 10;
    IF (InParam2 = 0) THEN
      SIGNAL divide_by_zero;
    ELSE
      SET OParam3 = InParam1 + InParam2;
    ...
    END IF;
    ...
  END cs2;
  ...
END cs1;
BTEQ> CALL signalsp7 (10, 0, OParam3);
```

Related Information

- *condition_name*, see [DECLARE CONDITION](#).
- SQLSTATE codes and their meanings, see [SQLSTATE Mappings](#).
- The Diagnostics Area, see [Diagnostics Area](#).
- Condition name is specified in a SIGNAL statement, it must be declared in the scope that applies to the SIGNAL statement, see [Example: Using a Condition Name in a SIGNAL Statement](#).
- Condition name in a SIGNAL statement is equivalent to the usage of the SQLSTATE value to which the condition name corresponds, see [Example: Using a Condition Name In a SIGNAL Statement and a Handler Declaration](#).
- Rules for the Diagnostics Area and condition handling, see [RESIGNAL](#) and [BEGIN END SIGNAL](#) statement specifies a completion condition and no handler is defined within the compound statement to handle the condition, see [BEGIN END](#).
- More than one condition declaration is specified for the same condition name, see [Example: The Condition Declaration and the SIGNAL Statement](#)

- Data type of the value specified in the condition information item of a signal information specification [Structure of the Diagnostics Area](#).

RESIGNAL

RESIGNAL resignals or invokes a condition from a handler declaration. The RESIGNAL statement can be specified explicitly only in a handler declaration.

Invocation

Executable.

Stored procedures only.

Syntax

```
RESIGNAL [ condition_name | SQLSTATE [VALUE] SQLSTATE_code ]
[ SET condition_information_item = value ] ;
```

Syntax Elements

condition_name

The name of a variable declared to identify a condition within an SQL stored procedure.

If *condition_name* specifies a condition that corresponds to an SQLSTATE value, the value of that SQLSTATE is assigned to RETURNED_SQLSTATE in the Condition Area.

SQLSTATE [VALUE] *SQLSTATE_code*

The value for an SQLSTATE to be assigned to RETURNED_SQLSTATE in the Condition Area.

value

A text or numeric value to be assigned to the specified condition information name.

condition_information_item

One of the following field names from the Condition Area of the Diagnostics Area as shown in the following table.

CLASS_ORIGIN

The identification of the naming authority that defined the class value of RETURNED_SQLSTATE.

The value must be *ISO 9075* if the class value is defined in the ANSI/ISO SQL:2011 standard or *Teradata* if the class value is a Teradata class extension to the SQL:2011 standard.

Data type: VARCHAR(128) CHARACTER SET UNICODE

Default: null

CONDITION_IDENTIFIER

The condition name specified in a SIGNAL or RESIGNAL statement.

Data type: VARCHAR(128) CHARACTER SET UNICODE

Default: null

CONDITION_NUMBER

Takes values from 1 to 16, where 16 is the maximum number of conditions that can be stored in the diagnostics area.

Data type: INTEGER

Default: 0

MESSAGE_LENGTH

The length of MESSAGE_TEXT in characters.

Data type: INTEGER

Default: 0

MESSAGE_TEXT

The text of the Error or Warning message returned by the previous SQL statement execution or a message specified in a SIGNAL or RESIGNAL statement as signal information.

Data type: VARCHAR(128) CHARACTER SET UNICODE

Default: null

RETURNED_SQLSTATE

The SQLSTATE value returned by the previous SQL statement, the SQLSTATE value specified in a SIGNAL or RESIGNAL statement, or the SQLSTATE value associated with the condition name if a condition name is specified in a SIGNAL or RESIGNAL statement.

Data type: CHARACTER(5) CHARACTER SET LATIN

Default: null

SUBCLASS_ORIGIN

The identification of the naming authority that defined the subclass value of RETURNED_SQLSTATE.

The value must be *ISO 9075* if the class value is defined in the ANSI/ISO SQL:2011 standard or *Teradata* if the class value is a Teradata subclass extension to the SQL:2011 standard.

Data type: VARCHAR(128) CHARACTER SET UNICODE

Default: null

Usage Notes

RESIGNAL always propagates a condition outward. When a RESIGNAL statement is submitted from a handler action, the outer containing compound statements are searched for the most appropriate condition handler declaration. A RESIGNAL statement with a signal value does not clear the diagnostics area.

The following rules apply to RESIGNAL:

- If a RESIGNAL statement is used outside a condition handler, the request aborts during stored procedure compilation and returns an error to the requestor.
- If a condition name is specified in a RESIGNAL statement the condition name must be declared in the scope that applies to the handler containing the RESIGNAL statement. Otherwise, the request aborts during stored procedure compilation and returns an error to the requestor.
- The usage of a condition name in a RESIGNAL statement is equivalent to using the SQLSTATE value to which the condition name corresponds if the condition name is associated with an SQLSTATE value.
- If more than one condition declaration is specified for the same condition name, the one that is the most local to the scope of the compound statement containing the RESIGNAL statement is used.
- If a signal value is specified in a RESIGNAL statement, the Statement Area is modified with the details of the RESIGNAL statement and the existing Condition Areas, if any, are stacked such that the n^{th} Condition Area is placed at the position of the $(n+1)^{\text{th}}$ condition area in the Diagnostics Area. If signal information is specified in the RESIGNAL statement, Condition Area 1 is modified with the details given in the signal information before pushing down the existing Condition Areas. A new Condition Area 1 corresponding to the signal value is added to the Diagnostics Area.
 - If the signal value is a condition name, CONDITION_IDENTIFIER in Condition Area 1 is set to contain condition name. If condition name is associated with an SQLSTATE value, RETURNED_SQLSTATE in Condition Area 1 is set to contain this SQLSTATE value.
 - Otherwise, if the signal value is an SQLSTATE value, RETURNED_SQLSTATE in Condition Area 1 is set to contain this SQLSTATE value.
 - A handler is searched in a containing outer compound statement to handle the condition raised by the RESIGNAL statement.
- If there is no containing outer compound statement that has a handler to handle the condition raised by the RESIGNAL statement, one of the following happens:
 - An exception: The stored procedure exits with the exception condition raised by the RESIGNAL statement.
 - A completion condition: The execution continues from the statement following the RESIGNAL statement in the handler containing the RESIGNAL statement.
 - A user-defined exception condition: An exception that sets SQLCODE to 7603 and SQLSTATE to '45000' is raised.

- If there is a handler in a containing outer compound statement that can handle the condition raised by the RESIGNAL statement, one of the following happens:
 - CONTINUE handler: Stored procedure execution continues from the statement following the statement that invoked the handler containing the RESIGNAL statement.
 - EXIT handler: Stored procedure execution continues from the statement following the END compound statement containing the handler whose action clause has just completed successfully.
- If a RESIGNAL statement is submitted without any signal value, one of the following things happens:
 - The Diagnostics Area is cleared and its original contents, with which the handler containing the RESIGNAL statement was invoked, are restored in the Diagnostics Area.
The CONDITION_IDENTIFIER and RETURNED_SQLSTATE in Condition Area 1 reflect the original condition with which the handler was invoked.
 - If signal information is specified, Condition Area 1 is modified with the details given in the signal information specification of the RESIGNAL statement.
 - The original condition with which the handler was invoked is propagated outward and the containing outer compound statements are searched for a handler for this condition.
- If a RESIGNAL statement uses a user-defined condition, and no handler declaration is defined to handle the condition in the scope of the compound statement containing the RESIGNAL statement, Vantage reports a warning during compilation.

Rules specified in “SIGNAL” for signal information also apply to the signal information in a RESIGNAL statement.

Contents of the Diagnostics Area

The following table specifies the contents of the Statement Area after the execution of a RESIGNAL statement specified with a signal value:

Field	Value
COMMAND_FUNCTION	RESIGNAL
COMMAND_FUNCTION_CODE	91
NUMBER	If NUMBER < 16, increment by 1. If NUMBER ≤ 16, value not changed.
MORE	Y if the value of NUMBER is changed. N if the value of NUMBER is not changed.
ROW_COUNT	0
TRANSACTION_ACTIVE	0 if no transaction is active. 1 if a transaction is active.

The following table specifies the contents of Condition Area 1 after the execution of a RESIGNAL statement specified with signal information. If a signal value is also specified in the RESIGNAL statement, this Condition Area is pushed to condition number 2.

Field	Value
CLASS_ORIGIN	If CLASS_ORIGIN is specified in signal information, this field contains the specified value. Otherwise, the existing value is retained.
CONDITION_IDENTIFIER	The existing value is retained because this field cannot be modified.
CONDITION_NUMBER	The existing value is retained because this field cannot be modified.
MESSAGE_TEXT	The value specified for signal information in a RESIGNAL statement. Otherwise, the existing value is retained.
MESSAGE_LENGTH	The length of the MESSAGE_TEXT If RESIGNAL does not specify a message text value, then this field is set to 0.
RETURNED_SQLSTATE	The existing value is retained because this field cannot be modified.
SUBCLASS_ORIGIN	If SUBCLASS_ORIGIN is specified in signal information, this field contains the specified value. Otherwise, the existing value is retained.

The following table specifies the contents of Condition Area 1 after the execution of a RESIGNAL statement specified with a signal value:

Field	Value
CLASS_ORIGIN	If the RESIGNAL statement specifies: <ul style="list-style-type: none"> The SQLSTATE value associated with the condition specified in the RESIGNAL statement The SQLSTATE value specified in the RESIGNAL statement: then the content of CLASS_ORIGIN is determined as follows <ul style="list-style-type: none"> If the class value is defined by the ANSI/ISO SQL standard, then CLASS_ORIGIN is ISO 9075. If the class value is defined by Teradata, then CLASS_ORIGIN is Teradata. Data type: VARCHAR(128) CHARACTER SET UNICODE. If the RESIGNAL statement specifies a user-defined condition, CLASS_ORIGIN contains a null.
CONDITION_IDENTIFIER	The condition name specified in the RESIGNAL statement. Data type: VARCHAR(128) CHARACTER SET UNICODE. If no condition name is specified, then CONDITION_IDENTIFIER contains a null.
CONDITION_NUMBER	1 Data type: INTEGER

Field	Value
MESSAGE_ TEXT	null Data type: VARCHAR(128) CHARACTER SET UNICODE
MESSAGE_ LENGTH	0 Data type: INTEGER
RETURNED_ SQLSTATE	One of the following: <ul style="list-style-type: none"> • The SQLSTATE value associated with the condition specified in the RESIGNAL statement • The SQLSTATE value specified in the RESIGNAL statement • Null Data type: CHARACTER(5) CHARACTER SET LATIN
SUBCLASS_ ORIGIN	If the RESIGNAL statement specifies: <ul style="list-style-type: none"> • The SQLSTATE value associated with the condition specified in the RESIGNAL statement • The SQLSTATE value specified in the RESIGNAL statement then the content of SUBCLASS_ORIGIN is determined as follows: <ul style="list-style-type: none"> • If the class value is defined by the ANSI/ISO SQL standard, then CLASS_ORIGIN is ISO 9075. • If the class value is defined by Teradata, then CLASS_ORIGIN is Teradata. Data type: VARCHAR(128) CHARACTER SET UNICODE. If the RESIGNAL statement specifies a user-defined condition, SUBCLASS_ORIGIN contains a null.

After the execution of RESIGNAL statement that does not specify a signal value, Vantage sets the contents of the Diagnostics Area as follows:

- The Statement Area is set to the contents of the Statement Area of the Diagnostics Area with which the handler containing the RESIGNAL statement was invoked.
- The Condition Areas are set to the Condition Areas of the Diagnostics Area with which the handler containing the RESIGNAL statement was invoked.

Example: SET Statement Exceptions

At runtime in this example, the last SET statement in the procedure definition raises an exception that returns SQLCODE 2802 and SQLSTATE '22012'.

The CONTINUE handler defined to handle SQLSTATE '22012' is invoked and the RESIGNAL statement is executed.

Because the condition *out_of_range* is associated with SQLSTATE '22003', SQLSTATE '22003' is propagated to compound statement *cs1* and the EXIT handler is invoked.

After successfully executing the handler action statements, control exits *cs1* and the stored procedure terminates successfully.

```

CREATE PROCEDURE resignalsp3 (INOUT IOParam INTEGER,
                             OUT  OParam INTEGER)
cs1: BEGIN
  DECLARE out_of_range CONDITION FOR SQLSTATE '22003';
  DECLARE EXIT HANDLER FOR SQLSTATE '22003'
    SET OParam = 0;
  cs2: BEGIN
    DECLARE CONTINUE HANDLER FOR SQLSTATE '22012'
      RESIGNAL out_of_range;
    SET IOParam = 0;
    SET OParam = 20 / IOParam;
  END cs2;
END cs1;
BTEQ> CALL resignalsp3 (0, OParam);

```

Example: The CONTINUE Handler and the RESIGNAL Statement

The following example illustrates the return point when a CONTINUE handler in an outer compound statement handles the condition raised by a RESIGNAL statement in the inner compound statement.

At runtime, the SET statement raises exception SQLSTATE '22012'.

The CONTINUE handler is invoked and the RESIGNAL statement is executed as part of the handler action statements.

The RESIGNAL statement references the condition declared in *cs2* because the condition declaration is more local to the scope of the compound statement containing the RESIGNAL statement.

Because *Condition1* is associated with SQLSTATE '21000', the exception condition is propagated to the containing outer compound statement *cs1* and the CONTINUE handler for SQLSTATE '21000' is invoked.

After successfully executing the handler action statement, control returns to the statement following the last SET statement in the procedure definition.

```

CREATE PROCEDURE resignalsp5(INOUT IOParam INTEGER,
                             OUT  OParam INTEGER)
cs1:BEGIN
  DECLARE Condition1 CONDITION FOR SQLSTATE '22012';
  DECLARE EXIT HANDLER FOR Condition1
    SET OParam = 0;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '21000'
    SET OParam = 100;
  cs2:BEGIN
    DECLARE Condition1 CONDITION FOR SQLSTATE '21000';
    DECLARE CONTINUE HANDLER FOR SQLSTATE '22012'
      BEGIN
        RESIGNAL Condition1;
      END
  END cs2;
END cs1;

```

```

        ...
    END;
    SET IOParam = 0;
    SET OParam = 20 / IOParam;
    ...
END cs2;
END cs1;
BTEQ> CALL resignalssp5 (0, OParam);

```

Example: Using a Signal Value and Signal Information in a RESIGNAL Statement

The following example illustrates the usage of a signal value and signal information in a RESIGNAL statement.

During the stored procedure execution, the RESIGNAL statement updates MESSAGE_TEXT in Condition Area 1 with *'Sampling error'* and MESSAGE_LENGTH is implicitly set to 14.

Because a signal value is specified in the RESIGNAL statement, the existing Condition Areas of the diagnostics area are pushed down and a new Condition Area is added in the diagnostics area with RETURNED_SQLSTATE as 'T7473' and CONDITION_NUMBER as 1.

```

CREATE PROCEDURE setsignalssp3(OUT OPar CHAR(100))
cs1: BEGIN
    ...
    cs2: BEGIN
        DECLARE samp_error CONDITION;
        DECLARE CONTINUE HANDLER FOR samp_error
            RESIGNAL SQLSTATE 'T7473'
            SET MESSAGE_TEXT = 'Sampling error';
        ...
        SIGNAL samp_error;
    ...
    END cs2;
END cs1;
BTEQ> .COMPILE FILE setsignalssp3.sp1
BTEQ> CALL setsignalssp3(OPar);

```

The following example illustrates how to propagate the original exception outwards from a handler. Example: Propagating the Original Exception Outwards From a Handler

During stored procedure execution, the last INSERT statement in the procedure definition raises a duplicate row exception, and the handler declared for SQLSTATE '23505' is invoked.

The handler action statement that inserts SQLSTATE, CURRENT_TIMESTAMP, 'spSample1', and 'Failed to Insert record' results in another exception with SQLSTATE '42000' and activates the generic SQLEXCEPTION handler.

The handler issues a RESIGNAL statement with the original exception that the handler was supposed to handle.

The Diagnostics Area is cleared and restored to the original state at the time the handler was invoked. This causes original exception SQLSTATE '42000' to be restored in Condition Area 1.

Because signal information is specified in the RESIGNAL statement, MESSAGE_TEXT in Condition Area 1 is modified with 'Table does not exist'.

MESSAGE_LENGTH is implicitly set to 20. The initial condition handler in the procedure definition is then invoked to handle exception SQLSTATE '42000'.

After the successful completion of the handler action statements, control returns to the statement following the INSERT handler action statement.

```
CREATE PROCEDURE spSample3(IN pName      CHAR(30),
                           IN pAmt      INTEGER,
                           Osqlstate CHAR(5),
                           Omsg        CHAR(30))
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '42000'
    GET DIAGNOSTICS EXCEPTION 1
      Osqlstate = RETURNED_SQLSTATE, Omsg = MESSAGE_TEXT;
  L1:BEGIN
    DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
      BEGIN
        DECLARE EXIT HANDLER FOR SQLEXCEPTION
          BEGIN
            DELETE FROM tab1;
            RESIGNAL SET MESSAGE_TEXT = 'Table does not exist';
          END;
        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 L1;
  ...
END;
BTEQ> .COMPILE FILE nblk3.sp1
BTEQ> CREATE SET TABLE tab1(c1 CHAR(30), c2 INTEGER);
BTEQ> DROP TABLE Proc_Error_Tbl;
BTEQ> CALL spSample3('Richard', 100, OSqlstate, OMsg);
```

Example: How a Calling Stored Procedure Handles the Condition Returned By a Called Stored Procedure

The following illustrates how a calling stored procedure handles the condition returned by a called stored procedure terminated with an exception condition.

During the execution of outer stored procedure *resignalsp1* Example:, the CALL statement invokes the inner stored procedure *resignalsp2*.

In *resignalsp2*, the SIGNAL statement raises completion condition SQLSTATE '02000'.

The SIGNAL statement first clears the diagnostics area. Then the Statement Area is updated and a Condition Area 1 is added with a RETURNED_SQLSTATE '02000'. The CONTINUE HANDLER is invoked for this condition.

The RESIGNAL statement raises user-defined condition *nodata*.

The Statement Area is updated with the details of the RESIGNAL statement.

Condition Area 1 is pushed down in the Diagnostics Area as Condition Area 2, and a new Condition Area 1 is added corresponding to *nodata*.

There is no handler for this user-defined condition and *resignalsp2* is terminated with exception condition ERRRTSNOCOND (SQLCODE 7603 and SQLSTATE '45000').

In *resignalsp1*, the CALL statement reports exception ERRRTSNOCOND that is handled by the condition handler declared to handle SQLSTATE '45000' in *cs2*.

The GET DIAGNOSTICS statement retrieves Condition Area 1 from the Diagnostics Area and assigns '*nodata*' to *condname*.

The RESIGNAL statement raises user-defined condition *nodata* that is handled by the handler defined for *nodata* in the inner procedure.

The CALL statement returns OParam1 = 0, pcondno = 1, and count = 0.

The outer stored procedure is defined as follows:

```
CREATE PROCEDURE resignalsp1 (OUT  OParam1 INTEGER,
                             INOUT pcondno INTEGER,
                             OUT  count  INTEGER)

cs1 :BEGIN
    DECLARE nodata CONDITION;
    DECLARE cnt INTEGER DEFAULT VALUE 0;
    DECLARE CONTINUE HANDLER FOR nodata
        SET count = 0;
    cs2:BEGIN
        DECLARE CONTINUE HANDLER FOR SQLSTATE '45000'
        BEGIN
            GET DIAGNOSTICS EXCEPTION pcondno
                condname = CONDITION_IDENTIFIER;
            IF (condname = 'nodata') THEN
```

```

        RESIGNAL 'nodata';
    END IF;
END;
SET OParam1 = 0;
CALL resignalssp2(cnt); /* returns exception '45000' */
END cs2;
END cs1;

```

The inner stored procedure is defined as follows:

```

CREATE PROCEDURE resignalssp2 (OUT OParam1 INTEGER)
cs1 :BEGIN
    DECLARE cnt INTEGER DEFAULT VALUE 0;
    cs2:BEGIN
        DECLARE nodata CONDITION;
        DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
            RESIGNAL nodata;
        SET OParam1 = 0;
        SELECT COUNT(*) INTO cnt FROM tab1;
        IF (cnt = 0) THEN
            SIGNAL SQLSTATE '02000';
            ...
        ELSE
            SET OParam1 = cnt;
        END IF;
    END cs2;
END cs1;
BTEQ> CREATE SET TABLE tab1 (c1 INTEGER);
BTEQ> CALL resignalssp1(OParam1, 1, condname);

```

Example: Retrieving Information in a RESIGNAL Statement

The following example illustrates the retrieval of information corresponding to Condition Area 2 when a user-defined condition is specified in a RESIGNAL statement.

During the stored procedure execution, the SELECT INTO statement raises completion condition SQLSTATE '02000'.

The Statement Area is filled with the details of the SELECT INTO statement and a Condition Area is added to the Diagnostics Area with information related to the completion condition.

The handler for SQLSTATE '02000' is invoked to handle completion condition SQLSTATE '02000'.

The RESIGNAL statement within the handler action raises user-defined condition *nodata*.

Condition Area 1 is pushed down in the Diagnostics Area to Condition Area 2, and a new Condition Area 1 is added to the Diagnostics Area corresponding to the user-defined condition *nodata*.

User-defined condition *nodata* is propagated to the outer compound statement.

The nodata handler in compound statement *cs1* handles user-defined condition *nodata*.

The first GET DIAGNOSTICS statement retrieves Condition Area 1 and assigns 'nodata' from CONDITION_IDENTIFIER to *condid*.

The second GET DIAGNOSTICS statement retrieves Condition Area 2 and assigns SQLSTATE '02000' from RETURNED_SQLSTATE to *sqlstate1*.

Stored procedure execution resumes after the operations preceding END cs2. The CALL statement returns OParam1 = 0, pcondno = 2, and sqlstate1 = '02000'.

```
CREATE PROCEDURE resig6 (OUT  OParam1 INTEGER,
                        INOUT pcondno INTEGER,
                        OUT   sqlstate1 CHAR(5),
                        OUT   condid CHAR(10))

cs1 :BEGIN
  DECLARE nodata CONDITION;
  DECLARE CONTINUE HANDLER FOR nodata
  BEGIN
    GET DIAGNOSTICS EXCEPTION 1
      condid = CONDITION_IDENTIFIER;
    GET DIAGNOSTICS EXCEPTION pcondno
      sqlstate1 = RETURNED_SQLSTATE;
  END;
  cs2: BEGIN
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
    BEGIN
      RESIGNAL nodata;
      ...
    END;
    SET OParam1 = 0;
    SELECT c1
    INTO OParam1 FROM tab1; -- Returns warning  NO DATA FOUND
    ...
  END cs2;
END cs1;
BTEQ> CREATE SET TABLE  tab1 (c1 INTEGER);
BTEQ> CALL resig6(OParam1,2,sqlstate1, condid);
```

Example: Pushing Out a Condition When the Diagnostics Area is Full

The following example illustrates that when the Diagnostics Area is full, Condition Area 16 is pushed out to accommodate another condition. An SQLSTATE is specified in a RESIGNAL statement.

During the stored procedure execution, the SELECT INTO statement in *cs16* raises completion condition SQLSTATE '02000'.

The Statement Area is filled in with the details of the SELECT INTO statement and a Condition Area 1 is added in the Diagnostics Area.

The handler in *cs16* is invoked for completion condition SQLSTATE '02000'. The RESIGNAL statement in *cs16* raises condition '23505'.

The existing Condition Area is pushed down in the Diagnostics Area its condition number is incremented by 1.

The Statement Area is updated and a new condition area with Condition Number 1 is added in the Diagnostics Area.

The condition is handled by the handler defined in *cs15*. The RESIGNAL statement in *cs15* raises condition SQLSTATE '23505'.

The existing Condition Areas are pushed down in the Diagnostics Area. Condition Area 2 becomes Condition Area 3, and a new Condition Area 1 is added for condition SQLSTATE '23505'. This shifting of Condition Areas and adding of a new Condition Area happens for all the RESIGNAL statements.

Finally, the RESIGNAL statement in *cs2* raises condition SQLSTATE '23505', a new Condition Area 1 is added at the top of the Diagnostics Area, and all other Condition Areas are pushed down by one position.

The condition is handled by the handler in *cs1* and the RESIGNAL statement in *cs1* raises condition SQLSTATE '23505'.

Now the total number of conditions in the Diagnostics Area has reached 16, the limit on the maximum number of Condition Areas that can be stored there.

Condition Area 16 is moved out of the Diagnostics Area, all other Condition Areas are pushed down by 1 position, and the new condition is added as Condition Area 1.

NUMBER in the Statement Area remains at 16 and MORE is set to Y. The condition raised is handled by the CONTINUE handler in *cs0*.

The GET DIAGNOSTICS statement in *cs0* retrieves RETURNED_SQLSTATE '23505' from Condition Area 16 and assigns it to *sqlstate1*. Because the handler type is CONTINUE, the stored procedure continues at the SET statement in *cs16*.

The CALL statement returns OParam1 = 0 and sqlstate1 = '23505'.

```
CREATE PROCEDURE resig6 (OUT OParam1 INTEGER,
                        OUT sqlstate1 CHAR(5))
cs0: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
GET DIAGNOSTICS EXCEPTION 16
sqlstate1 = RETURNED_SQLSTATE;
cs1: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
RESIGNAL SQLSTATE '23505';
```

```

cs2: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    RESIGNAL SQLSTATE '23505';
cs3: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    RESIGNAL SQLSTATE '23505';
cs4: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    RESIGNAL SQLSTATE '23505';
cs5: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    RESIGNAL SQLSTATE '23505';
cs6: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    RESIGNAL SQLSTATE '23505';
cs7: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    RESIGNAL SQLSTATE '23505';
cs8: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    RESIGNAL SQLSTATE '23505';
cs9: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    RESIGNAL SQLSTATE '23505';
cs10: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    RESIGNAL SQLSTATE '23505';
cs11: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    RESIGNAL SQLSTATE '23505';
cs12: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    RESIGNAL SQLSTATE '23505';
cs13: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    RESIGNAL SQLSTATE '23505';
cs14: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    RESIGNAL SQLSTATE '23505';
cs15: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    RESIGNAL SQLSTATE '23505';
cs16: BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'

```

```

        RESIGNAL SQLSTATE '23505';
        SELECT c1 INTO OParam1 from tab1;
        -- Returns warning NO DATA FOUND
        SET OParam1 = 0;
        END cs16;
    END cs15;
END cs14;
END cs13;
END cs12;
END cs11;
END cs10;
END cs9;
END cs8;
END cs7;
END cs6;
END cs5;
END cs4;
END cs3;
END cs2;
END cs1;
END cs0;
BTEQ> CREATE SET TABLE tab1 (c1 INTEGER);
BTEQ> CALL resig6(OParam1, sqlstate1);

```

Related Information

- *condition_name*, see [DECLARE CONDITION](#).
- SQLSTATE codes and their meanings, see [SQLSTATE Mappings](#).
- Using a condition name in a RESIGNAL statement, see [Example: SET Statement Exceptions](#).
- More than one condition declaration is specified for the same condition name, see [Example: The CONTINUE Handler and the RESIGNAL Statement](#).
- Signal value is specified in a RESIGNAL statement, see [Example: Using a Signal Value and Signal Information in a RESIGNAL Statement](#).
- CONDITION_IDENTIFIER, see [Example: Retrieving Information in a RESIGNAL Statement](#).
- Signal value is an SQLSTATE value, see [Example: Pushing Out a Condition When the Diagnostics Area is Full](#).
- Handler searched in a containing outer compound statement, see [Example: Retrieving Information in a RESIGNAL Statement](#) and [Example: Pushing Out a Condition When the Diagnostics Area is Full](#).
- CONTINUE handler, see [Example: The CONTINUE Handler and the RESIGNAL Statement](#).
- Signal information is specified, .
- Rules specified in SIGNAL, see [SIGNAL](#).

GET DIAGNOSTICS

GET DIAGNOSTICS retrieves information about successful, exception, or completion conditions from the Diagnostics Area.

Invocation

Executable.

Stored procedures only.

Syntax

```
GET DIAGNOSTICS [ EXCEPTION condition_number ] diagnostic_spec [, ...] ;
```

Syntax Elements

diagnostic_spec

```
{ parameter_name | variable_name } = statement_information_item
```

parameter_name

A parameter whose value is set to the value contained in *statement_information_item*.

variable_name

A variable whose value is set to the value contained in *statement_information_item*.

EXCEPTION

A language element that indicates to return information from the Condition Area of the Diagnostics Area.

condition_number

A number, parameter, or variable that resolves to the number of the Condition Area from which information is to be retrieved.

parameter_name

An output parameter to which the *condition_information_item* retrieved from the specified Condition Area is assigned.

variable_name

An output variable to which the *condition_information_item* retrieved from the specified Condition Area is assigned.

condition_information_item

The name of the Condition Area field from which condition information is to be retrieved.

statement_information_item

One of the following field names from the Statement Area of the Diagnostics Area as shown in the following table.

COMMAND_FUNCTION

An identifying text string for the executed SQL statement.

Data type: VARCHAR(128) CHARACTER SET LATIN

Default: null

COMMAND_FUNCTION_CODE

A number that uniquely identifies each command function.

Data type: INTEGER

Default: 0

MORE

A code that indicates whether all the conditions raised during the execution of the SQL statement are stored in the Diagnostics Area or not.

Data type: CHARACTER(1) CHARACTER SET LATIN

Default: N

N means that all conditions raised during SQL statement execution are stored in the Diagnostics Area.

NUMBER

The number of exception of completion conditions that has been stored in the Diagnostics Area as a result of executing the previous SQL statement.

Data type: INTEGER

Default: 0

ROW_COUNT

The number of rows affected by executing a searched DELETE request, an INSERT request, a MERGE request, or a searched UPDATE request; or as a direct result of executing the previous SQL statement.

Data type: INTEGER

Default: 0

TRANSACTION_ACTIVE

A code that indicates whether the transaction is currently active or not.

Data type: INTEGER

Default: 0

0 means the transaction is not currently active.

Usage Notes

If you specify a statement information item in a GET DIAGNOSTICS statement, Vantage retrieves the requested information from the Statement Area into the simple target specification.

If you specify an EXCEPTION in the GET DIAGNOSTICS statement, Vantage retrieves the requested condition information item from the Condition Area corresponding to the condition number from the Diagnostics Area into the simple target specification.

GET DIAGNOSTICS statements do *not* change the contents of the Diagnostics Area. If GET DIAGNOSTICS raises an exception condition, only the status variables SQLSTATE, SQLCODE, and ACTIVITY_COUNT are set.

The following rules apply to GET DIAGNOSTICS.

If a GET DIAGNOSTICS statement specifies an EXCEPTION and the value of the condition number is any of the following constants, the statement aborts during compilation and returns an error.

- NULL
- A value < 1
- A value > 16, which is the maximum number of Condition Areas that can be stored in the Diagnostics Area.

If a GET DIAGNOSTICS statement specifies an EXCEPTION and the value of the condition number is any of the following, the statement aborts at runtime and returns an error:

- NULL
- A value < 1
- A value > the number of conditions stored in the Diagnostics Area when the GET DIAGNOSTICS statement is executed

If a GET DIAGNOSTICS statement specifies an EXCEPTION and does not violate either of the previous rules, Vantage retrieves the information from the Condition Area with the specified condition number.

The right hand side of the statement information item in a GET DIAGNOSTICS statement must specify one of the following Statement Area field names:

- COMMAND_FUNCTION
- COMMAND_FUNCTION_CODE

- MORE
- NUMBER
- ROW_COUNT
- TRANSACTION_ACTIVE

The right hand side of a condition information item in a GET DIAGNOSTICS statement must be one of the following Condition Area field names:

- CLASS_ORIGIN
- CONDITION_IDENTIFIER
- CONDITION_NUMBER
- MESSAGE_LENGTH
- MESSAGE_TEXT
- RETURNED_SQLSTATE
- SUBCLASS_ORIGIN

Otherwise, the request aborts during compilation and returns an error to the requestor.

If the Diagnostics Area is empty, as would happen if GET DIAGNOSTICS is the first statement in a client-invoked stored procedure and the statement information is requested, the default values for the requested statement information items are returned.

The declared data type of *value* specified in a statement information item or condition information item must be compatible with the data type of the corresponding statement or condition information item name.

Otherwise, the requests aborts during compilation and returns an error.

The table on the following page describes the compatibility rules applicable to valid data types of a *value* specification. The ÷ symbol in a cell indicates that the combination is compatible, and a blank cell indicates that the combination is *not* compatible.

Data Type	CHAR ACTER	VAR CHAR	INTEGER	BYTEINT	SMALL INT	BIG INT	DECIMAL (n,0)	NUMERIC (n,0)
CHAR ACTER	÷	÷						
VAR CHAR	÷	÷						
INTEGER			÷	÷	÷	÷	÷	÷
BYTEINT			÷	÷	÷	÷	÷	÷
SMALL INT			÷	÷	÷	÷	÷	÷
BIGINT			÷	÷	÷	÷	÷	÷
DECIMAL (n,0)			÷	÷	÷	÷	÷	÷

Data Type	CHAR ACTER	VAR CHAR	INTEGER	BYTEINT	SMALL INT	BIG INT	DECIMAL (n,0)	NUMERIC (n,0)
NUMERIC (n,0)			÷	÷	÷	÷	÷	÷

Example: Using the Statement Information Item in a GET DIAGNOSTICS Statement

The following example illustrates the usage of the statement information item field ROW_COUNT in a GET DIAGNOSTICS statement. During the execution of the procedure, the GET DIAGNOSTICS statement sets the *rowcount* parameter to zero. The CALL statement returns OParam = 0 and rowcount = 0.

```
CREATE PROCEDURE getdiag1 (OUT  OParam  INTEGER,
                          INOUT rowcount INTEGER)
BEGIN
  SELECT c1 INTO OParam FROM tab1; -- Returns warning  NODATA FOUND
  GET DIAGNOSTICS rowcount = ROW_COUNT;
  IF (rowcount = 0) THEN
    SET OParam = 0;
  END IF;
END;
BTEQ> CREATE SET TABLE tab1 (c1 INTEGER);
BTEQ> CALL getdiag1(OParam, NULL);
```

Example: Retrieving Information in the Diagnostics Area Using RETURNED_SQLSTATE

The following example illustrates the retrieval of information related to the completion condition in the Diagnostics Area using the statement information item field RETURNED_SQLSTATE. This also illustrates that the condition number, *pcondno*, is not null in this example.

During the execution of the procedure, the SELECT INTO statement clears the Diagnostics Area before it executes and raises the completion condition SQLSTATE '02000'.

The Statement Area is updated and a Condition Area is added to the Diagnostics Area with the information related to the completion condition.

SQLSTATE '02000' is propagated to the outer compound statement. The CONDITION HANDLER in compound statement *cs1* handles the completion condition with SQLSTATE '02000'.

The GET DIAGNOSTICS statement retrieves Condition Area 1 from the Diagnostics Area and assigns the RETURNED_SQLSTATE value '02000' to *sqlstate1*.

The CALL statement returns OParam1 = 0, pcondno = 1, and sqlstate1 = '02000'.

```
CREATE PROCEDURE getdiag5 (OUT  OParam1  INTEGER,
                          INOUT pcondno  INTEGER,
```

```

                                OUT  sqlstate1 CHARACTER(5))
cs1 :BEGIN
  DECLARE nodata CONDITION FOR SQLSTATE '02000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
    GET DIAGNOSTICS EXCEPTION pcondno
    sqlstate1 = RETURNED_SQLSTATE;
  cs2: BEGIN
    SET OParam1 = 0;
    SELECT c1 INTO OParam1 FROM tab1;
    -- Returns warning NO DATA FOUND
  END cs2;
END cs1;
BTEQ> CREATE SET TABLE  tab1 (c1 INTEGER);
BTEQ> CALL getdiag5(OParam1,1,sqlstate1);

```

Example: Using TRANSACTION_ACTIVE in a GET DIAGNOSTICS Statement in Teradata Session Mode

The following example illustrates the usage of the statement information item field TRANSACTION_ACTIVE in a GET DIAGNOSTICS statement in Teradata session mode.

In this example, the procedure is created in Teradata session mode.

During its execution, a duplicate row exception is raised because of the second INSERT statement, so the system rolls back the transaction.

The CONTINUE HANDLER is invoked and the GET DIAGNOSTICS statement retrieves the TRANSACTION_ACTIVE and COMMAND_FUNCTION statement information item fields from the Statement Area of the Diagnostics Area.

When the procedure finishes executing, *OParam* has the value 0 because there is no transaction active when the GET DIAGNOSTICS statement is submitted and *Stmt* has the string 'INSERT'.

```

CREATE PROCEDURE getdiag3 (OUT OParam INTEGER,
                           OUT Stmt  CHARACTER(40))
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    GET DIAGNOSTICS OParam = TRANSACTION_ACTIVE,
    Stmt = COMMAND_FUNCTION;
  INSERT INTO Tab1 VALUES(100);
  INSERT INTO Tab1 VALUES(100);
END;
BTEQ> CREATE SET TABLE tab1 (c1 INTEGER);
BTEQ> CALL getdiag3(OParam, Stmt);

```

Example: Using TRANSACTION_ACTIVE in a GET DIAGNOSTICS Statement for a Procedure Created in ANSI Session Mode

The following example illustrates the usage of the statement information item field TRANSACTION_ACTIVE in a GET DIAGNOSTICS statement for a procedure created in ANSI session mode.

During the execution of the procedure, the duplicate row exception raised because of the second INSERT statement does not roll back the transaction because duplicate rows are permitted in ANSI session mode.

The CONTINUE HANDLER is invoked and the GET DIAGNOSTICS statement sets *OParam* with the value of the TRANSACTION_ACTIVE field from the Statement Area.

When the procedure finishes executing, *OParam* has the value 1 because the transaction is active when the GET DIAGNOSTICS statement is submitted.

```
CREATE PROCEDURE getdiag4 (OUT OParam INTEGER)
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    GET DIAGNOSTICS OParam = TRANSACTION_ACTIVE;
  INSERT INTO Tab1 VALUES(100);
  INSERT INTO Tab1 VALUES(100);
END;
BTEQ> CALL getdiag4(OParam);
```

Example: Runtime Behavior of a GET DIAGNOSTICS Statement When the Diagnostics Area is Empty

The following example illustrates the runtime behavior of a GET DIAGNOSTICS statement when the Diagnostics Area is empty and a statement information item, in this case ROW_COUNT, is requested.

During the execution of the procedure, the first statement executed is the GET DIAGNOSTICS statement. Because the Diagnostics Area is empty at the beginning of the client-invoked stored procedure execution, the *rowcount* parameter is set to the default value of ROW_COUNT, which is zero.

```
CREATE PROCEDURE getdiag3 (OUT rowcount INTEGER)
BEGIN
  GET DIAGNOSTICS rowcount = ROW_COUNT;
  ...
END;
BTEQ> CALL getdiag3(rowcount);
```

Related Information

- Valid condition information item names, see the *condition_name* Syntax Element in the Syntax table for [SIGNAL](#).

- GET DIAGNOSTICS statement, see [Example: Retrieving Information in the Diagnostics Area Using RETURNED_SQLSTATE](#).
- The Diagnostics Area, see [Example: Runtime Behavior of a GET DIAGNOSTICS Statement When the Diagnostics Area is Empty](#).

Host Variables and Multistatement Requests

This section describes special topics and SQL statements whose use is restricted to embedded SQL applications.

Host Structures

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: Producing a Report on the Employee Ethnicity Demographics

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 Not Supported In ANSI Session Mode

ANSI session mode does not support arrays; therefore, it does not support host structures nor does it support qualified host variables.

Host Structures Supported In Teradata Session Mode

Teradata session mode supports IBM-style host structures to a maximum of two levels.

Teradata session 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

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 Preprocessor2 and indirectly defined from within SQL.

The colon-prefixed variables in the USING request modifier and the variables in stored procedure local variables and parameters perform the same function as embedded SQL host variables.

Purpose of Host Variables

Host variables provide input values to SQL statements or receive output values from SQL requests. 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 database.
indicator variable	that indicates any of the following:

A host ...	Is a host variable ...
	<ul style="list-style-type: none"> • 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, Preprocessor2 extracts values from the specified host input variables and sends them to the 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 the database returns the results of an SQL data returning statement to the client application program, Preprocessor2 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.

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.

Some statement-independent rules are noted below:

- All host variables must be preceded by a COLON character.

Specify the COLON character to distinguish a variable from a table column reference. Example 1:

```
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.

Example 2:

```
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.

In Teradata mode, the COLON character preceding the host variables (*intofieldn*, for example) is optional, but its use is very strongly recommended.

- The COLON character is mandatory before an indicator host variable (*indvar1* and *indvar2*) in the following example.

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'
```

:indvarn indicates whether the associated *:intofieldn* is null, for character and byte string data, whether any truncation occurred in returning the database data to the field.

:intofieldn contains the value of *column_n* when the statement is executed. If *column_n* is null, then the value of *:intofieldn* is indeterminate.

- Pad characters before or after the COLON character are optional. For more information on COLON character usage with host variables with the individual statements, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- 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 are allowed in a WHERE clause.

- 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 are not allowed in the SELECT list.
- You can use host and indicator variables 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.

COLON Character Usage With Host Variables

The ANSI/ISO 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.

The best practice is to precede *all* host variables with a COLON character, even when your session is running in Teradata 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.
- 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.
- 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 question mark (?) 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 executed as a dynamic SQL statement using parameter markers instead of host variables.

```
INSERT INTO parts
VALUES (:part_no, :part_desc)
```

```
INSERT INTO parts
VALUES (?, ?)
```

Related Information

- “USING request modifier,” [Local Variables](#) and [Parameters](#), see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- Host variables providing input values to SQL statements, see [Input Host Variables](#) or for receive output values from SQL requests, see [Output Host Variables](#).
- Host main variable, see [Indicator Variables](#) for more information on host indicator variables.
- Handling nulls in an application or indicator variables, see [Indicator Variables](#) Teradata SQL and COLON character, see [Mandatory COLON Character Usage in Teradata Mode](#) and [Optional COLON Character Usage in Teradata Mode](#).
- Using placeholders in dynamic SQL statements, see [PREPARE \(Dynamic\)](#).

Input Host Variables

When an SQL statement with host variable inputs is sent to the database client for processing, Preprocessor2 extracts the input values from the corresponding host input variables and then sends them to the database 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.

Rules

- 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) is specified and the indicator variable shows NULL.

- If an indicator variable 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 care in using CHARACTER host variables as input to VARCHAR fields.

The system 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 CHARACTER(3) with a value of A ('A ') loaded into a 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 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 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 CHARACTER(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 question mark (?) token, referred to as a parameter marker.

When the SQL statement is executed, Preprocessor2 substitutes the appropriate host variable for the question mark (?) 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 question mark (?), 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.

Preprocessor2 extracts the value from the host variable when the statement is executed and passes it to the database in place of the question mark (?) parameter marker token.

Related Information

- INOUT, see [Parameters](#) and [Rules for IN, OUT, and INOUT Parameters](#)
- Associated indicator host variable or indicator variables, see [Indicator Variables](#)

Output Host Variables

When the results of a data returning SQL statement are received from the database, Preprocessor2 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.

Database Column Data Type	Host Variable Data Type
BYTE(m) VARBYTE(m)	BYTE(n) VARBYTE(n) CHARACTER(n) VARCHAR(n) BYTEINT SMALLINT INTEGER BIGINT DECIMAL NUMERIC FLOAT REAL DOUBLE PRECISION CHARACTER(n) CHARACTER SET GRAPHIC VARCHAR(n) CHARACTER SET GRAPHIC
CHARACTER(m) VARCHAR(m)	CHARACTER(n) VARCHAR(n)
DATE (Teradata)	CHARACTER(n) VARCHAR(n) INTEGER BIGINT DECIMAL NUMERIC FLOAT REAL DOUBLE PRECISION
DATE (ANSI) TIME TIMESTAMP INTERVAL	CHARACTER(n)
BYTEINT SMALLINT INTEGER BIGINT DECIMAL NUMERIC	BYTEINT SMALLINT INTEGER BIGINT DECIMAL NUMERIC

Database Column Data Type	Host Variable Data Type
FLOAT REAL DOUBLE PRECISION	FLOAT REAL DOUBLE PRECISION
GRAPHIC(m) VARGRAPHIC(m)	CHARACTER(n) CHARACTER SET GRAPHIC VARCHAR(n) CHARACTER SET GRAPHIC
UDT	Not currently supported. The fromsql transform target data type defined by a CREATE TRANSFORM statement for the UDT.

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 16 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.
VARBYTE	$m \leq n$	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 16 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.

Data Type	Condition	Description
	$m > n$	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	$m \leq n$	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.
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 SMALLINT INTEGER BIGINT DECIMAL NUMERIC FLOAT REAL 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.

Related Information

For more information about the UDT data type, see the information about CREATE TRANSFORM in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

SQL Character Strings as Host Variables

Preprocessor2 treats SQL character strings as a third kind of host variable that is neither input nor output.

An *SQL string* is a series of characters used to complete an embedded SQL statement. 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 sections of *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446.

Character strings can require a leading COLON character when referenced in an embedded SQL statement. For details, see the individual statement syntax documentation in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 and in this section.

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

You can, if you wish, associate an indicator host variable with any main host variable. The value of indicator variables is set by the sending agent (client application-to-Vantage or Vantage-to-client application) in a client-server data exchange to inform the receiving agent if the host variable is null.

Syntax

```
[:] host_variable_name [ INDICATOR ] :indicator_variable_name
```

Syntax Elements

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

- 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 the database if the host variable is null.

For an output host variable, the database 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 ...
<i>-n</i> where <i>n</i> is typically 1	null.
0	non-null and successfully returned to the output host variable.
<i>+n</i>	truncated. The truncation occurred when returning a character or byte string to the main variable associated with the indicator variable. The value <i>n</i> reports the original length of the string before truncation.

Processing of Indicator Variables

The database processes indicator variables 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, the database 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, the database implicitly casts USING values that are CharFix and have the right length for the target DateTime or Interval type.

See the information about the USING request modifier in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Example: The Indicator Variable

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.

Example: Defining the Department Number As Null

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. EXEC SQL
  INSERT INTO EMPLOYEE
  VALUES (:EMPNO INDICATOR :EMPNO-INDIC, :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.

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.

WHEN ...	THEN ...
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.

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

```

Related Information

- Embedded SQL-related topics, see the following sections: [SQL Cursors](#), [Result Code Variables](#), [Client-Server Connectivity Statements](#), or [Multisession Programming With Embedded SQL](#).
- PREPARE and DESCRIBE, see [COMMENT \(Returning Form\)](#) and [PREPARE \(Dynamic\)](#).
- Using cursors with multistatement requests, see [DECLARE CURSOR](#).

SQL Control Statements

This section describes the SQL stored procedure control flow statements that enable SQL to be a computationally complete language.

These control statements are only valid inside a stored procedure. You *cannot* use them interactively or within embedded SQL applications.

BEGIN END

Delimits a compound statement in a stored procedure.

ANSI Compliance

BEGIN END is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Stored procedures only.

Syntax

```
[ label_name : ] BEGIN
  [ local_declaration ] [...]
  [ cursor_declaration ] [...]
  [ condition_handler ] [...]
  [ statement ] [...]
END [ label_name ] ;
```

Syntax Elements

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. However, 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, or a condition declared using the DECLARE CONDITION statement.

In the case of nested compound statements, variables and conditions 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 to avoid 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.

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.

Usage Notes

- 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 ...	Execution ...
LEAVE	terminates the BEGIN END statement with which that label is associated at runtime. Control moves to the next statement following the terminated block.

FOR this statement ...	Execution ...
	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.
ITERATE	returns a syntax error when the stored procedure body is parsed during stored procedure creation.

- Order of declarations

In a BEGIN END compound statement you can specify any number of declarations, and statements to execute 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 and condition declarations.
2. Cursor declarations.
3. Condition handler declarations.
4. One of the following:
 - a single static or dynamic SQL statement or control statement
 - a compound statement enclosing a list of statements.

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.

- 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.
- You can nest BEGIN END compound statements. 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 execute stored procedures from within a BEGIN END statement.
- The scope of the local variables, conditions, parameters, and cursors declared in a compound statement is the entire compound statement, including all nested compound statements.
- If a local variable, condition, parameter or cursor name in an inner compound statement is the same as one in an outer compound statement, the local variable, condition, parameter, or cursor name

in the inner compound statement takes precedence during execution over the name in the outer compound statement.

- Exception, completion, and user-defined 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.

- Exception, completion, and user-defined conditions raised 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. It remains unhandled. The only exception is the RESIGNAL statement, whose condition is propagated outside the compound statement action clause in a handler.

The following table compares unhandled exception, completion and user-defined conditions.

IF the unhandled condition is ...	THEN ...
an exception or user-defined condition	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 handler action continues with the next statement.

Example: A Valid Stored Procedure with Nested Compound Statements

The following example illustrates a valid stored procedure with nested compound statements.

```
CREATE PROCEDURE spAccount(OUT p1 CHARACTER(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: Using An Outer Compound Statement's Variable In the Inner Compound Statement

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 IOParm1 INTEGER,
                           OUT OParam2 INTEGER)
L1: BEGIN
  DECLARE K INTEGER DEFAULT 10;
  L2: BEGIN
    DECLARE K INTEGER DEFAULT 20;
    SET OParam2 = K;
    SET IOParm1 = 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: Creating a Valid Stored Procedure with Local Variable and Condition Handler Declarations

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 handler action clause raises another '42000' exception, which is handled by the CONTINUE handler.

```
CREATE PROCEDURE spSample3(OUT p1 CHARACTER(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: Reusing Local Variables and Condition Handlers for the Same SQLSTATE Code

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;

```

Related Information

- LEAVE statement, see [LEAVE](#)
- ITERATE statement, see [ITERATE](#).
- Local variable and condition declarations, see [DECLARE](#) and [DECLARE CONDITION](#).
- Cursor declarations, see [Cursor Declarations](#).
- Condition handler declarations, see [DECLARE HANDLER \(Basic Syntax\)](#) and the subsequent DECLARE HANDLER sections.
- Static or dynamic SQL statement, control statements, and compound statements enclosing a list of statements, see [DDL Statements in Stored Procedures](#).
- Local variable, condition, parameter or cursor name in an inner compound statement, see [ITERATE](#).

CASE

Provides conditional execution of statements based on the equality of two operands (simple CASE statement) or the value of a conditional expression (searched CASE statement).

The CASE statement is different from the SQL CASE expression, which returns the result of an expression.

ANSI Compliance

CASE is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable

Stored procedures only.

Simple CASE Syntax

```
CASE operand_1
  when_clause [...]
  [ ELSE statement [...] ]
END CASE ;
```

when_clause

```
WHEN operand_2 THEN statement [...]
```

Related Information:

[Simple CASE Statement](#)

Searched CASE Syntax

```
CASE
  when_clause [...]
  [ ELSE statement [...] ]
END CASE ;
```

when_clause

```
WHEN conditional_expression THEN statement [...]
```

statement

```
{ SQL_statement |
  compound_statement |
  assignment_statement |
  condition_statement |
  [ label_name : ] iteration_statement [ label_name ] |
  ITERATE label_name |
  LEAVE label_name
}
```

compound_statement

```
[ label_name : ] BEGIN
  [ local_declaration ] [...]
  [ cursor_declaration ] [...]
  [ condition_handler ] [...]
  [ statement; ] [...]
END [ label_name ] ;
```

assignment_statement

```
SET assignment_target = assignment_source
```

condition_statement

```
{ CASE_statement | IF_statement }
```

iteration_statement

```
{ WHILE conditional_expression
  DO statement; [...] |

  LOOP
    statement; [...]
  END LOOP |

  FOR for_loop_variable AS [ cursor_name CURSOR FOR ] cursor_specification
    DO statement; [...]
  END FOR |

  REPEAT
    statement; [...]
    UNTIL conditional_expression
  END REPEAT
}
```

local_declaration

```
DECLARE {
  variable_name [, ...] data_type [ DEFAULT { literal | NULL } ] |
  condition_name CONDITION [ FOR SQLSTATE [VALUE] sqlstate_cod ]
} ;
```

cursor_declaration

```
DECLARE cursor_name [ [NO] SCROLL ] CURSOR
[ WITHOUT RETURN |
  WITH RETURN [ONLY] [ TO { CALLER | CLIENT } ]
]
[ FOR { cursor_specification [ FOR { READ ONLY | UPDATE } ] |
```

```

        statement_name
    }
] ;

```

IF_statement

See [IF](#).

CASE_statement

Either form of the CASE statement.

condition_handler

```

DECLARE { CONTINUE | EXIT } HANDLER FOR
{
    { SQLSTATE [ VALUE ] sqlstate_code | condition_name } [,...] |

    { SQLEXCEPTION | SQLWARNING | NOT FOUND } [,...]

} handler_action_statement ;

```

cursor_specification

```

SELECT cursor_spec [,...]
FROM {
    table_name [,...] |

    table_name {
        INNER |
        { LEFT | RIGHT | FULL } OUTER
    } JOIN table_name ON condition
}

```

cursor_spec

```

{ column_name [ [AS] alias_name ] |

  expression [AS] alias_name |

```

```
*
}
```

Related Information:

[Searched CASE Statement](#)

CASE Syntax Elements

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.

conditional_expression

A boolean condition used to determine whether a statement or statements in the THEN clause should be executed.

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.

Usage Notes

Simple CASE Statement

In this form of the conditional statement, you can execute 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:

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 executed.
 - 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 executed. 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 executes a list of statements when the conditional expression in the WHEN clause evaluates to true. You can execute 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:

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 executed.
 - 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 executed.

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.

Semantic Differences Between CASE Statement and CASE Expression

The semantics for stored procedure CASE statements and the CASE expression of ordinary interactive SQL are different. For example, expressions have values, but statements do not; expressions cannot be executed, but statements can; expressions can be combined with other expressions, but statements cannot. See the information about CASE expressions in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

CASE Statement or IF-THEN-ELSEIF-ELSE Statement

The alternative to using CASE statements is using an IF-THEN-ELSEIF-ELSE statement.

CASE statements are generally preferred when there are more than two conditions or values to be checked.

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.

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.

Examples

Example: Simple CASE

The following stored procedure includes a simple CASE statement.

```

CREATE PROCEDURE spSample(IN  pANo    INTEGER,
                           IN  pName   CHARACTER(30),
                           OUT pStatus CHARACTER(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 preceding example, the appropriate SET statement of a WHEN clause is executed depending on the value of the local vNoAccts.

IF the value of vNoAccts is ...	THEN it matches ...	AND this statement is executed ...
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: Searched CASE

The following stored procedure includes a searched CASE statement.

```

CREATE PROCEDURE spSample (IN pANo INTEGER,
                           IN pName CHARACTER(30), OUT pStatus CHARACTER(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 preceding example, the appropriate SET statement of a WHEN clause is executed 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 executed ...
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: Using FOR Loop Aliases in 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;

```

Related Information

- Simple CASE statement, see [ITERATE](#).
- Searched CASE statement, see [ITERATE](#).
- IF-THEN-ELSEIF-ELSE statement, see [IF](#).
- Examples and rules governing exception conditions, see [DDL Statements in Stored Procedures](#).

DECLARE

Declares one or more local variables.

ANSI Compliance

DECLARE is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Nonexecutable control declaration.

Stored procedures only.

Syntax

```

DECLARE variable_name [,...]
{ predefined_data_type | UDT_name }
[ attribute ]
[ DEFAULT

```

```

{ literal |
  NEW [SYSUDTLIB.] constructor_name |
  NULL
}
] ;

```

Syntax Elements

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 runtime, 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`

predefined_data_type

The data type of the declared local variable.

This can be either a predefined data type or a UDT, except for the VARIANT_TYPE UDT data type.

UDT_name

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 defaults to the character set of the user creating/compiling the stored procedure.

You can also specify CASESPECIFIC or NOT CASESPECIFIC with *data_type*.

DEFAULT *literal*

The default value for the local variables.

If specified, this must be a literal and compatible with the specified data type. However, Vantage performs an implicit conversion if a default DateTime value differs from the specified DateTime data type. For details, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

If a default value is specified for a local variable declaration, that default is assigned to all variables in the list.

A DEFAULT clause cannot contain an expression.

The variable is initialized to NULL if you do not specify a default for it.

Usage Notes

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 and condition 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. However, Vantage performs an implicit conversion if a default DateTime value differs from the specified DateTime data type. For details, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Example: Specifying Declaration

The declaration is completely specified:

```
DECLARE hErrorMsg CHARACTER(30) DEFAULT 'NO ERROR';
```

Example: Specifying Multiple Local Variables of the Same Data Type

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 CHARACTER(30).

```
DECLARE hFirstName, hLastName CHARACTER(30);
```

Example: Assigning a Default Value For Each Local Variable

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 CHARACTER(30) DEFAULT 'NO ERROR';
```

Example: Declaring the Variable MyCircle

The following DECLARE statement declares the variable *MyCircle*, which has the structured UDT type *CircleUdt*, to have a default value determined by the constructor external routine named *circle* with input parameters of 1, 1, and 9:

```
DECLARE MyCircle CircleUdt DEFAULT NEW circle(1,1,9);
```

Example: Declaring hBirthdate as DATE Data Type

The following statement declares *hBirthdate* to be of DATE data type with a default value of '1998-01-06'.

```
DECLARE hBirthdate DATE DEFAULT '1998-01-06';
```

FOR

Executes a statement for each row fetched from a table.

ANSI Compliance

FOR is ANSI/ISO SQL:2011-compliant.

Required Privileges

- SELECT privilege on the database object referenced in the specified *cursor_specification*.
- UPDATE or DELETE privilege if the cursor is updatable.

Invocation

Executable.

Stored procedures only.

FOR Syntax

```
[ label_name : ] FOR for_loop_variable
  AS [ cursor_name CURSOR FOR ] cursor_specification
```

```
DO statement [...]
END FOR [ label_name ] ;
```

cursor_specification

```
SELECT cursor_spec [,...]
FROM {
    table_name [,...] |

    table_name {
        INNER |
        { LEFT | RIGHT | FULL } OUTER
    } JOIN table_name ON condition
}
```

statement

```
{ SQL_statement |
  compound_statement |
  assignment_statement |
  condition_statement |
  [ label_name : ] iteration_statement [ label_name ] |
  ITERATE label_name |
  LEAVE label_name
}
```

compound_statement

```
[ label_name : ] BEGIN
  [ local_declaration ] [...]
  [ cursor_declaration ] [...]
  [ condition_handler ] [...]
  [ statement; ] [...]
END [ label_name ] ;
```

assignment_statement

```
SET assignment_target = assignment_source
```

condition_statement

```
{ CASE_statement | IF_statement }
```

iteration_statement

```
{ WHILE conditional_expression
  DO statement; [...] |

  LOOP
    statement; [...]
  END LOOP |

  FOR for_loop_variable AS [ cursor_name CURSOR FOR ] cursor_specification
    DO statement; [...]
  END FOR |

  REPEAT
    statement; [...]
  UNTIL conditional_expression
  END REPEAT
}
```

cursor_spec

```
{ column_name [ [AS] alias_name ] |

  expression [AS] alias_name |

  *
}
```

local_declaration

```
DECLARE {
  variable_name [,...] data_type [ DEFAULT { literal | NULL } ] |
  condition_name CONDITION [ FOR SQLSTATE [VALUE] sqlstate_cod ]
} ;
```

cursor_declaration

```

DECLARE cursor_name [ [NO] SCROLL ] CURSOR
  [ WITHOUT RETURN |
    WITH RETURN [ONLY] [ TO { CALLER | CLIENT } ]
  ]
  [ FOR { cursor_specification [ FOR { READ ONLY | UPDATE } ] |
    statement_name
  }
] ;

```

condition_handler

```

DECLARE { CONTINUE | EXIT } HANDLER FOR
{
  { SQLSTATE [ VALUE ] sqlstate_code | condition_name } [,...] |

  { SQLEXCEPTION | SQLWARNING | NOT FOUND } [,...]

} handler_action_statement ;

```

FOR Syntax Elements

IF_statement

See [IF](#).

CASE_statement

See [CASE](#).

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.

Usage Notes

DECLARE CURSOR and FOR Statements

FOR statements contain DECLARE CURSOR statements.

LEAVE and ITERATE

You can execute LEAVE and ITERATE statements within a FOR block.

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.

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. <ul style="list-style-type: none"> A positioned UPDATE can execute 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 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.

Rules for FOR Cursors

- ABORT, COMMIT, and ROLLBACK statements are not permitted in an updatable cursor.
An attempt to execute 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

- 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

- 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

- 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).

Examples

Example: FOR-Loop Insert

```
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: FOR-Loop Delete

```
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: FOR-Loop Update

```
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 = 50000;
  END IF;
  UPDATE customer
    SET amount = hAmount WHERE CURRENT OF c_customer;
  INSERT INTO Cust_temp VALUES (hCustNbr,
                                hCustName);
END FOR;
```

Related Information

- Differences between DECLARE CURSOR and FOR statements, see [DECLARE CURSOR Statement and FOR Statement Cursors](#).
- LEAVE and ITERATE statements, see [ITERATE](#) and [LEAVE](#).

IF

Provides conditional execution based on the truth value of a condition.

ANSI Compliance

IF is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable

Stored procedures only.

IF Syntax

```
IF conditional_expression THEN statement [...]
  [ ELSEIF conditional_expression THEN statement [...] ]
  [ ELSE statement [...] ]
END IF ;
```

IF Syntax Elements

conditional_expression

A boolean condition used to evaluate whether a statement or statements embedded within the IF block should be executed.

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.

For details, see *statement* in [FOR](#).

Usage Notes

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.

Valid Forms of the IF Statement

- IF-THEN-END IF
- IF-THEN-ELSE-END
- IF-THEN-ELSEIF-END
- IF-THEN-ELSEIF-THEN-ELSE-END

IF-THEN-END IF

This form of IF executes 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 executes the statements within the IF and ELSE bounds when *conditional_expression* evaluates to TRUE. Otherwise, the statements within the ELSE and END IF bounds execute.

In the following example, only one of the specified INSERT statements executes, 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 Behavior

1. The statements between the IF and ELSEIF boundaries execute 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 execute. Subsequent ELSEIF clauses do not execute.
4. When no statement in the IF/END IF block evaluates to TRUE, then none of the statements can execute.

In the following example, either one and only one of the ELSEIF clauses executes 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;
```

In the following example, one and only one of the ELSEIF clauses executes 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 Behavior

1. The statements between the IF and ELSEIF boundaries execute 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 execute. Subsequent ELSEIF clauses do not execute 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 executes.

In the following example, one and only one of the DML statements associated with an ELSEIF or ELSE clause executes, 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;

```

In the following example, one and only one of the DML statements associated with an ELSEIF or ELSE clause executes, 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, 'Nonvalid customer');
ELSE
  INSERT INTO temp_table VALUES (hNoAccts, 'More than one customer');
END IF;

```

ITERATE

Terminates the execution of an iterated SQL statement and begins the next iteration of the iterative statement in the loop.

ITERATE executes the following actions depending on the referenced iteration statement.

IF ITERATE specifies the label of ...	THEN ...
FOR statement	<p>IF a next row in the cursor, exists, execution continues with the next statement in the FOR loop.</p> <p>IF a next row in the cursor does not exist, the cursor is closed and execution continues with the next statement outside the corresponding END FOR terminator for the FOR loop.</p>
LOOP statement	the first statement inside the LOOP block executes unconditionally.
REPEAT statement	execution continues with the first statement inside the REPEAT loop without any evaluation of the UNTIL clause.
WHILE statement	IF the conditional expression for the WHILE statement evaluates to TRUE, execution continues with the first statement inside the WHILE loop.

IF ITERATE specifies the label of ...	THEN ...
	If the condition expression for the WHILE statement does not evaluate to TRUE, execution continues with the next statement outside the corresponding END WHILE terminator for the loop.

ANSI Compliance

ITERATE is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable

Stored procedures only.

Syntax

```
ITERATE label_name ;
```

Syntax Elements

label_name

The name of the label on which iteration is to be executed.

Usage Notes

- 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.

Example: Using ITERATE to Iterate a WHILE Statement

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: Using ITERATE to Iterate An Outer Loop

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, hAccountNum);
END WHILE LOOP1;
UPDATE transaction
    SET account_num = hAccountNum + 10;

```

Example: Using ITERATE to Iterate Outside a FOR Loop

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

Breaks execution of a labeled iteration or compound statement and continues execution outside an iteration statement.

- 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 only if the label is associated with the outermost or the only compound statement in 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 executes the following actions depending on the referenced iteration statement:

IF LEAVE specifies the label of ...	THEN ...
Any statement	LEAVE terminates the execution 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 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.
```

ANSI Compliance

LEAVE is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Stored procedures only.

Syntax

```
LEAVE label_name ;
```

Syntax Elements

label_name

The name of the label for the iteration statement or BEGIN END block to be terminated by the LEAVE.

Usage Notes

- 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 or the BEGIN END compound statement in which you embed the LEAVE statement.

Example: Terminating the Execution of a Stored Procedure

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: Using LEAVE with an Iteration Statement

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

Repeats the execution of one or more statements embedded within the defined iteration statement.

ANSI Compliance

LOOP is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Stored procedures only.

Syntax

```
[ label_name : ] LOOP
  statement [ ... ]
END LOOP [ label_name ] ;
```

Syntax Elements

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.

For details, see *statement* in [FOR](#).

Usage Notes

- 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.

- Causes of 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 LOOP Statement

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

Repeats the execution of one or more statements until the specified condition evaluates to true.

ANSI Compliance

REPEAT is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Stored procedures only.

Syntax

```
[ label_name : ] REPEAT statement [...]
  UNTIL conditional_expression
  END REPEAT [ label_name ] ;
```

Syntax Elements

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 executed.

The list can contain any of the following:

- DML, DDL or DCL statements, including dynamic SQL.
- Control statements, including BEGIN END.

For details, see *statement* in [FOR](#).

conditional_expression

A boolean condition used to evaluate whether a statement or statements embedded within the REPEAT loop should be executed.

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*.

Usage Notes

- 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
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.
- 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 unconditional. REPEAT always executes the sequence of statements at least once.	makes the first iteration and subsequent iterations only if a specified condition is true.
executes statements until a specified condition is met.	executes statements as long as a specified condition is met.

Example: Using a REPEAT Statement

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;
```

Related Information

For more information about rules governing exceptions, see [LEAVE](#).

SET

Assigns a value to a local variable or parameter in a stored procedure.

ANSI Compliance

SET is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Stored procedures only.

Syntax

```
SET assignment_target = assignment_source ;
```

Syntax Elements

assignment_target

The name of the variable or parameter to be assigned a value.

assignment_source

The value to be assigned to *assignment_source*.

Usage Notes

- 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 left hand side (LHS) of the SET expression.
- Assignment source is always on the right hand side (RHS) of the SET expression.
- The data type of the assignment source must be compatible with the data type specified for the assignment target. Vantage performs implicit conversions for DateTime data types when the source data type differs from the target data type. For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

FOR this
component of
a SET
assignment ...

Valid Constructs Are...

Invalid Constructs Are...

assignment target

- Local variable name

- QUERY_BAND variable name

FOR this component of a SET assignment ...	Valid Constructs Are...	Invalid Constructs Are...
	<ul style="list-style-type: none"> OUT or INOUT parameter name 	<ul style="list-style-type: none"> Status variable FOR loop column and correlation name IN parameter
assignment source	a literal or an expression containing one of the following: <ul style="list-style-type: none"> Local variable IN or INOUT parameter FOR loop column and correlation names when the SET statement is within the scope of the FOR statement Constant expression Status variable An expression that evaluates to a UDT 	<ul style="list-style-type: none"> OUT parameter FOR loop column and correlation names when the SET statement is not within the scope of the FOR statement

Example: Using the SET Statement to Assign Values

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

Repeats the execution of a statement or statement list while a specified condition evaluates to true.

ANSI Compliance

WHILE is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Stored procedures only.

Syntax

```
[ label_name : ] WHILE conditional_expression DO
  statement [...]
END WHILE [ label_name ] ;
```

Syntax Elements

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 executed.

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 executed.

The list can contain any of the following:

- DML, DDL or DCL statements, including dynamic SQL.
- Control statements, including BEGIN END.

For details, see *statement* in [FOR](#).

Usage Notes

- You can qualify WHILE with a label.
- You can specify a LEAVE or ITERATE statement within a WHILE statement.

Example: Using WHILE

```
WHILE hCounter > 0
DO
  INSERT INTO transaction (trans_num, account_num)
```

```
VALUES (hCounter, hAccountNum);
SET hCounter = hCounter - 1;
END WHILE;
```

Example: Using WHILE to Set the High Number

```
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;
```

Related Information

For more information about the LEAVE or ITERATE statement, see [ITERATE](#) and [LEAVE](#).

Static Embedded SQL Statements

This section describes declarative and other miscellaneous static embedded SQL-only statements.

Statements for Positioned Cursors

The following statements, employed both with embedded SQL and stored procedures, are used with positioned cursors.

Related Information

- [DELETE \(Positioned Form\)](#)
- [UPDATE \(Positioned Form\)](#)

BEGIN DECLARE SECTION

Identifies the start of an embedded SQL declare section for an application written in C.

ANSI Compliance

BEGIN DECLARE SECTION is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Nonexecutable preprocessor declaration.

Embedded SQL only.

Syntax

```
BEGIN DECLARE SECTION
```

Usage Notes

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements are mandatory for applications written in C.

Preprocessor2 issues a warning if it finds either statement in a COBOL or PL/I application.

All host variables must be defined within the declare section.

You must specify the complete BEGIN DECLARE SECTION statement, including the SQL prefix and terminator, on a single line. Only a pad character can separate the words of the statement.

Related Information

[END DECLARE SECTION.](#)

COMMENT (Returning Form)

Returns the comment (if any) that belongs to an object.

The returning form of COMMENT returns data.

ANSI Compliance

COMMENT is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
COMMENT [ON] object_kind object_reference
      INTO [:] host_variable_name
      [ [ INDICATOR ] :host_indicator_name ]
```

Syntax Elements***object_kind***

One of these objects:

- COLUMN
- DATABASE
- FUNCTION
- 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
- user-defined function name
- view name

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.

Usage Notes

- 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 execute 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

Specifies a default database.

ANSI Compliance

DATABASE is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Invocation

Executable.

Embedded SQL only.

Syntax

```
DATABASE { database_name | database_name_variable }
```

Syntax Elements

database_name

An SQL identifier.

database_name_variable

A host variable.

The colon is mandatory.

Usage Notes

- 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 Preprocessor2 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 Preprocessor2.

DECLARE STATEMENT

Declares the names used to identify prepared dynamic SQL statements.

ANSI Compliance

DECLARE STATEMENT is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Nonexecutable preprocessor declaration.

Embedded SQL only.

Syntax

```
DECLARE statement_name [,...] STATEMENT
```

Syntax Elements

statement_name

The name associated with a previously prepared statement.

If you declare multiple statement names, you must separate each name with a COMMA character.

Usage Notes

DECLARE STATEMENT is used for program documentation only.

DECLARE TABLE

Declares tables used by the embedded SQL statements within an application.

ANSI Compliance

DECLARE TABLE is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Nonexecutable preprocessor declaration.

Embedded SQL only.

Syntax

```
DECLARE { table_name | view_name } TABLE ( column_spec [,...] )
```

Syntax Elements

column_spec

```
column_name data_type [ null_attribute ]
```

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 the *null_attribute* is NOT NULL, nulls are not permitted and there is no default value specified.

If the *null_attribute* is NOT NULL WITH DEFAULT, nulls are not permitted and a default values is specified.

If the *null_attribute* is not specified, null are permitted.

Usage Notes

DECLARE TABLE is useful for program documentation, but Preprocessor2 treats it only as a comment.

Preprocessor2 does not verify 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

Identifies the end of an embedded SQL declare section for an application written in C.

ANSI Compliance

END DECLARE SECTION is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Nonexecutable preprocessor declaration.

Embedded SQL only.

Syntax

```
END DECLARE SECTION
```

Usage Notes

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements are mandatory for applications written in C.

Preprocessor2 issues a warning if it finds either statement in a COBOL or PL/I application.

The complete END DECLARE SECTION statement (including the SQL prefix and terminator) must be specified on a single line. Only a pad character can separate the words of the statement.

Related Information

[BEGIN DECLARE SECTION.](#)

END-EXEC Statement Terminator

Terminates an SQL statement in an embedded SQL client COBOL application program.

ANSI Compliance

END-EXEC is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Nonexecutable preprocessor declaration.

Embedded SQL only.

Syntax

```
END-EXEC [ . ]
```

Usage Notes

- END-EXEC is mandatory for all SQL statements embedded in a client COBOL application program.
The statement terminator for SQL embedded within C and PL/I applications is the SEMICOLON character.
- You cannot use END-EXEC with interactive SQL statements.

Related Information

- [EXEC SQL Statement Prefix](#)

EXEC

Executes an SQL macro.

ANSI Compliance

EXEC is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
EXEC macro_name [ ( parameter_list ) ]
```

Syntax Elements

macro_name

The name of the macro to be executed.

parameter_list

The SQL macro parameters.

Usage Notes

The statement *must* be spelled EXEC, *not* EXECUTE, to distinguish it from the dynamic SQL statement EXECUTE.

Any macro specified by *macro_name* can contain no more than one Teradata SQL statement.

Any macro specified by *macro_name* cannot return data.

You must use a macro cursor to execute the following types of macros:

- Multistatement
- Data returning

EXEC SQL Statement Prefix

Denotes the beginning of an SQL statement in an embedded SQL application.

ANSI Compliance

EXEC SQL is ANSI/ISO SQL:2011-compliant with extensions.

Required Privileges

None.

Invocation

Nonexecutable preprocessor declaration.

Embedded SQL only.

Syntax

```
EXEC SQL [ FOR [ : ] count_value ]
        embedded_sql_statement sql_statement_terminator
```

Syntax Elements

FOR

Specifies that parameter arrays are supported for the SQL statement that follows. Note that iterated support is provided only for multiple INSERTs.

count_value

A user-defined INTEGER host variable or a literal INTEGER constant that specifies the number of iterations to be executed.

embedded_sql_statement

The embedded SQL statement to be executed by the client application program.

sql_statement_terminator

The SQL statement terminator for the client language.

FOR COBOL, the SQL statement terminator is END-EXEC.

FOR C and PL/I, the SQL statement terminator is semi-colon (;)

Usage Notes

- General Rules for Using EXEC SQL

The following rules apply to EXEC SQL:

- EXEC SQL must precede each SQL statement embedded in any client application program, regardless of the language used to write the application code.
- The phrase EXEC SQL must be coded together on a single line.
- The SQL statement that follows an EXEC SQL phrase can begin immediately following the phrase or it can begin on a new line.
- You cannot use EXEC SQL with interactive SQL statements.

- Rules for Using EXEC SQL With DML Arrays

The following rules apply to using EXEC SQL with arrays using a FOR clause:

Iteration support is provided for simple single INSERT statements only.

Other DML statements, such as those in the following list, are not supported:

- DELETE
- INSERT ... SELECT
- SELECT
- UPDATE

You can specify a *count_value* using either a host variable or a literal INTEGER value.

All host variable parameter arrays must be single-dimensioned.

Arrays of arrays are not supported except in C, and then the system supports only arrays of character strings.

You are responsible for setting all host variables before they are used, including *count_value*.

Literal constants embedded in the SQL string are not iterated.

Instead, they are propagated in every inserted row.

The collection of host variables used in the iterated statement is treated as independent parallel arrays.

Preprocessor2 checks the value of FOR *count_value* to determine if it is less than or equal to any of the array sizes.

If the count value exceeds the array size for any column, Preprocessor2 aborts the request and returns an error message.

The same host variable can be specified for multiple fields.

Any given iteration will use the same value because there is one index for all of the arrays.

References to arrays of host program *structs* are not supported, only references to arrays of variables.

Example: Simple Array Example

This example provides a simple example of SQL DML array processing. Note that *count_value* is supplied as an INTEGER literal value of 19.

```
EXEC SQL FOR 19
  INSERT INTO table1
  VALUES (:var1, :var2, :var3);
```

Example: Array Example For Dynamic SQL

This example demonstrates the use of SQL DML array processing using dynamic SQL within a program written in C. Note that *count_value* is supplied using a host variable named *cNewEmployees*:

```
char empname[50][20];
integer empnum[50];
float empsal[50];
intc NewEmployees = 50;
VARCHAR stmtstr[100];
char      *ins001=
  "INSERT INTO EMPLOYEE (EMPLOYEE_NUMBER, LAST_NAME, SALARY_AMOUNT)"
  "VALUES (?, ?, ?);";
      strcpy(stmtstr.arr,ins001);
      stmtstr.len = strlen(ins001);
EXEC SQL
  PREPARE insStmt FROM :stmtstr;
```

```
EXEC SQL FOR :cNewEmployees
EXECUTE insStmt USING :empnum, :empname, :empsal;
```

Related Information

- [END-EXEC Statement Terminator](#)

WHENEVER

Specifies the action to be taken when an exception condition occurs.

ANSI Compliance

WHENEVER is ANSI/ISO SQL:2011-compliant with extensions.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
WHENEVER condition action
```

Syntax Elements

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 is the definition of the values for the SQLCODE and SQLSTATE variables when the condition occurs.

When SQLERROR is a condition in which an SQL error occurs:

- SQLCODE has the following value: <0
- When NOT FOUND is a condition in which no data is found:
 - SQLCODE has the following value: +100
 - SQLSTATE has the following value: 02xxx

When SQLWARNING is a condition in which an SQL warning occurs (SQLWARNING is a non-ANSI Teradata extension):

- SQLCODE has the following value: a positive number other than +100
- SQLSTATE is not defined.

action

The action to be executed when condition occurs.

The valid actions are:

- CONTINUE
- GO TO :*host_label*
- GOTO :*host_label*
- PERFORM *code*
- CALL *function_call*

where:

- :*host_label* specifies a valid target of a client language GO TO statement. Use of a preceding colon is strongly recommended.
- *code* specifies the name of a section or paragraph in the application to be executed when the exception condition occurs. The PERFORM action is valid only for COBOL.
- *function_call* specifies the function to be called when the exception condition occurs.

Usage Notes

- 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.
- 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. <i>host_label</i> 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 executed (called) and control has been returned to the calling program.

IF an exception condition applies and the action is ...	THEN the application program continues execution at the ...
	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 executed. 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 execution of an SQL statement ...	THEN the following exception condition applies ...
any negative number	SQLERROR
a positive number other than +100	SQLWARNING
+100	NOT FOUND

Related Information

- SQLSTATE, see [SQLSTATE](#).
- Rules for the object of a GO TO, see *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446.

INCLUDE

Incorporates an external source file into the application program.

ANSI Compliance

INCLUDE is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Nonexecutable preprocessor declaration.

Embedded SQL only.

Syntax

```
INCLUDE include_file_name
```

Syntax Elements

include_file_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.

Usage Notes

- Preprocessor2 searches the directory path for a file having the specified name and a language specific file extension:

For this language ...	Preprocessor2 searches for this INCLUDE file type ...
C	pc
COBOL	pb
PL/I	pi

- The length of the INCLUDE filename can be up to 30 bytes.
- The INCLUDE statement is effectively replaced by the included text in the application program input to Preprocessor2.
- INCLUDE statements cannot be nested.
Included text can contain any embedded SQL statements except another INCLUDE.
- You can specify INCLUDE SQLCA and INCLUDE SQLDA statements in the included text.

Related Information

[INCLUDE SQLCA](#) and [INCLUDE SQLDA](#).

INCLUDE SQLCA

Defines the SQL Communications Area (SQLCA) in a client embedded SQL application program.

ANSI Compliance

INCLUDE SQLCA is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Nonexecutable preprocessor declaration.

Embedded SQL only.

Syntax

```
INCLUDE SQLCA
```

Usage Notes

- When operating in Teradata session mode, you must declare exactly one SQL Communications Area in your embedded SQL application program.

You can use either an INCLUDE SQLCA statement or an equivalent user-supplied declaration.

- When operating in ANSI session mode, Preprocessor2 flags INCLUDE SQLCA statements as an error.

ANSI/ISO SQL requires you to explicitly define a result code variable named SQLSTATE.

If you are operating in ANSI mode, you can also define an SQLCODE result code variable to receive error codes.

ANSI/ISO SQL no longer supports SQLCODE.

- The complete INCLUDE SQLCA statement, including the EXEC SQL prefix and the appropriate terminator, must be coded on one line.

Only a pad character can separate the words of the statement.

No other statements can appear on the line.

- Preprocessor2 replaces the INCLUDE SQLCA statement with the appropriate language definition of the SQL Communications Area.
- For applications written in COBOL, the SQL Communications Area declaration must appear in the WORKING STORAGE SECTION.

Related Information

- SQL result codes, see [Result Code Variables](#), [SQL Communications Area \(SQLCA\)](#) and [SQLSTATE Mappings](#).
- SQLSTATE, see [SQLSTATE](#).
- SQLCODE, see [SQLCODE](#).

INCLUDE SQLDA

Defines the SQL Descriptor Area (SQLDA) in a C or PL/I application program.

ANSI Compliance

INCLUDE SQLDA is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Nonexecutable preprocessor declaration.

Embedded SQL only.

Syntax

```
INCLUDE SQLDA
```

Usage Notes

- Either an INCLUDE SQLDA statement or an equivalent user-supplied declaration of the SQL Descriptor Area is required in every application program that uses dynamic SQL.
- The complete SQLDA statement, including the EXEC SQL prefix and the appropriate terminator, must be coded on one line.

Only a pad character can separate the words of the statement.

No other statements can appear on the same line.

- Preprocessor2 replaces the INCLUDE SQLDA statement with the appropriate language definition of the SQL Descriptor Area.
- For PL/I applications, 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.
- For COBOL applications, you cannot use INCLUDE SQLDA statements because COBOL does not support based structures.

As a result, if a COBOL program requires one or more SQL Descriptor Areas, you must code them yourself and insert them into the WORKING STORAGE SECTION of the program.

Related Information

[SQL Descriptor Area \(SQLDA\).](#)

Dynamic Embedded SQL Statements

Dynamic SQL is a facility that permits an interactive user to submit SQL statements dynamically to an application written using embedded SQL or stored procedures.

When contrasted with dynamic SQL, the facilities ordinary SQL provides are sometimes referred to as *static SQL*.

Always use static SQL if possible, because dynamic SQL has a significant processing overhead that often impedes system performance.

Using Dynamic SQL

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.

For example, dynamic SQL would be useful for a spreadsheet-driven application where 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 which SQL statements are required to support the ad hoc formulas the user may type, you should code the application using dynamic SQL features to support the dynamic requirements of the spreadsheet.

Performing SQL Statements Dynamically

You can execute SQL statements dynamically in either prepared or immediate form, as described in the following table.

Note:

Stored procedures only support the immediate form of dynamic SQL.

Dynamic SQL Statement Type	Description
Prepared	Valid statement text is prepared and preserved for the duration of the session and the statement can be executed 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.
Immediate	Valid statement text is executed one time only.

Dynamic SQL Statement Type	Description
	If a statement must be executed more than one time in an application, then you must incur the overhead of preparing and performing it each time.

Related Information

- Stored procedures and dynamic SQL, see [Using Dynamic SQL in Stored Procedures](#).
- Preparing dynamic SQL statements in embedded SQL applications, see [EXECUTE \(Dynamic SQL Form\)](#) and [PREPARE \(Dynamic\)](#).
- Preparing and executing dynamic SQL statements in embedded SQL applications, see [EXECUTE IMMEDIATE](#).

Dynamic SQL Statement Syntax

The following SQL statements, unique to dynamic SQL are described in the sections that follow:

- [DESCRIBE](#)
- [EXECUTE \(Dynamic SQL Form\)](#)
- [EXECUTE IMMEDIATE](#)
- [PREPARE \(Dynamic\)](#)

There is also a form of DECLARE CURSOR that is specific to dynamic SQL (see [DECLARE CURSOR \(Dynamic SQL Form\)](#)).

These statements are used only by embedded SQL. Stored procedures also support dynamic SQL, but in a completely different way than embedded SQL. See [Using Dynamic SQL](#) for further information.

DESCRIBE

Obtains information about the data to be returned when a previously prepared dynamic SQL statement is executed.

ANSI Compliance

DESCRIBE is a Teradata extension to the ANSI/ISO SQL:2011 standard.

DESCRIBE functions like the ANSI/ISO SQL:2011 statements DESCRIBE INPUT and DESCRIBE OUTPUT.

Required Privileges

None.

Invocation

Executable.

Dynamic SQL.

Embedded SQL only.

Syntax

```
DESCRIBE statement_name INTO [:] descriptor_area
  [ USING { NAMES | ANY | BOTH | LABELS } ]
  [ FOR STATEMENT { statement_number | [:] statement_number_variable } ]
```

Syntax Elements

statement_name

The name associated with the previously prepared statement. Must be a valid SQL identifier, not enclosed within apostrophes.

descriptor_area

The area to receive the information about the data which will be returned when the previously prepared statement is executed.

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.

statement_number

The statement number within the request for which the information is required.

Must be a valid integer numeric literal.

statement_number_variable

The statement number within the request for which the information is required.

Must identify a host variable of type INTEGER or SMALLINT.

Usage Notes

- General Rules
 - 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 itself cannot be executed as a dynamic SQL statement.
- USING Clause Rules

IF this form of USING is specified ...	THEN column ...
NAMES	names are placed in the SQLNAME fields of the SQLDA. This also happens if you do not specify a USING clause.
LABELS	titles are placed in the SQLNAME fields of the SQLDA. If you specify a TITLE clause for a column in the select list, then that title is returned. If a column was defined with a TITLE at CREATE TABLE time and no title is specified in the SELECT statement, then the title specified for the CREATE TABLE statement is returned. In the absence of either, the default title, the 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) as follows: The first set of elements contains the names and column information. 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 FOR STATEMENT clause is intended to support dynamic multistatement requests, but can also be used for single statement requests.
 - If you do not specify a FOR STATEMENT clause, the descriptive information is returned for the first or only SQL statement of the prepared dynamic SQL request.
 - If you specify a FOR STATEMENT clause, the descriptive information is returned for that SQL statement of the prepared dynamic SQL request that 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

- *descriptor_area*, see *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446.
- [EXECUTE \(Dynamic SQL Form\)](#)
- [EXECUTE IMMEDIATE](#)
- [PREPARE \(Dynamic\)](#)

EXECUTE (Dynamic SQL Form)

Executes a prepared dynamic SQL statement.

ANSI Compliance

The dynamic SQL form of EXECUTE is ANSI/ISO SQL:2011-compliant.

Required Privileges

The privileges required depend on the SQL statement and tables accessed.

Invocation

Executable.

Dynamic SQL statement.

Embedded SQL only.

Syntax

```
EXECUTE statement_name
  [ USING using_spec [, ...] |
    USING DESCRIPTOR [:] descriptor_area
  ]
```

Syntax Elements

using_spec

```
[:] host_variable_name [ [INDICATOR] :host_indicator_name ]
```

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 code *descriptor_area* as a name or as a pointer reference (*sqldaname) in C programs when the SQLDA structure is declared as a pointer.

Usage Notes

- General Rules

An SQLDA should be defined for the application.

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
- Macro
- Multistatement request

For these cases, a dynamic cursor must be declared and the application program should access the results using an appropriate FETCH statement.

EXECUTE itself cannot be executed as a dynamic SQL statement.

- USING Clause Rules

- The USING clause identifies variables used as input to the SQL statement specified by *statement_name*.
- The specified 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 n^{th} variable must correspond to the n^{th} parameter marker.

- The descriptor name identifies an input SQLDA structure previously defined by the application. This SQLDA contains all necessary information about the input variable set.

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 n^{th} variable described by the SQLDA must correspond to the n^{th} parameter marker.

Related Information

- [DESCRIBE](#)
- [EXECUTE IMMEDIATE](#)
- [PREPARE \(Dynamic\)](#)

- *descriptor_area*, see *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446.
- FETCH statement, see [FETCH \(Embedded SQL Form\)](#)

EXECUTE IMMEDIATE

Prepares and executes a dynamic SQL statement.

ANSI Compliance

EXECUTE IMMEDIATE is ANSI/ISO SQL:2011-compliant.

Required Privileges

The privileges required depend on the SQL statement and tables accessed.

Invocation

Executable.

Dynamic SQL statement.

Embedded SQL only.

Syntax

```
EXECUTE IMMEDIATE statement_string
```

Syntax Elements

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.

Usage Notes

Whether specified as a string expression or as a host variable, the dynamic SQL statement can be no longer than 32000 characters.

If specified as a host variable, the *statement_string_variable* must follow the rules for SQL strings for the client programming language, as given in the following table:

IN this language ...	<i>statement_string</i> is a ...
COBOL	non-numeric literal.

IN this language ...	<i>statement_string</i> is a ...
C	
PL/I	character string expression.

If the statement string variable is VARCHAR, the statement text cannot be longer than 64K.

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 is designed to provide for this special case.

The dynamic SQL statement:

- Cannot be any of the following specific SQL statements.
- Cannot be a data returning statement.

ABORT	EXECUTE IMMEDIATE
BEGIN TRANSACTION	FETCH
CHECKPOINT	LOGOFF
CLOSE	LOGON
COMMIT	OPEN
CONNECT	POSITION
DESCRIBE	PREPARE
ECHO	REWIND
END TRANSACTION	ROLLBACK
EXECUTE	

- Cannot be a Preprocessor2 declarative.
- Cannot include host variable references.
- Requires a dynamic cursor when executed dynamically.

Related Information

- [DESCRIBE](#)
- [EXECUTE \(Dynamic SQL Form\)](#)
- [PREPARE \(Dynamic\)](#)

PREPARE (Dynamic)

Prepares a dynamic SQL statement for execution and assigns a name to it.

ANSI Compliance

PREPARE is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Dynamic SQL.

Embedded SQL only.

PREPARE Syntax

```
PREPARE statement_name
  [ INTO [:] descriptor_area
  [ USING { NAMES | ANY | BOTH | LABELS } ] ]
  [ FOR STATEMENT { statement_number | [:] numeric_variable } ]
]
FROM { statement_string | [:] statement_string_variable }
```

PREPARE Syntax Elements

statement_name

The name to be associated with the prepared statement.

statement_name must be a valid SQL identifier and must not be enclosed within apostrophes.

descriptor_area

Specifies the SQLDA to receive the descriptive information about the data returned when the prepared statement is executed.

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.

statement_number

A valid integer literal that identifies the statement number within the request for which the descriptive information is requested

numeric_variable

A host variable of type INTEGER or SMALLINT that represents the statement number in the request for which the descriptive information is requested.

The preceding colon is optional.

Usage Notes

Preparing an SQL Statement

Using the syntax elements defined here, the process is as follows:

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 executed, still in the client application language.
3. Execute 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. Execute the EXECUTE or EXECUTE IMMEDIATE statement for *statement_name*.
5. The database software posts return codes to SQLCODE and SQLSTATE.

Rules

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 text, USING data, and parcel overhead).

If specified as a host variable, the statement string must follow the rules for SQL strings for the client programming language.

IN this language ...	<i>statement_string</i> 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

If the statement string variable is varchar, the statement text cannot be longer than 64K.

The dynamic SQL statement cannot be:

- Any of the following specific SQL statements:

ABORT	EXECUTE IMMEDIATE
BEGIN TRANSACTION	FETCH
CHECKPOINT	LOGOFF
CLOSE	LOGON
COMMIT	OPEN
CONNECT	POSITION
DESCRIBE	PREPARE
ECHO	REWIND
END TRANSACTION	ROLLBACK
EXECUTE	

- A Preprocessor2 declarative.

The dynamic SQL statement can include parameter markers, or placeholder tokens (the question mark), 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 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 = ?;
```

Using 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 not a placeholder or the NULL keyword.

The following uses of untyped placeholders are 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 ? = ?
```

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;
```

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.
- PREPARE cannot be executed as a dynamic SQL statement.

Related Information

- [DESCRIBE](#)
- [EXECUTE \(Dynamic SQL Form\)](#)
- [EXECUTE IMMEDIATE](#)
- SQL string rules, see *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446 for details.
- INTO, USING and FOR STATEMENT clauses of the PREPARE statement, see [DESCRIBE](#).

Client-Server Connectivity Statements

This section documents the statements used to execute and maintain the connectivity between a client application program performing embedded SQL statements and the database, known also as the *server*.

All SQL statements described in this section are for use in embedded SQL applications only. They cannot be used interactively.

Connecting a Client Application to Vantage

The Teradata-embedded SQL preprocessor, which runs on a client system, needs to make a connection to Vantage. Connections are required both during precompilation and at runtime.

There is no relationship between the preprocessor connection to Vantage 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 Vantage:

IF you specify the SQLCHECK or -sc option as...	THEN the preprocessor...
NOSYNTAX	does not require connection.
<ul style="list-style-type: none"> FULL (or use FULL as the default) SQLFLAGGER(ENTRY) 	requires connection.

You can establish a preprocessor connection 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.

Note:

For the mainframe version only, Preprocessor2 will not operate without at least one started TDP on the mainframe, even if no data access is required (that is, NOSYNTAX). If a precompile step is done on the mainframe with no started TDP, Preprocessor2 will output the message:

```
SPP9980 Fatal Error:
      Unexpected CLI return 280 on DBCHINI call
```

Runtime Execution Connection

Runtime connection to Vantage 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.

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 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	the previous connection is dropped before the new connection is attempted.
a transaction is active	the connection request is rejected with an 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: <ul style="list-style-type: none"> • COMMIT • ROLLBACK (or ABORT) • LOGOFF

Any explicit connection to Vantage requires the following:

- TDP ID
- User ID
- Password

TDP ID and user ID preprocessor options do not affect the application logon at execution time.

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
workstation-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 an SQL request without specifying an explicit connection, 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.

Related Information

- Preprocessor invocation options, see *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446.
- User ID security, TDP IDs, and user IDs, see *Teradata® Director Program Reference*, B035-2416.
- Default TDP ID obtained from the HSHSPB data area module, see *Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems*, B035-2417.
- Implicit connection mechanism, see *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418.

CONNECT

Explicitly connects a client application program to Vantage.

ANSI Compliance

CONNECT is a Teradata extension to the ANSI/ISO SQL:2011 standard.

A CONNECT statement is defined in the ANSI/ISO SQL:2011 standard, but the ANSI form has slightly different syntax.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

CONNECT Syntax

```
CONNECT [ : ] user_id_variable
      IDENTIFIED BY [ : ] password_variable
      [ AS { connection_name | :connection_name_variable } ]
```

CONNECT Syntax Elements

user_id_variable

The host variable that contains the Vantage user ID to be used for connection.

The user ID is restricted to eight characters.

password_variable

The host variable that contains the password for the specified user ID.

The password is restricted to eight characters.

Use of a preceding colon character 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

Name of the connection.

:connection_name_variable

The host variable that contains the connection name.

The preceding colon character is mandatory.

Usage Notes

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 Vantage

CONNECT statements have no effect on preprocessor connections to Vantage.

Difference Between Implicit and 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.

General Rules

- Use of the CONNECT statement is optional. If the application program executes any SQL statement which requires access to Vantage and the program is not currently connected to Vantage, an implicit connection is attempted.
- If the application program executes a CONNECT statement while already connected to Vantage, the previous connection is dropped.
- Both *user_id-variable* and *password_variable* 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 executed as a dynamic statement.
- The application program is connected to Vantage using the specified user ID and password.

Rules for AS (*connection_name* | :*connection_name_variable*) 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.

- *:connection_name_variable* must be a fixed or a varying length character variable no longer than 30 bytes.

Related Information

- CONNECT statements, see *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446 for further information.

- An alternative way to connect to Vantage from a client application program, see [LOGON](#).
- How the preprocessor can connect to Vantage without using CONNECT or LOGON statements, see *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446.
- Examples you can easily adapt to the CONNECT statement, see [Example 1: Logging On Using a Host Variable](#), [Example 2](#) and [Example 3](#).

GET CRASH

Displays the crash maintenance options established by the SET CRASH statement.

ANSI Compliance

GET CRASH is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
GET CRASH WAIT, TELL INTO [:] wait_variable, [:] tell_variable
```

Syntax Elements

wait_variable

wait_across_crash (WAC) option setting value.

tell_variable

tell_about_crash (TAC) option setting value.

Usage

GET CRASH is valid only for workstation platforms. Mainframe precompilers generate an error if this statement is submitted to the preprocessor.

LOGOFF

Disconnects an embedded SQL application program from Vantage.

ANSI Compliance

LOGOFF is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
LOGOFF [
  CURRENT |
  ALL |
  connection_name |
  :connection_name_variable
]
```

Syntax Elements

CURRENT

Logs off the current session only.

ALL

Logs off all sessions connected with the current user.

connection_name

The name of the connection.

:connection_name_variable

The host variable that contains the connection name.

The preceding colon character is mandatory.

Usage Notes

- The LOGOFF statement is optional. If omitted, a disconnect from Vantage is implicitly executed when the application program terminates.
- LOGOFF can be used without regard to whether the connection to Vantage 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.

If an application is running in any of the other transaction modes, LOGOFF does not cause outstanding transactions to be committed.

- LOGOFF cannot be executed 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*. *:namevar* must be a fixed or varying length character variable no longer than 30 bytes.

Examples 1 - 4

The following RDTIN fields are important for the following LOGOFF statements that specify a disconnect object:

This RDTIN field ...	Must ...
RdtVersn	be set to 10.
RdtAux1	be set to one of the following values: <ul style="list-style-type: none"> • 0, which means to disconnect the current connection • 1, which means to disconnect all connections • 2, which means to disconnect named connection
RdtExt	be set to 'Y', indicating the existence of an extension area, only if a named connection is specified.
RdtXTotL	include the size of the RDTXCONM extension area, only if a named connection is specified.

Additionally, the RdtX007 (RDTXCONM) structure must be included as one of the extension areas because it communicates the connection name if you only specify a named connection.

Example 1

The following example disconnects using an explicit connection name:

```
EXEC SQL LOGOFF SESSION1;
```

Lines Generated by C Preprocessor2 for Example 1

```
{
static struct {
```

```

SQLInt32 RdtCType;
SQLInt16 RdtVersn;
SQLInt16 RdtDec;
char      RdtUserid[8];
SQLInt32 RdtEntty;
char      *RdtCA;
char      *RdtDAIn;
char      *RdtDAOOut;
char      *RdtSql;
char      *RdtRtCon;
SQLInt32 RdtAux1;
SQLInt32 RdtAux2;
char      RdtLCS;
char      RdtComit;
char      RdtRelse;
char      RdtExt;
char      RdtSepBT;
char      RdtUCStm;
char      RdtCmpat;
char      RdtComp;
SQLInt16 RdtXTotL;
char      RdtXFill[2];
struct {
    SQLInt16 RdtXLen;
    SQLInt16 RdtXType;
    SQLInt32 RdtXCode;
} RdtX005;
struct {
    SQLInt16 RdtXLen;
    SQLInt16 RdtXType;
    SQLInt16 RdtXConL;
    char      RdtXConT[30];
} RdtX007;
} RDTIN006 =
{200,10,0,{ ' ' },0,0,0,0,0,0,2,0,'N','B','N','Y','N','N',' ' ,'C',
48,{ ' ' },{8,5,255},{36,7,8,'S','E','S','S','I','O','N','1'}};
RDTIN006.RdtCA = (char *)&sqlca;
RDTIN006.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN006);
SQL_RDTRTCON = RDTIN006.RdtRtCon;
}

```

Example 2

The following example disconnects using a connection name supplied by means of the VARCHAR host variable connamev:

```
EXEC SQL LOGOFF :CONNAMEV;
```

Lines Generated by C Preprocessor2 for Example 2

```
{
static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;
    SQLInt16 RdtDec;
    char      RdtUserid[8];
    SQLInt32 RdtEntty;
    char      *RdtCA;
    char      *RdtDAIn;
    char      *RdtDAOOut;
    char      *RdtSql;
    char      *RdtRtCon;
    SQLInt32 RdtAux1;
    SQLInt32 RdtAux2;
    char      RdtLCS;
    char      RdtComit;
    char      RdtRelse;
    char      RdtExt;
    char      RdtSepBT;
    char      RdtUCStm;
    char      RdtCmpat;
    char      RdtComp;
    SQLInt16 RdtXTotL;
    char      RdtXFill[2];
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        SQLInt32 RdtXCode;
    } RdtX005;
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        SQLInt16 RdtXConL;
        char      RdtXConT[30];
    } RdtX007;
```

```

    } RDTIN007 =
    {200,10,0,{ ' ' },0,0,0,0,0,0,2,0,'N','B','N','Y','N','N',' ','C',
    48,{ ' ' },{8,5,255},{36,7,}}};
RDTIN007.RdtX007.RdtXConL = CONNAMEV.len;
memcpy(RDTIN007.RdtX007.RdtXConT, CONNAMEV.arr, CONNAMEV.len);
RDTIN007.RdtCA = (char *)&sqlca;
RDTIN007.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN007);
SQL_RDTRTCON = RDTIN007.RdtRtCon;
}

```

Example 3

The following example disconnects using a connection name supplied by means of the fixed length host variable connamef:

```
EXEC SQL LOGOFF :CONNAMEF;
```

Lines Generated by C Preprocessor2 for Example 3

```

{
static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;
    SQLInt16 RdtDec;
    char      RdtUserid[8];
    SQLInt32 RdtEntty;
    char      *RdtCA;
    char      *RdtDAIn;
    char      *RdtDAOut;
    char      *RdtSql;
    char      *RdtRtCon;
    SQLInt32 RdtAux1;
    SQLInt32 RdtAux2;
    char      RdtLCS;
    char      RdtComit;
    char      RdtRelse;
    char      RdtExt;
    char      RdtSepBT;
    char      RdtUCStm;
    char      RdtCmpat;
    char      RdtComp;
    SQLInt16 RdtXTotL;
    char      RdtXFill[2];

```

```

    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        SQLInt32 RdtXCode;
    } RdtX005;
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        SQLInt16 RdtXConL;
        char      RdtXConT[30];
    } RdtX007;
    } RDTIN008 =
    {200,10,0,{ ' ' },0,0,0,0,0,0,2,0,'N','B','N','Y','N','N',' ','C',
    48,{ ' ' },{8,5,255},{36,7,}}};
    RDTIN008.RdtX007.RdtXConL = strlen(CONNAMEF);
    memcpy(RDTIN008.RdtX007.RdtXConT,CONNAMEF,strlen(CONNAMEF));
    RDTIN008.RdtCA = (char *)&sqlca;
    RDTIN008.RdtRtCon = SQL_RDTRTCON;
    TDARDI(&RDTIN008);
    SQL_RDTRTCON = RDTIN008.RdtRtCon;
}

```

Example 4

The following example disconnects all connections:

```
EXEC SQL LOGOFF ALL;
```

Lines Generated by C Preprocessor2 for Example 4

```

{
static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;
    SQLInt16 RdtDec;
    char      RdtUserid[8];
    SQLInt32 RdtEntty;
    char      *RdtCA;
    char      *RdtDAIn;
    char      *RdtDAOOut;
    char      *RdtSql;
    char      *RdtRtCon;
    SQLInt32 RdtAux1;
    SQLInt32 RdtAux2;
}

```

```

char      RdtLCS;
char      RdtComit;
char      RdtRelse;
char      RdtExt;
char      RdtSepBT;
char      RdtUCStm;
char      RdtCmpat;
char      RdtComp;
SQLInt16  RdtXTotL;
char      RdtXFill[2];
struct {
    SQLInt16  RdtXLen;
    SQLInt16  RdtXType;
    SQLInt32  RdtXCode;
} RdtX005;
} RDTIN009 =
{200,10,0,{ ' ' },0,0,0,0,0,0,1,0,'N','B','N','Y','N','N',' ','C',
12,{ ' ' },{8,5,255}};
RDTIN009.RdtCA = (char *)&sqlca);
RDTIN009.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN009);
SQL_RDTRTCON = RDTIN009.RdtRtCon;
}

```

Example: LOGOFF CURRENT

The code generated for a LOGOFF CURRENT statement is the same as LOGOFF without specifying a disconnect object.

The RDTIN RdtAux1 field must be set to 0.

LOGON

Explicitly connects an embedded SQL application program to Vantage.

ANSI Compliance

LOGON is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
LOGON [ : ] logon_string
      [ AS { connection_name | :connection_name_variable } ]
```

Syntax Elements

logon_string

Variable containing the logon string to be used.

connection_name

Specified name of the connection.

connection_name_variable

The host variable that contains the connection name.

The preceding colon character is mandatory.

Usage Notes

- 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.

- General Rules

- LOGON is optional.

If it is used, LOGON must be the first SQL statement executed by the application.

If omitted, connection to Vantage is made implicitly.

- *logon_string* 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 character before the host variable is optional.
- The application program is connected to Vantage 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 executed as a dynamic statement.

- Rules for AS (*connection_name* | :*connection_name_variable*) 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 `:connection_name_variable` must be a fixed or varying length character variable no longer than 30 bytes.

Examples 1-3

For all of following examples, the referenced host variables are defined as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
char logstr[103];
VARCHAR CONNAMEV[30];
char CONNAMEF[31];
char STMTNAMF[31];
VARCHAR STMTNAMV[30];
EXEC SQL END DECLARE SECTION;
```

The following RDTIN fields are important for these examples:

This RDTIN field ...	Must ...
RdtVersn	be set to 10.
RdtExt	be set to 'Y' to indicate the existence of an extension area.
RdtXTotL	include the size of the RDTXCONM extension area.

Additionally, the RdtX007 (RDTXCONM) structure must be included as one of the extension areas because it communicates the connection name.

Example 1: Logging On Using a Host Variable

This example logs on using a host variable to communicate the connection name.

```
EXEC SQL LOGON :logstr AS SESSION1;
```

Lines Generated by C Preprocessor2 for Example 1

```
{
static struct {
    SQLInt16 LogonStrLen;
    char      LogonStr[102];
} Sql Stmt010;
static struct {
    SQLInt32 RdtCType;
```

```

SQLInt16 RdtVersn;
SQLInt16 RdtDec;
char      RdtUserid[8];
SQLInt32 RdtEntty;
char      *RdtCA;
char      *RdtDAIn;
char      *RdtDAOOut;
char      *RdtSql;
char      *RdtRtCon;
SQLInt32 RdtAux1;
SQLInt32 RdtAux2;
char      RdtLCS;
char      RdtComit;
char      RdtRelse;
char      RdtExt;
char      RdtSepBT;
char      RdtUCStm;
char      RdtCmpat;
char      RdtComp;
SQLInt16 RdtXTotL;
char      RdtXFill[2];
struct {
    SQLInt16 RdtXLen;
    SQLInt16 RdtXType;
    SQLInt32 RdtXCode;
} RdtX005;
struct {
    SQLInt16 RdtXLen;
    SQLInt16 RdtXType;
    SQLInt16 RdtXConL;
    char      RdtXConT[30];
} RdtX007;
struct {
    SQLInt16 RdtXLen;
    SQLInt16 RdtXType;
    char      RdtLogMech[8];
    char      *RdtLogData;
} RdtX010;
} RDTIN010 =
{110,10,0,{ ' ' },0,0,0,0,0,0,0,0,0,'N','B','N','Y','N','N',' ','C',
64,{ ' ' },{8,5,255},{36,7,8,'S','E','S','S','I','O','N','1'},{16,
10,{ ' ' },0}};
Sql_Stmt010.LogonStrLen = strlen(logstr);
memcpy(Sql_Stmt010.LogonStr,logstr,strlen(logstr));

```

```

RDTIN010.RdtSql = (char *)&Sql_Smt010);
RDTIN010.RdtCA = (char *)&sqlca);
RDTIN010.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN010);
SQL_RDTRTCON = RDTIN010.RdtRtCon;
}

```

Example 2

Similar to Example 1, this example logs on using a host variable to communicate the connection name.

```
EXEC SQL LOGON :logstr AS :CONNAMEV;
```

Lines Generated by C Preprocessor2 for Example 2

```

{
static struct {
    SQLInt16 LogonStrLen;
    char      LogonStr[102];
} Sql_Smt011;
static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;
    SQLInt16 RdtDec;
    char      RdtUserid[8];
    SQLInt32 RdtEntty;
    char      *RdtCA;
    char      *RdtDAIn;
    char      *RdtDAOOut;
    char      *RdtSql;
    char      *RdtRtCon;
    SQLInt32 RdtAux1;
    SQLInt32 RdtAux2;
    char      RdtLCS;
    char      RdtComit;
    char      RdtRelse;
    char      RdtExt;
    char      RdtSepBT;
    char      RdtUCStm;
    char      RdtCmpat;
    char      RdtComp;
    SQLInt16 RdtXTotL;
    char      RdtXFill[2];
    struct {

```

```

        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        SQLInt32 RdtXCode;
    } RdtX005;
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        SQLInt16 RdtXConL;
        char      RdtXConT[30];
    } RdtX007;
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        char      RdtLogMech[8];
        char      *RdtLogData;
    } RdtX010;
} RDTIN011 =
{110,10,0,{ ' ' },0,0,0,0,0,0,0,0,0,0,'N','B','N','Y','N','N',' ','C',
64,{ ' ' },{8,5,255},{36,7},{16,10,{ ' ' },0}};
Sql_Stmt011.LogonStrLen = strlen(logstr);
memcpy(Sql_Stmt011.LogonStr,logstr,strlen(logstr));
RDTIN011.RdtSql = (char *)&Sql_Stmt011;
RDTIN011.RdtX007.RdtXConL = CONNAMEV.len;
memcpy(RDTIN011.RdtX007.RdtXConT,CONNAMEV.arr,CONNAMEV.len);
RDTIN011.RdtCA = (char *)&sqlca;
RDTIN011.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN011);
SQL_RDTRTCON = RDTIN011.RdtRtCon;
}

```

Example 3

The following example logs on using a fixed length character connection name passed to using a host variable.

```
EXEC SQL LOGON :logstr AS :CONNAMEF;
```

Lines Generated by C Preprocessor2 for Example 3

```

{
static struct {
    SQLInt16 LogonStrLen;
    char      LogonStr[102];
} Sql_Stmt012;

```

```

static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;
    SQLInt16 RdtDec;
    char      RdtUserid[8];
    SQLInt32 RdtEntty;
    char      *RdtCA;
    char      *RdtDAIn;
    char      *RdtDAOOut;
    char      *RdtSql;
    char      *RdtRtCon;
    SQLInt32 RdtAux1;
    SQLInt32 RdtAux2;
    char      RdtLCS;
    char      RdtComit;
    char      RdtRelse;
    char      RdtExt;
    char      RdtSepBT;
    char      RdtUCStm;
    char      RdtCmpat;
    char      RdtComp;
    SQLInt16 RdtXTotL;
    char      RdtXFill[2];
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        SQLInt32 RdtXCode;
    } RdtX005;
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        SQLInt16 RdtXConL;
        char      RdtXConT[30];
    } RdtX007;
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        char      RdtLogMech[8];
        char      *RdtLogData;
    } RdtX010;
} RDTIN012 =
{110,10,0,{ ' ' },0,0,0,0,0,0,0,0,0,0,'N','B','N','Y','N','N',' ','C',
 64,{ ' ' },{8,5,255},{36,7},{16,10,{ ' ' },0}};
Sql_Stmt012.LogonStrLen = strlen(logstr);

```

```

memcpy(Sql_Stmt012.LogonStr,logstr,strlen(logstr));
RDTIN012.RdtSql = (char *)(&Sql_Stmt012);
RDTIN012.RdtX007.RdtXConL = strlen(CONNAMEF);
memcpy(RDTIN012.RdtX007.RdtXConT,CONNAMEF,strlen(CONNAMEF));
RDTIN012.RdtCA = (char *)(&sqlca);
RDTIN012.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN012);
SQL_RDTRTCON = RDTIN012.RdtRtCon;
}

```

Related Information

- An alternative way to connect to Vantage from a client application program, see [CONNECT](#)
- How Preprocessor2 can connect to Vantage without using CONNECT or LOGON statements, see *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446.

SET BUFFERSIZE

Specifies the response buffer length to be used when processing an SQL request.

ANSI Compliance

SET BUFFERSIZE is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Nonexecutable.

Preprocessor declarative.

Embedded SQL only.

Syntax

```
SET BUFFERSIZE size
```

Syntax Elements

size

An 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 of 256 to 1MB.

Usage Notes

- Explicitly Specified Response Buffer Length

Preprocessor2 requests that follow the SET BUFFERSIZE statement sequentially use the value for buffer size specified by *size*. The request buffer size is not affected by this statement.

- Default Response Buffer Length

If no SET BUFFERSIZE statement is used, the default response buffer length is used for all requests.

- The value of *size* must be either zero or a valid number within the range of 256 to 65535. 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 the database displays preprocessor warning message SPP1500.

SET CHARSET

Specifies a character set to be used for translation of data to and from the database at program execution.

ANSI Compliance

SET CHARSET is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
SET CHARSET { character_set_name | :character_set_name_variable }
```

Syntax Elements

character_set_name

The character set to be used in translating data between the client and the database.

:character_set_name_variable

A host variable that contains the name of the character set to be used in translating data between the client and the database.

Use of the colon character is mandatory.

For information on UTF-8 and UTF-16 character sets and specific UTF-8 and UTF-16 programming considerations or about :character_set_name_variable, see *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446.

Usage Notes

- SET CHARSET overrides the default character set used for communication between the application and the database at runtime.
- Whether specified as *character_set_name* or as :character_set_name_variable, the character set name must be a valid character set identifier.
- If used to identify the character set name rather than the character set code, *character_set_name* must be enclosed in apostrophes, based on the APOST/QUOTE preprocessor option setting.
- If used to identify the character set name, :character_set_name_variable must follow the rules for SQL strings for the client language.
- If used to identify the character set code, :character_set_name_variable 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 the database at execution time.

SET CONNECTION

Changes the existing session connection to a new connection.

ANSI Compliance

SET CONNECTION is ANSI/ISO SQL:2011-compliant.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
SET CONNECTION { connection_name | :connection_name_variable }
```

Syntax Elements

connection_name

Name of the connection variable to which the current connection is being changed.

:connection_name_variable

The host variable that contains the connection name.

The preceding colon character is mandatory.

Usage Notes

- 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 executed to make a background connection the current one.

Each connection name must be unique (up to 30 bytes) and is case-sensitive.

:namevar must be a fixed or varying length character variable no longer than 30 bytes.

- SET CONNECTION and Multisession Programming
SET CONNECTION is designed to be used with multisession programming, permitting applications to switch connections among multiple concurrent sessions.

Examples 1 - 3

The following RDTIN fields are important for these examples:

This RDTIN field ...	Must ...
RdtCType	be set to 150.
RdtVersn	be set to 10.
RdtExt	be set to 'Y' to indicate the existence of an extension area.
RdtXTotL	include the size of the RDTXCONM extension area.

Additionally, the RdtX007 (RDTXCONM) structure must be included as one of the extension areas because it communicates the connection name.

Example: Establishing a Session Connection

The following example establishes a session connection using an explicitly specified connection name:

```
EXEC SQL SET CONNECTION SESSION1;
```

Lines Generated by C Preprocessor2 for Example 1

```
{
static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;
    SQLInt16 RdtDec;
    char      RdtUserid[8];
    SQLInt32 RdtEntty;
    char      *RdtCA;
    char      *RdtDAIn;
    char      *RdtDAOOut;
    char      *RdtSql;
    char      *RdtRtCon;
    SQLInt32 RdtAux1;
    SQLInt32 RdtAux2;
    char      RdtLCS;
    char      RdtComit;
    char      RdtRelse;
    char      RdtExt;
    char      RdtSepBT;
    char      RdtUCStm;
    char      RdtCmpat;
    char      RdtComp;
    SQLInt16 RdtXTotL;
    char      RdtXFill[2];
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        SQLInt32 RdtXCode;
    } RdtX005;
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        SQLInt16 RdtXConL;
        char      RdtXConT[30];
    } RdtX007;
} RDTIN013 =
```

```

        {150,10,0,{ ' ' },0,0,0,0,0,0,0,0,0,0,'N','B','N','Y','N','N',' ','C',
        48,{ ' ' },{8,5,255},{36,7,8,'S','E','S','S','I','O','N','1'}}};
RDTIN013.RdtCA = (char *)(&sqlca);
RDTIN013.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN013);
SQL_RDTRTCON = RDTIN013.RdtRtCon;
}

```

Example 2

The following example establishes a session connection using a VARCHAR connection name passed to SET CONNECTION by means of a host variable named connamev:

```
EXEC SQL SET CONNECTION :CONNAMEV;
```

Lines Generated by C Preprocessor2 for Example 2

```

{
static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;
    SQLInt16 RdtDec;
    char      RdtUserid[8];
    SQLInt32 RdtEntty;
    char      *RdtCA;
    char      *RdtDAIn;
    char      *RdtDAOOut;
    char      *RdtSql;
    char      *RdtRtCon;
    SQLInt32 RdtAux1;
    SQLInt32 RdtAux2;
    char      RdtLCS;
    char      RdtComit;
    char      RdtRelse;
    char      RdtExt;
    char      RdtSepBT;
    char      RdtUCStm;
    char      RdtCmpat;
    char      RdtComp;
    SQLInt16 RdtXTotL;
    char      RdtXFill[2];
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
    }
}

```

```

        SQLInt32 RdtXCode;
        } RdtX005;
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        SQLInt16 RdtXConL;
        char      RdtXConT[30];
        } RdtX007;
    } RDTIN014 =
    {150,10,0,{ ' ',0,0,0,0,0,0,0,0,'N','B','N','Y','N','N',' ','C',
    48,{ ' ',{8,5,255},{36,7,}}};
    RDTIN014.RdtX007.RdtXConL = CONNAMEV.len;
    memcpy(RDTIN014.RdtX007.RdtXConT,CONNAMEV.arr,CONNAMEV.len);
    RDTIN014.RdtCA = (char *)&sqlca;
    RDTIN014.RdtRtCon = SQL_RDTRTCON;
    TDARDI(&RDTIN014);
    SQL_RDTRTCON = RDTIN014.RdtRtCon;
    }

```

Example 3

The following example establishes a session connection using a CHAR connection name passed to SET CONNECTION by means of a host variable named *connamef*:

```
EXEC SQL SET CONNECTION :CONNAMEF;
```

Lines Generated by C Preprocessor2 for Example 3

```

{
static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;
    SQLInt16 RdtDec;
    char      RdtUserid[8];
    SQLInt32 RdtEntty;
    char      *RdtCA;
    char      *RdtDAIn;
    char      *RdtDAOOut;
    char      *RdtSql;
    char      *RdtRtCon;
    SQLInt32 RdtAux1;
    SQLInt32 RdtAux2;
    char      RdtLCS;
    char      RdtComit;

```

```

char      RdtRelse;
char      RdtExt;
char      RdtSepBT;
char      RdtUCStm;
char      RdtCmpat;
char      RdtComp;
SQLInt16  RdtXTotL;
char      RdtXFill[2];
struct {
    SQLInt16  RdtXLen;
    SQLInt16  RdtXType;
    SQLInt32  RdtXCode;
} RdtX005;
struct {
    SQLInt16  RdtXLen;
    SQLInt16  RdtXType;
    SQLInt16  RdtXConL;
    char      RdtXConT[30];
} RdtX007;
} RDTIN015 =
{150,10,0,{ ' ' },0,0,0,0,0,0,0,0,0,0,'N','B','N','Y','N','N',' ','C',
48,{ ' ' },{8,5,255},{36,7,}};
RDTIN015.RdtX007.RdtXConL = strlen(CONNAMEF);
memcpy(RDTIN015.RdtX007.RdtXConT,CONNAMEF,strlen(CONNAMEF));
RDTIN015.RdtCA = (char *)&sqlca;
RDTIN015.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN015);
SQL_RDTRTCON = RDTIN015.RdtRtCon;
}

```

SET CRASH

Sets the wait_across_crash (WAC) and tell_about_crash (TAC) options for handling node crashes.

ANSI Compliance

SET CRASH is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

SET CRASH Syntax

```
SET CRASH { WAIT_NOTELL | NOWAIT_TELL }
```

SET CRASH Syntax Elements

WAIT_

Specifies 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

Specifies that WAC is to be set to N and TAC is to be set to Y at runtime.

Usage Notes

Platform

SET CRASH is enabled only for workstation-attached platforms. Mainframe precompilers generate an error if this statement is precompiled.

When Crash Options Apply

The crash options are in effect for all embedded SQL statements executed after SET CRASH is executed, including LOGON and CONNECT requests, until another SET CRASH is executed.

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 ...	THEN it ...
a resetting node	<p>aborts.</p> <p>Preprocessing must be restarted after the node resets.</p> <p>This is equivalent to the situation where a utility or application is initiated on an external client that fails.</p>

IF the Preprocessor is running on ...	THEN it ...
any of the following: <ul style="list-style-type: none"> • Non-resetting node • LAN-attached client • Workstation-attached client 	reconnects its session (if connected to a database) 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 following the list:

- Non-resetting node
- Workstation-attached client
- Mainframe-attached client

The application reconnects its session and returns one of the following error codes from the database; the embedded SQL application takes action appropriate to the error condition:

Code	Description
Error 2825	No record of the last request found after Teradata Database restart.
Error 2826	Request completed but all output was lost due to Teradata Database restart.
Error 2828	Request was rolled back during Teradata Database recovery.
Error 3120	Request aborted because of a Teradata Database recovery.

Application Behavior When SET CRASH = NOWAIT_TELL

The behavior of embedded SQL applications when a node resets and SET CRASH = NOWAIT_TELL for the following environments:

- Non-resetting node
- Workstation-attached client
- Mainframe-attached client

is explained as follows:

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.

SET ENCRYPTION

Turns encryption on or off for an SQL statement or a block of SQL statements.

SET ENCRYPTION ON enables data encryption for all embedded SQL statements across the network that are executed after it.

Encryption continues until SET ENCRYPTION OFF is executed.

ANSI Compliance

SET ENCRYPTION is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
SET ENCRYPTION { ON | OFF }
```

Usage Notes

- Because data encryption is supported by CLIV2 for workstation platforms, SET ENCRYPTION is supported in C and COBOL preprocessors for workstation platforms.

- SET ENCRYPTION is not supported for all mainframe preprocessors. All mainframe precompilers generate a compilation error message for the SET ENCRYPTION statement.
- When you run your applications with a DBS server that does not support date encryption, pp2 runtime sets the return code to 500 for any embedded SQL statement you have requested to be encrypted. Check SQLCODE and use PPRTEXT to display the error message.

Multisession Asynchronous Programming With Embedded SQL

This section describes the features that support multisession asynchronous programming with embedded SQL.

Multisession Programming With Embedded SQL

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

USE this statement ...	TO ...
CONNECT or LOGON with an AS <i>session_id</i> 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 <i>session_id</i>	disconnect an application from a specific named session.
LOGOFF ALL	disconnect an application from all sessions.

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.

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.

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.

SET CONNECTION Statement

The SET CONNECTION statement permits an application to switch among multiple sessions.

Status Variables and Data Structures for Embedded SQL Applications

Embedded SQL applications use several standardized status variables and data structures to communicate between the application and the database.

The following standard host variables that receive completion and exception status codes are described in [Result Code Variables](#):

- SQLSTATE
- SQLCODE

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.

Related Information

- ASYNC modifier, see [ASYNC Statement Modifier](#).
- WAIT, see [WAIT](#).
- TEST, see [TEST](#).
- SET CONNECTION, see [SET CONNECTION](#).
- Standard host variables that receive completion and exception status codes, see [Result Code Variables](#).
- SQLSTATE or SQLCODE, see [SQLSTATE](#)
- SQLDA, see [SQL Descriptor Area \(SQLDA\)](#).
- The Teradata analog of SQLCODE and SQLSTATE, called the SQL Communications Area (SQLCA), see [SQL Communications Area \(SQLCA\)](#).
- Activity counts, see [ACTIVITY_COUNT](#) and [SQLSTATE Mappings](#).

Multisession Asynchronous Request Programming Support

This section describes the embedded SQL statements that support multisession programming:

- ASYNC
- TEST
- WAIT

The SET CONNECTION statement also supports multisession embedded SQL programming by making it possible to switch among multiple sessions.

ASYNC Statement Modifier

Initiates the asynchronous execution of an SQL statement.

ANSI Compliance

The ASYNC clause is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
ASYNC (
  { async_statement_identifier | :async_statement_identifier_variable_name }
) async_SQL_statement
```

Syntax Elements***async_statement_identifier***

A case-sensitive, application-supplied identifier assigned to the asynchronously executed SQL statement so it can be accessed and its status can be tested and reported by the TEST and WAIT statements.

Each asynchronous statement identifier can be no more than 30 characters in length and must be unique across all active connections.

:async_statement_identifier_variable_name

The name of a host variable that supplies multiple *async_statement_identifier* strings.

Using a host variable permits a single ASYNC statement to support multiple asynchronous sessions simultaneously.

The identifier must be a fixed or varying length character string no more than 30 characters long.

The preceding colon is mandatory.

async_SQL_statement

The executable SQL statement.

async_SQL_statement can be passed to ASYNC indirectly through dynamic SQL using a host variable.

Usage Notes

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 statements.

You *can* use dynamic SQL to pass an asynchronous SQL statement to ASYNC indirectly through a host variable (see [Example: Using dynamic SQL to Pass the Asynchronous SQL Statement](#)).

ASYNC cannot be used with any of the following embedded SQL declarative statements:

- BEGIN DECLARE SECTION
- DECLARE CURSOR
- DECLARE STATEMENT
- DECLARE TABLE
- END DECLARE SECTION
- INCLUDE
- INCLUDE SQLCA
- INCLUDE SQLDA
- SET BUFFERSIZE
- WHENEVER

ASYNC cannot be used with any of the following executable embedded SQL statement:

- ABORT
- BEGIN TRANSACTION
- COMMIT
- CONNECT
- DATABASE
- DESCRIBE
- END TRANSACTION
- FETCH
- GET CRASH
- LOGOFF
- LOGON
- POSITION
- REWIND
- SET BUFFERSIZE
- SET CHARSET
- SET CRASH

Example: Using the ASYNC Statement Modifier

The following example shows how to use the ASYNC statement modifier.

The following RDTIN fields are important for using ASYNC:

This RDTIN field ...	Must ...
RdtVersn	be set to 10.
RdtExt	be set to 'Y' to indicate the existence of an extension area.
RdtXTotL	include the size of the RDTXASYN extension area.

The RdtX008 (RDTXASYN) structure must be included as one of the extension areas because it communicates the connection name.

```
EXEC SQL ASYNC (INSEMP)
      INSERT EMPLOYEE VALUES (2010,1003,2216,8201,'JONES',
      'FREDDY','20/06/14','19/05/25',200000);
```

Lines Generated by C Preprocessor2 for Example 1

```
{
static struct {
    SQLInt32 StrLen;
    char      Str[93];
} Sql_Smt016 =
{93,{' '}};
static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;
    SQLInt16 RdtDec;
    char      RdtUserid[8];
    SQLInt32 RdtEntty;
    char      *RdtCA;
    char      *RdtDAIn;
    char      *RdtDAOOut;
    char      *RdtSql;
    char      *RdtRtCon;
    SQLInt32 RdtAux1;
    SQLInt32 RdtAux2;
    char      RdtLCS;
    char      RdtComit;
    char      RdtRelse;
    char      RdtExt;
    char      RdtSepBT;
    char      RdtUCStm;
    char      RdtCmpat;
    char      RdtComp;
    SQLInt16 RdtXTotL;
```

```

char      RdtXFill[2];
struct {
    SQLInt16 RdtXLen;
    SQLInt16 RdtXType;
    SQLInt32 RdtXCode;
} RdtX005;
struct {
    SQLInt16 RdtXLen;
    SQLInt16 RdtXType;
    SQLInt32 RdtXAsyC;
    struct {
        SQLInt16 RdtXAsyL;
        char      RdtXAsyT[30];
    } RdtXAsyS;
} RdtX008;
} RDTIN016 =
{300,10,0,{ ' ' },0,0,0,0,0,0,0,0,'N','B','N','Y','N','N',' ','C',
52,{ ' ' },{8,5,255},{40,8,1,{6,'I','N','S','E','M','P',' ',' ',
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',
' ',' ',' ',' ',' ',' ',' ',' ',' '}}}};
memcpy(Sql_Stmt016.Str,"INSERT EMPLOYEE VALUES ( 2010,1003,2216,8201,'\
JONES', 'FREDDY', '20/06/14','19/05/25',200000 )",93);
RDTIN016.RdtSql = (char *)(&Sql_Stmt016);
RDTIN016.RdtCA = (char *)(&sqlca);
RDTIN016.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN016);
SQL_RDTRTCON = RDTIN016.RdtRtCon;
}

```

Examples

These examples present ASYNC statement modifier SQL text without any client programming code context.

Example: Asynchronous Requests to Open a Cursor

This example submits an asynchronous request to open a cursor.

ASYNC (request_1) OPEN cursor_1

Example: Asynchronous Requests to Perform a Searched Update

This example submits an asynchronous request to perform a searched update of a table.

```
ASYNC (request_1) UPDATE table_1
SET a = :a
```

Example: Asynchronous Requests to Execute a Macro

This example submits an asynchronous request to execute a macro.

```
ASYNC (request_1) EXEC macro_1
```

Example: Using dynamic SQL to Pass the Asynchronous SQL Statement

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](#) for more information about testing the completion status of an asynchronous request.

See [WAIT](#) for more information about waiting for an asynchronous request to complete.

See [Dynamic SQL Statement Syntax](#) for more information about dynamic SQL, descriptions of [EXECUTE \(Dynamic SQL Form\)](#) and [PREPARE \(Dynamic\)](#) for information about how to prepare and execute an SQL statement dynamically.

TEST

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.

ANSI Compliance

TEST is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
TEST {
  async_statement_identifier |
  :async_statement_identifier_variable_name
} COMPLETION
```

Syntax Elements

async_statement_identifier

A case-sensitive, application-supplied identifier for an asynchronously executed SQL statement assigned by the ASYNC modifier.

Each asynchronous statement identifier can be no more than 30 characters in length and must be unique across all active connections.

:*async_statement_identifier_variable_name*

the name of a host variable that contains an asynchronous statement identifier.

The identifier must be a fixed or varying length character string no more than 30 characters long.

The preceding colon is mandatory.

Usage Notes

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 *:async_statement_identifier_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 statements.

Examples 1 - 3

The following RDTIN fields are important for these examples:

This RDTIN field ...	Must ...
RdtCType	be set to 460.
RdtVersn	be set to 10.
RdtExt	be set to 'Y' to indicate the existence of an extension area.
RdtXTotL	include the size of the RDTXASYN extension area.

Additionally, the RdtX008 (RDTXASYN) structure must be included as one of the extension areas because it communicates the connection name.

Example: Explicitly Specifying an Async Statement Identifier

This example uses an explicitly specified async statement identifier.

```
EXEC SQL TEST ASYNSTMT1 COMPLETION;
```

Lines Generated by C Preprocessor2 for Example 1

```
{
    static struct {
        SQLInt32 RdtCType;
        SQLInt16 RdtVersn;
        SQLInt16 RdtRfu1;
        char      RdtUserid[8];
        SQLInt32 RdtEntty;
        char      *RdtCA;
        char      *RdtDAIn;
        char      *RdtDAOut;
        char      *RdtSql;
        char      *RdtRtCon;
        SQLInt32 RdtAux1;
        SQLInt32 RdtAux2;
```

```

char    RdtLCS;
char    RdtComit;
char    RdtRelse;
char    RdtExt;
char    RdtSepBT;
char    RdtUCStm;
char    RdtCmpat;
char    RdtComp;
SQLInt16 RdtXTotL;
char    RdtXFill[2];
struct {
    SQLInt16 RdtXLen;
    SQLInt16 RdtXType;
    SQLInt32 RdtXCode;
} RdtX005;
struct {
    SQLInt16 RdtXLen;
    SQLInt16 RdtXType;
    SQLInt32 RdtXAsyC;
    struct {
        SQLInt16 RdtXAsyL;
        char    RdtXAsyT[30];
    } RdtXAsyS;
} RdtX008;
} RDTIN011 =
{460,9,0,{ ' ' },0,0,0,0,0,0,0,0,0,0, 'N', 'C', 'N', 'Y', 'N', 'N', ' ', 'C'
,52,{ ' ' },{8,5,255},{40,8,1,{9, 'A', 'S', 'Y', 'N', 'S', 'T', 'M', 'T'
, '1', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '}}};
...
RDTIN011.RdtX008.RdtXAsyL = strlen(STMTNAMF);
memcpy(RDTIN011.RdtX008.RdtXAsyT, STMTNAMF, strlen(STMTNAMF));
...
}

```

Example 2

The following example uses a host variable to obtain the async statement identifier:

```
EXEC SQL TEST :STMTNAMV COMPLETION;
```

Lines Generated by C Preprocessor2 for Example 2

```

{
static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;
    SQLInt16 RdtDec;
    char      RdtUserid[8];
    SQLInt32 RdtEntty;
    char      *RdtCA;
    char      *RdtDAIn;
    char      *RdtDAOOut;
    char      *RdtSql;
    char      *RdtRtCon;
    SQLInt32 RdtAux1;
    SQLInt32 RdtAux2;
    char      RdtLCS;
    char      RdtComit;
    char      RdtRelse;
    char      RdtExt;
    char      RdtSepBT;
    char      RdtUCStm;
    char      RdtCmpat;
    char      RdtComp;
    SQLInt16 RdtXTotL;
    char      RdtXFill[2];
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        SQLInt32 RdtXCode;
    } RdtX005;
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        SQLInt32 RdtXAsyC;
        struct {
            SQLInt16 RdtXAsyL;
            char      RdtXAsyT[30];
        } RdtXAsyS;
    } RdtX008;
} RDTIN017 =
{460,10,0,{ ' ' },0,0,0,0,0,0,0,0,'N','B','N','Y','N','N',' ' ,'C',
52,{ ' ' },{8,5,255},{40,8,1,}};

```

```

RDTIN017.RdtX008.RdtXAsyS.RdtXAsyL = STMTNAMV.len;
memcpy(RDTIN017.RdtX008.RdtXAsyS.RdtXAsyT, STMTNAMV.arr, STMTNAMV.len);
RDTIN017.RdtCA = (char *)(&sqlca);
RDTIN017.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN017);
SQL_RDTRTCON = RDTIN017.RdtRtCon;
}

```

Example 3

The following example uses a host variable to pass the async statement identifier as a fixed length character string

```
EXEC SQL TEST :STMTNAMF COMPLETION;
```

Lines Generated by C Preprocessor2 for Example 3

```

{
static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;
    SQLInt16 RdtDec;
    char      RdtUserid[8];
    SQLInt32 RdtEntty;
    char      *RdtCA;
    char      *RdtDAIn;
    char      *RdtDAOut;
    char      *RdtSql;
    char      *RdtRtCon;
    SQLInt32 RdtAux1;
    SQLInt32 RdtAux2;
    char      RdtLCS;
    char      RdtComit;
    char      RdtRelse;
    char      RdtExt;
    char      RdtSepBT;
    char      RdtUCStm;
    char      RdtCmpat;
    char      RdtComp;
    SQLInt16 RdtXTotL;
    char      RdtXFill[2];
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
    }
}

```

```

        SQLInt32 RdtXCode;
    } RdtX005;
struct {
    SQLInt16 RdtXLen;
    SQLInt16 RdtXType;
    SQLInt32 RdtXAsyC;
    struct {
        SQLInt16 RdtXAsyL;
        char      RdtXAsyT[30];
    } RdtXAsyS;
    } RdtX008;
} RDTIN018 =
{460,10,0,{ ' ' },0,0,0,0,0,0,0,0, 'N', 'B', 'N', 'Y', 'N', 'N', ' ', 'C',
52,{ ' ' },{8,5,255},{40,8,1,}};
RDTIN018.RdtX008.RdtXAsyS.RdtXAsyL = strlen(STMTNAMF);
memcpy(RDTIN018.RdtX008.RdtXAsyS.RdtXAsyT,STMTNAMF,strlen(STMTNAMF));
RDTIN018.RdtCA = (char *)(&sqlca);
RDTIN018.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN018);
SQL_RDTRTCON = RDTIN018.RdtRtCon;
}

```

Examples 1-3

The following examples present TEST statement SQL text without any client programming code context.

Example: Testing the Statement Identified by the Name *req_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: Testing the Statement Identified by the Host Variable *:reqid_var*

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: Using TEST with WAIT

This example uses TEST with WAIT. The program asynchronously spawns two 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](#) for information about how to submit an asynchronous request.
- See [WAIT](#) for information about how to wait on and test the status of an asynchronous request.

WAIT

Pauses execution of the invoking program and waits for the completion of one or more asynchronous SQL statements.

ANSI Compliance

WAIT is a Teradata extension to the ANSI/ISO SQL:2011 standard.

Required Privileges

None.

Invocation

Executable.

Embedded SQL only.

Syntax

```
WAIT {
  { statement_spec [,...] | ALL } COMPLETION |

  ANY COMPLETION INTO [:] statement_variable [:] session_variable
}
```

Syntax Elements

statement_spec

```
{ async_statement_identifier | :async_statement_identifier_variable_name }
```

async_statement_identifier

A case-sensitive, application-supplied identifier for an asynchronously executed SQL statement assigned by the ASYNC modifier.

Each asynchronous statement identifier can be no more than 30 characters in length and must be unique across all active connections.

async_statement_identifier_variable_name

The name of a host variable that contains an asynchronous statement identifier to be passed to the WAIT statement.

The identifier must be a fixed or varying length character string no more than 30 characters long.

This permits an application to supply multiple values of an *async_statement_identifier* to WAIT by means of a host variable.

The preceding colon is not mandatory, but conforms to good programming practices.

ALL

Pauses execution for all current asynchronously executed SQL statements.

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.

Usage Notes

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.

- SQLCODE is set to -650.
- SQLSTATE is set to '04000'.

IF you specify this option ...	THEN the WAIT statement returns when ...
ALL	all asynchronous statements have finished.
ANY COMPLETION INTO	<p>any of the outstanding asynchronous statements finishes.</p> <p>The asynchronous statement identifier is returned to the host variable <i>async_statement_variable</i> in the INTO clause, and the session identifier is returned to the host variable <i>session_variable</i>.</p> <p>The host variables <i>async_statement_variable</i> and <i>session_variable</i> must be defined as a fixed or varying length character variable with a maximum length of 30 bytes.</p> <p>If the asynchronous statement identifier returned is longer than the length defined for the output host variable, then the exception condition “output host variable is too small to hold returned data” is raised.</p> <ul style="list-style-type: none"> • SQLCODE is set to -304. • SQLSTATE is set to '22003'.

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 statements.

Examples 1 - 4

The following RDTIN fields are important for the following WAIT statement examples:

This RDTIN field ...	Must ...
RdtCType	be set to 470.
RdtAux1	be set to one of the following values: <ul style="list-style-type: none"> • Wait for all asynchronous statements to complete. • Wait for any asynchronous statement to complete. • Wait for a list of named asynchronous statements to complete.
RdtVersn	be set to 10.
RdtExt	be set to 'Y', indicating the existence of an extension area, only if an asynchronous statement is specified.
RdtXTotL	include the size of the RDTXASYN extension area, only if an asynchronous statement is specified.

Additionally, the RdtX008 (RDTXASYN) structure must be included as one of the extension areas because it communicates the connection name if an asynchronous statement is specified.

Example: Passing Fixed Length Character Values to the Host Variables

This example passes fixed length character values to the host variables declared for the statement and session variables in an ANY COMPLETION INTO clause:

```
EXEC SQL WAIT ANY COMPLETION INTO :STMTNAMF, :CONNAMEF;
```

Lines Generated by C Preprocessor2 for Example 1

```
{
static struct {
    char      sqldaid[8];
    SQLInt32  sqldabc;
    short     sqln;
    short     sqld;
    struct {
        short  sqltype;
        short  sqllen;
        char   *sqldata;
        char   *sqlind;
        struct {
            short  length;
            char   data[30];
        };
    };
};
```

```

        } sqlname;
    } sqlvar[2];
} Sql_DA019_Struct0 =
    {'S','Q','L','D','A',' ',' ',' ',104,2,2,{{460,31,0,0,{0,{' '}}}},
    {460,31,0,0,{0,{' '}}}}};
Sql_DA019_Struct0.sqlvar[0].sqldata = STMTNAMEF;
Sql_DA019_Struct0.sqlvar[1].sqldata = CONNAMEF;
{
static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;
    SQLInt16 RdtDec;
    char      RdtUserid[8];
    SQLInt32 RdtEntty;
    char      *RdtCA;
    char      *RdtDAIn;
    char      *RdtDAOut;
    char      *RdtSql;
    char      *RdtRtCon;
    SQLInt32 RdtAux1;
    SQLInt32 RdtAux2;
    char      RdtLCS;
    char      RdtComit;
    char      RdtRelse;
    char      RdtExt;
    char      RdtSepBT;
    char      RdtUCStm;
    char      RdtCmpat;
    char      RdtComp;
    SQLInt16 RdtXTotL;
    char      RdtXFill[2];
    struct {
        SQLInt16 RdtXLen;
        SQLInt16 RdtXType;
        SQLInt32 RdtXCode;
    } RdtX005;
} RDTIN019 =
    {470,10,0,{' '},0,0,0,0,0,0,2,0,'N','B','N','Y','N','N',
    ' ','C',12,{' '},{8,5,255}}};
RDTIN019.RdtDAOut = (char *)(&Sql_DA019_Struct0);
RDTIN019.RdtCA = (char *)(&sqlca);
RDTIN019.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN019);
SQL_RDTRTCON = RDTIN019.RdtRtCon;

```

```

    }
}

```

Example 2

This example passes varying length character values to the host variables declared for the statement and session variables in an ANY COMPLETION INTO clause:

```
EXEC SQL WAIT ANY COMPLETION INTO :STMTNAMV, :CONNAMEV;
```

Lines Generated by C Preprocessor2 for Example 2

```

{
static struct {
    char      sqldaid[8];
    SQLInt32  sqldabc;
    short     sqln;
    short     sqld;
    struct {
        short  sqltype;
        short  sqllen;
        char   *sqldata;
        char   *sqlind;
        struct {
            short  length;
            char   data[30];
        } sqlname;
    } sqlvar[2];
    } Sql_DA020_Struct0 =
    {'S','Q','L','D','A',' ',' ',' ',104,2,2,{{448,30,0,0,{0,{' '}}}},
    ,{448,30,0,0,{0,{' '}}}}};
    Sql_DA020_Struct0.sqlvar[0].sqldata = (char *)&STMTNAMV;
    Sql_DA020_Struct0.sqlvar[1].sqldata = (char *)&CONNAMEV;
    {
static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;
    SQLInt16 RdtDec;
    char     RdtUserid[8];
    SQLInt32 RdtEntty;
    char     *RdtCA;
    char     *RdtDAIn;
    char     *RdtDAOOut;
    char     *RdtSql;

```

```

char      *RdtRtCon;
SQLInt32  RdtAux1;
SQLInt32  RdtAux2;
char      RdtLCS;
char      RdtComit;
char      RdtRelse;
char      RdtExt;
char      RdtSepBT;
char      RdtUCStm;
char      RdtCmpat;
char      RdtComp;
SQLInt16  RdtXTotL;
char      RdtXFill[2];
struct {
    SQLInt16 RdtXLen;
    SQLInt16 RdtXType;
    SQLInt32 RdtXCode;
} RdtX005;
} RDTIN020 =
{470,10,0,{' '},0,0,0,0,0,0,2,0,'N','B','N','Y','N','N',
 ' ','C',12,{' '},{8,5,255}};
RDTIN020.RdtDAOut = (char *)(&Sql_DA020_Struct0);
RDTIN020.RdtCA = (char *)(&sqlca);
RDTIN020.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN020);
SQL_RDTRTCON = RDTIN020.RdtRtCon;
}
}

```

Example 3

This example uses the ALL COMPLETION option to wait until all active asynchronous statements have completed:

```
EXEC SQL WAIT ALL COMPLETION;
```

Lines Generated by C Preprocessor2 for Example 3

```

{
static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;
    SQLInt16 RdtDec;
    char      RdtUserid[8];

```

```

SQLInt32 RdtEntty;
char      *RdtCA;
char      *RdtDAIn;
char      *RdtDAOOut;
char      *RdtSql;
char      *RdtRtCon;
SQLInt32 RdtAux1;
SQLInt32 RdtAux2;
char      RdtLCS;
char      RdtComit;
char      RdtRelse;
char      RdtExt;
char      RdtSepBT;
char      RdtUCStm;
char      RdtCmpat;
char      RdtComp;
SQLInt16 RdtXTotL;
char      RdtXFill[2];
struct {
    SQLInt16 RdtXLen;
    SQLInt16 RdtXType;
    SQLInt32 RdtXCode;
} RdtX005;
} RDTIN021 =
{470,10,0,{ ' ' },0,0,0,0,0,0,1,0,'N','B','N','Y','N','N',' ','C',
12,{ ' ' },{8,5,255}};
RDTIN021.RdtCA = (char *)(&sqlca);
RDTIN021.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN021);
SQL_RDTRTCON = RDTIN021.RdtRtCon;
}

```

Example 4

This example uses multiple explicit asynchronous statements.

```
EXEC SQL WAIT ASYNSTMT1, ASYNSTMT2 COMPLETION;
```

Lines Generated by C Preprocessor2 for Example 4

```

{
static struct {
    SQLInt32 RdtCType;
    SQLInt16 RdtVersn;

```

```

SQLInt16 RdtDec;
char      RdtUserid[8];
SQLInt32 RdtEntty;
char      *RdtCA;
char      *RdtDAIn;
char      *RdtDAOut;
char      *RdtSql;
char      *RdtRtCon;
SQLInt32 RdtAux1;
SQLInt32 RdtAux2;
char      RdtLCS;
char      RdtComit;
char      RdtRelse;
char      RdtExt;
char      RdtSepBT;
char      RdtUCStm;
char      RdtCmpat;
char      RdtComp;
SQLInt16 RdtXTotL;
char      RdtXFill[2];
struct {
    SQLInt16 RdtXLen;
    SQLInt16 RdtXType;
    SQLInt32 RdtXCode;
} RdtX005;
struct {
    SQLInt16 RdtXLen;
    SQLInt16 RdtXType;
    SQLInt32 RdtXAsyC;
    struct {
        SQLInt16 RdtXAsyL;
        char      RdtXAsyT[30];
    } RdtXAsyS[2];
} RdtX008;
} RDTIN022 =
{470,10,0,{ ' ' },0,0,0,0,0,0,3,0,'N','B','N','Y','N','N',' ','C',
84,{ ' ' },{8,5,255},{72,8,2,{9,'A','S','Y','N','S','T','M','T',
'1',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',
' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '}}}};
RDTIN022.RdtCA = (char *)&sqlca;
RDTIN022.RdtRtCon = SQL_RDTRTCON;
TDARDI(&RDTIN022);

```

```
SQL_RDTRTCON = RDTIN022.RdtRtCon;
}
```

Examples 1-4

The following examples present WAIT statement SQL text without any client programming code context.

Example: A Basic WAIT Statement

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: A More Complicated WAIT Statement

The following example shows a more complicated WAIT statement that specifies two asynchronous statement identifiers. WAIT returns control to the program when both *req_1* and *req_2* complete.

```
WAIT req_1, req_2 COMPLETION
```

Example: Outstanding Asynchronous SQL Statements

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: Completing Outstanding Asynchronous SQL Statements

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](#) for information about how to submit an asynchronous request.
- See [TEST](#) for information about how to test the status of an asynchronous request without waiting for it to complete.

How to Read Syntax

This document uses the following syntax conventions.

Syntax Convention	Meaning
KEYWORD	Keyword. Spell exactly as shown. Many environments are case-insensitive. Syntax shows keywords in uppercase unless operating system restrictions require them to be lowercase or mixed-case.
<i>variable</i>	Variable. Replace with actual value.
<i>number</i>	String of one or more digits. Do not use commas in numbers with more than three digits. Example: 10045
[x]	x is optional.
[x y]	You can specify x, y, or nothing.
{ x y }	You must specify either x or y.
x [...]	You can repeat x, separating occurrences with spaces. Example: x x x See note after table.
x [, ...]	You can repeat x, separating occurrences with commas. Example: x, x, x See note after table.
x [delimiter...]	You can repeat x, separating occurrences with specified delimiter. Examples: <ul style="list-style-type: none"> If <i>delimiter</i> is semicolon: x; x; x If <i>delimiter</i> is { , OR }, you can do either of the following: <ul style="list-style-type: none"> x, x, x x OR x OR x See note after table.

Note:

You can repeat only the immediately preceding item. For example, if the syntax is:

```
KEYWORD x [...]
```

You can repeat x. Do not repeat KEYWORD.

If there is no white space between x and the delimiter, the repeatable item is x and the delimiter. For example, if the syntax is:

```
[ x, [...] ] y
```

- You can omit x: y
 - You can specify x once: x, y
 - You can repeat x and the delimiter: x, x, x, y
-

SQL Descriptor Area (SQLDA)

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 executed 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 runtime.

ANSI Compliance

The SQL Descriptor Area is ANSI/ISO SQL:2011-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](#)).

You cannot use INCLUDE SQLDA to define SQLDA in COBOL.

SQLDA Structure

The following table describes the fields of SQLDA.

Field Name	Format	Description
SQLDAID	CHARACTER(8)	Contains the characters SQLDA.
SQLDABC	INTEGER	<p>Length of the SQLDA calculated as $(16 + (44 * \text{SQLN value}))$ and set by the preprocessor when a DESCRIBE statement or PREPARE statement with an INTO clause is executed.</p> <p>For input, the application must set this field to the size of the SQLDA structure.</p> <p>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.</p>
SQLN	SHORT INTEGER	<p>Total number of elements in the SQLVAR array.</p> <p>The application sets this field to the number of elements available for use by the preprocessor (SQLVAR).</p> <p>For input, SQLN must be set prior to an OPEN or EXECUTE statement.</p> <p>For output, SQLN must be set prior to a DESCRIBE or PREPARE INTO request.</p> <p>If the BOTH option of the USING clause is used, then you must specify <i>at least</i> twice the number of SQLVAR elements as columns to be returned.</p>
SQLD	SHORT INTEGER	<p>Number of elements in the SQLVAR array currently used to hold variable descriptions.</p> <p>For input, the application sets this field to the number of SQLVAR elements used for input variable information.</p> <p>Value must be set prior to an OPEN or EXECUTE statement.</p> <p>For output, the preprocessor returns the number of SQLVAR elements that the DESCRIBE or PREPARE INTO request used.</p> <p>If too few elements are defined to satisfy the DESCRIBE, SQLD is set to the number required and no SQLVAR elements are returned.</p> <p>If the BOTH option of the USING clause is used, then you must specify <i>at least</i> twice the number of SQLVAR elements as columns to be returned.</p>
SQLVAR	array	<p>Contains a repeating second level structure that describes the columns of the result data.</p> <p>The structure of an SQLVAR element is as follows:</p> <ul style="list-style-type: none"> SQLTYPE (SHORT INTEGER) Contains a code indicating the data type of the column and its nullability attribute. <p>See SQLDA Data Type Codes for a discussion of data type codes.</p> <p>For input, the application sets the input host variable type prior to an OPEN or EXECUTE statement.</p>

Field Name	Format	Description
		<p>For output, the type is returned by performing a DESCRIBE statement.</p> <p>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.</p> <ul style="list-style-type: none"> • 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) <p>For input, the application sets the input host variable length prior to an OPEN or EXECUTE statement.</p> <p>For output, the preprocessor returns the data length of the column.</p> <p>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.</p>
SQLDATA	pointer	<p>Indicates to the preprocessor either:</p> <ul style="list-style-type: none"> • 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. <p>The application must set this field prior to performing the OPEN, EXECUTE, or FETCH statements.</p> <p>See <i>Teradata® Preprocessor2 for Embedded SQL Programmer Guide</i>, B035-2446 for examples of how to set addresses in COBOL.</p>
SQLIND	pointer	<p>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.</p>
SQLNAME	VARCHAR (30)	<p>Contains either of the following:</p> <ul style="list-style-type: none"> • The column name. • The column title. <p>For input this field has no meaning as an input host variable.</p> <p>For output, the preprocessor sets this field based on information in the USING clause.</p> <p>This field is used only by the application and has no further meaning to the preprocessor.</p>

SQLDA Data Type Codes

This topic lists the data type encodings used by the database and the embedded SQL preprocessor.

The database 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.

Locating Data Type Encodings

A data type encoding is contained in the two byte SQLTYPE INTEGER subfield of the SQLVAR field in the SQLDA.

How to Interpret The Nullability of SQL Data Type Encodings

Use these guidelines to interpret the tables in [SQL Data Type Encodings](#) and [Unused and Internally Used SQL Data Type Encodings](#).

IF the code is ...	THEN ...
nullable	indicator variables are allowed.
non-nullable	no indicator variables are allowed.

SQL Data Type Encodings

The rules for determining each encoding, given the non-nullable encoding for the data type, are as follows.

TO determine this encoding ...	ADD this value to the non-nullable encoding for the data type ...
nullable	1
stored procedure IN parameter	500
stored procedure INOUT parameter	501
stored procedure OUT parameter	502

The following table lists the SQL data types and their SQLDA data type encodings.

Data Type	Data Type Encodings				
	Non-Nullable	Nullable	IN	INOUT	OUT
BYTEINT	756	757	1256	1257	1258
SMALLINT	500	501	1000	1001	1002
INTEGER	496	497	996	997	998
BIGINT	600	601	1100	1101	1102

Data Type	Data Type Encodings				
	Non-Nullable	Nullable	IN	INOUT	OUT
DECIMAL	484	485	984	985	986
FLOAT/REAL/DOUBLE PRECISION	480	481	980	981	982
BYTE	692	693	1192	1193	1194
VARBYTE	688	689	1188	1189	1190
LONG VARBYTE	696	697	1192	1193	1194
BLOB	400	401	900	901	902
BLOB AS DEFERRED	404	405	904	905	906
BLOB AS LOCATOR	408	409	908	909	910
BLOB AS DEFERRED BY NAME	412	413	912	913	914
DATE (DateForm=ANSIDate)	748	749	1248	1249	1250
DATE (DateForm=IntegerDate)	752	753	1252	1253	1254
TIME	760	761	1260	1261	1262
TIMESTAMP	764	765	1264	1265	1266
TIME WITH TIME ZONE	768	769	1268	1269	1270
TIMESTAMP WITH TIME ZONE	772	773	1272	1273	1274
INTERVAL YEAR	776	777	1276	1277	1278
INTERVAL YEAR TO MONTH	780	781	1280	1281	1282
INTERVAL MONTH	784	785	1284	1285	1286
INTERVAL DAY	788	789	1288	1289	1290
INTERVAL DAY TO HOUR	792	793	1292	1293	1294
INTERVAL DAY TO MINUTE	796	797	1296	1297	1298
INTERVAL DAY TO SECOND	800	801	1300	1301	1302
INTERVAL HOUR	804	805	1304	1305	1306
INTERVAL HOUR TO MINUTE	808	809	1308	1309	1310
INTERVAL HOUR TO SECOND	812	813	1312	1313	1314
INTERVAL MINUTE	816	817	1316	1317	1318
INTERVAL MINUTE TO SECOND	820	821	1320	1321	1322
INTERVAL SECOND	824	825	1324	1325	1326

Data Type	Data Type Encodings				
	Non-Nullable	Nullable	IN	INOUT	OUT
CHARACTER	452	453	952	953	954
VARCHARACTER	448	449	948	949	950
LONG VARCHARACTER	456	457	956	957	958
CLOB	416	417	916	917	918
CLOB AS DEFERRED	420	421	920	921	922
CLOB AS LOCATOR	424	425	924	925	926
CLOB AS DEFERRED BY NAME	428	429	928	929	930
JSON Data Inline	880	881	1380	1381	1382
JSON Data Locator	884	885	1384	1385	1386
JSON Data Deferred	888	889	1388	1389	1390
GRAPHIC	468	469	968	969	970
VARGRAPHIC	464	465	964	965	966
LONG VARCHAR CHARACTER SET GRAPHIC	472	473	972	973	974
1-D ARRAY	504	505	1004	1005	1006
n-D Array	508	509	1008	1009	1010
UDT (both distinct and structured types)	not supported				

Unused and Internally Used SQL Data Type Encodings

The following table lists SQLDA data type codes that are either not used or are used internally so are not user-visible.

Data Type	Data Type Encodings				
	Non-Nullable	Nullable	IN	INOUT	OUT
ZONED DECIMAL (sign trailing)	432	433	932	933	934
ZONED DECIMAL (sign trailing separate)	436	437	936	937	938
ZONED DECIMAL (sign leading)	440	441	940	941	942
ZONED DECIMAL (sign leading separate)	444	445	944	945	946

The rules for determining each encoding, given the non-nullable encoding for the data type, are the same as those listed in [SQL Data Type Encodings](#).

SQL Communications Area (SQLCA)

Preprocessor2 can return program status and error information to applications in several possible ways.

The primary communication method for applications written to run in Teradata session 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 executed an embedded SQL statement.

Embedded SQL applications can also retrieve full diagnostic text by using a supplied routine (see [PPRTEXT](#)).

SQLCA cannot be used with stored procedures.

ANSI Compliance

The SQL Communications Area and the INCLUDE SQLCA statement are not ANSI-SQL 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 status variables (see [SQLSTATE Mappings](#) for information about mapping SQLCODE and SQLSTATE variables to one another and to database error messages).

The ANSI/ISO SQL-92 standard explicitly deprecates SQLCODE. The ANSI/ISO SQL-99 standard no longer defines 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.

Note:

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](#), [SQLCODE](#) and [SQLSTATE Mappings](#) for further information.

Defining the SQLCA For an Application

An application program typically defines the SQLCA using an INCLUDE SQLCA statement (see [INCLUDE SQLCA](#)).

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 execution of an embedded SQL or stored procedure statement to ensure that the statement completes successfully. You also should always write application code, or use appropriate condition handlers if the application is a stored procedure, to handle unacceptable status variable values.

Result Reporting

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	<p>an error occurred during processing.</p> <p>The application can determine the source of the error using the SQLCODE in conjunction with the first SQLERRD element.</p> <p>SQLERRD shows the following:</p> <ul style="list-style-type: none"> • Error conditions detected by the precompiler execution environment • Error conditions reported by CLI/TDP/database <p>The first SQLERRD element is zero when the error is detected directly by the preprocessor.</p> <p>These items are listed below:</p> <ul style="list-style-type: none"> • A list of error codes • The text associated with the code • A possible explanation • A possible solution for these errors <p>When an error condition is detected by CLI, TDP, or the database, the SQLCODE is set to the negative of the sum of the error code plus 10000.</p> <p>The application can look at the SQLCODE without interrogating the first SQLERRD element.</p> <p>Database error conditions are reported in the following distinct styles:</p> <ol style="list-style-type: none"> 1. Database codes that have similar or equivalent Database 2 (DB2) values are mapped to the DB2 value in the SQLCODE field.

WHEN SQLCODE is ...	THEN ...
	<p>See Retryable Errors.</p> <p>2. Database codes that have no similar or equivalent code have the SQLCODE value set to the negative of the database code.</p> <p>In either case, the first SQLERRD element contains the actual value returned by the database.</p> <p>Use care in distinguishing between a precompiler execution time error and a database error because some SQLCODE values are found in both categories of error.</p>
zero	processing was successful, though there might be warnings.
positive	<p>termination was normal.</p> <p>Positive values other than 0 or +100 indicate database warnings, such as the end-of-data reached for a request.</p>

PPRTEXT

Applications can obtain additional diagnostic help for a nonzero SQLCODE by invoking PPRTEXT.

PPRTEXT returns the error code (normally the same value as in the first SQLERRD element) and the text 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 four-byte integer field to contain the actual error code.
- A varying character field up to 255 characters long to contain the error text.
- A two-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 for Embedded SQL Programmer Guide*, B035-2446.

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 for Embedded SQL Programmer Guide*, B035-2446 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.

For a list of retryable error codes, see *Teradata Vantage™ - Database Messages*, B035-1096.

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 for Embedded SQL Programmer Guide*, B035-2446.

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:

Field Name	Format	Description
SQLCAID	CHARACTER(8)	Contains the characters 'SQLCA. '
SQLCABC	INTEGER	Length of the SQLCA (136 (x'88')).
SQLCODE	INTEGER	<p>Primary indicator of the result of SQL statement execution.</p> <ul style="list-style-type: none"> • If SQLCODE is 0, the statement executed normally. • If SQLCODE is positive, a non-error exception occurred. Examples are no more data was found or various non-fatal warnings. • If SQLCODE is negative, the statement failed because of an error condition. <p>The possible values for SQLCODE and their definitions are detailed in <i>Messages Reference</i>.</p>
SQLERRM	VARCHAR(70)	<p>Contains the error message inserts for the SQLCODE associated with variable information.</p> <p>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 are not.</p> <p>The string begins with a 16-bit word that contains the length of the remaining data.</p> <p>The data consists of as many as 70 characters of insert text, with the character X'FF' serving as a separator between inserts.</p> <p>If the inserts and separator characters are greater than 70 characters, then the array is truncated at the right.</p> <p>As an example, with a SQLCODE of -552, which is a privileges violation, SQLERRM contains the following three or four inserts:</p> <ul style="list-style-type: none"> • 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

Field Name	Format	Description
SQLERRP	CHARACTER(8)	Contains the name of the preprocessor module that detected the error.
SQLERRD	6-word array	<p>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.</p> <p>In order, the six words are the following:</p> <ul style="list-style-type: none"> • The CLlv2, TDP or database 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 the database, 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.
SQLWARN	CHARACTER(11) array	<p>Indicates the presence of warning conditions. Except for SQLWARN6, each character takes either the value pad character or 'W'.</p> <p>The 11 characters of SQLWARN are defined as follows:</p> <ul style="list-style-type: none"> • SQLWARN0 indicates whether any of the remaining ten warning codes have been set, as shown by the following: <ul style="list-style-type: none"> ◦ W means one or more of the other ten codes contains a 'W' or SQLWARN6 contains a 'W' or 'R'. ◦ pad character means the remaining ten characters are also pad characters. • For SQLWARN1: <ul style="list-style-type: none"> ◦ W means 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. ◦ pad character means that no truncation occurred. • For SQLWARN 2: <ul style="list-style-type: none"> ◦ W means that a warning has been issued by the database. The SQLCODE status variable contains the warning code. ◦ pad character means that no warning issued. • For SQLWARN 3: <ul style="list-style-type: none"> ◦ W means 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. ◦ pad character means that the number of columns returned by a SELECT was a match to the number of host variables in the INTO clause.

Field Name	Format	Description
		<ul style="list-style-type: none"> • SQLWARN 4 is reserved for future use. • SQLWARN 5 is reserved for future use.\ • For SQLWARN 6: <ul style="list-style-type: none"> ◦ W means that the statement caused Teradata SQL implicitly to roll back a unit of work. An example of this might be because a deadlock was detected. ◦ R means that a retryable error occurred. ◦ pad character that there was no rollback or error • SQLWARN 7 is reserved for future use. • SQLWARN8 is reserved for future use. • SQLWARN9 is reserved for future use. • SQLWARNA is reserved for future use.
SQLEXT	CHARACTER(5)	Contains the SQLSTATE value associated with the SQLCODE

Mapping SQLCODE Values to SQLSTATE Values

SQLCODE Rules

SQLCODE is not defined by the ANSI/ISO SQL-99 standard. Its use was deprecated by the ANSI/ISO 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/ISO 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 database 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 database errors have classes of T1 through T9, with the digit in the class corresponding to the first digit of the database error code.

The subclass contains the remaining 3 digits of the database error code.

For example: An 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.

SQLCODE to SQLSTATE Mapping Table

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		SQLCODE	SQLSTATE	
	Class	Subclass		Class	Subclass
100	02	000	-741	08	002
901	01	901	-742	08	000
902	01	004	-743	08	000
-101	54	001	-744	08	000
-104(1)	42(2)	512	-752	08	752
-302	22	024	-804	T0	804
-303	22	509	-811	21	000
-304	22	003	-822	51	004
-305	22	002	-840	21	840
-413	22	003	-901	T0	T10
-501	24	501	-925	56	021
-502	24	502	-926	56	021
-504	52	008	-942	T0	T12
-508	24	508	-943	24	000
-510	53	028	-1001	T0	T13
-514	24	000	-1002	T0	T14
-515	07	515	-1003	T0	T15
-563	08	003	-1005	T0	T16

SQLCODE	SQLSTATE		SQLCODE	SQLSTATE	
	Class	Subclass		Class	Subclass
-650	04	000	-1006	07	T17
-651	03	000	-1007	22	007
-652	04	000	-1009	22	T04
-653	41	000	-1010	T0	T18
-740	08	003	-1013	22	023
(1) This code should be -84, not -104. (2) This usage is not consistent with IBM DB2. DB2 uses class 37, which applies only to dynamic SQL. Code 2A applies to static SQL.					

Mapping CLI Codes to SQLCODE Values

If the error is generated by CLI, SQLCODE is set to CLI error code + 10000.

The only two exceptions are CLI 214 and 304. These two codes are mapped to -740 for the workstation platforms (Linux).

SQLSTATE Mappings

This section provides the complete set of SQLSTATE mappings for embedded SQL and stored procedure applications.

SQLSTATE Codes

A *SQLSTATE code* is a status value in embedded SQL programs and stored procedures that reflects 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 in apostrophes as in the following sample 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/ISO SQL-99-defined SQLSTATE classes (see [SQLSTATE Class Definitions](#)).

- 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 Database Error Messages to SQLSTATE Values](#)

SQLSTATE Class Definitions

ANSI defines the SQLSTATE class definitions provided in the following table. Vantage 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

Class Code	Definition
0A	Unsupported feature
0B	Nonvalid transaction initiation
0D	Nonvalid target type specification
0E	Nonvalid schema name list specification
0F	Locator exception
0K	Resignal when handler not active
0L	Nonvalid grammar
0M	Nonvalid SQL-invoked procedure reference
0N	SQL/XML mapping error
0P	Nonvalid role specification
0S	Nonvalid 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 execution
0X	Nonvalid 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	Nonvalid cursor state
25	Nonvalid transaction state
26	Nonvalid statement name
27	Triggered data change violation
28	Nonvalid authorization ID specification Note: Teradata SQL does not directly support the ANSI concept of an authorization ID. The ANSI authorization ID is essentially a database user.
2B	Dependent privilege descriptors still exist

Class Code	Definition
2C	Nonvalid character set name
2D	Nonvalid transaction termination
2E	Nonvalid connection name
2F	SQL routine exception
30	Nonvalid SQL statement
31	Nonvalid target specification value
33	Nonvalid SQLDA name
34	Nonvalid cursor name
35	Nonvalid condition number
36	Cursor sensitivity exception
38	External routine exception
39	External routine invocation exception
3B	Savepoint exception
3C	Ambiguous cursor name
3D	Nonvalid catalog name Note: Teradata SQL does not directly support the ANSI concept of a catalog. The ANSI catalog is essentially the same thing as the Data Dictionary.
3F	Nonvalid schema name Note: Teradata SQL does not directly support the ANSI concept of a catalog. The ANSI catalog is essentially the same thing as the Data Dictionary.
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

Class Code	Definition
	Note: The Call-Level Interface referred to is not Teradata CLIV2, but rather an ANSI-standard CLI that is a dialect of the Microsoft Open Database Connectivity, or ODBC, specification.
HZ	Remote database access condition
U0	User-defined exception

Mapping Database Error Messages to SQLSTATE Values

The following table lists database 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, Vantage 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.

The mapping of the following database codes to the SQLSTATE codes listed below is not consistent with IBM DB2. DB2 uses class 37, which applies only to dynamic SQL. Code 2A applies to static SQL:

- 3527
- 3529
- 3530
- 3568
- 3582
- 3617
- 3627
- 3628
- 3704
- 3731
- 3733
- 3751
- 3759
- 3760
- 3775
- 3789
- 3816
- 3817
- 3818
- 3820
- 3821

Database Error Message	SQLSTATE Code		Database 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
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

Database Error Message	SQLSTATE Code		Database Error Message	SQLSTATE Code	
	Class Value	Subclass Value		Class Value	Subclass Value
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
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

Database Error Message	SQLSTATE Code		Database Error Message	SQLSTATE Code	
	Class Value	Subclass Value		Class Value	Subclass Value
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
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

Database Error Message	SQLSTATE Code		Database Error Message	SQLSTATE Code	
	Class Value	Subclass Value		Class Value	Subclass Value
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
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

Database Error Message	SQLSTATE Code		Database Error Message	SQLSTATE Code	
	Class Value	Subclass Value		Class Value	Subclass Value
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
5758	30	758	5801	01	801
5802	01	802	5803	01	803
5804	01	804	5805	01	805
5806	01	806	5807	01	807
5808	01	808	5809	01	809
5810	01	810	5811	01	811
5812	01	812	5813	01	813
5814	01	814	5815	01	815

Database Error Message	SQLSTATE Code		Database Error Message	SQLSTATE Code	
	Class Value	Subclass Value		Class Value	Subclass Value
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
5977	30	977	5980	30	980
7593	22	593	7594	01	594
7601	20	000	7603	45	000
7604	35	000	7610	24	502
7627	21	000	7631	24	501
7632	02	000	7682	26	000
7683	07	005	9100	23	100
9101	23	101	9113	42	113
9114	22	113	9425	30	425
9127	23	127	9434	30	434
9435	22	435	9450	22	435

Mapping CLI Codes to SQLSTATE Values

The following table lists CLI codes mapped to their corresponding SQLSTATE codes.

CLI Code	SQLSTATE Code		CLI Code	SQLSTATE Code	
	Class Value	Subclass Value		Class Value	Subclass Value
36	08	T36	280	08	V80
40	08	T40	282	08	V82
41	08	T41	286	08	V86
151	08	U51	361	0A	W61
171	0A	U71	362	0A	W62
181	0A	U81	363	0A	W63
185	0A	U85	364	08	W64
190	0A	U90	365	08	W65
229	08	W29	370	08	W70
230	08	W30	375	08	W75
231	08	W31	376	08	W76
232	08	W32	377	08	W77
233	08	W33	378	08	W78
234	08	W34	379	08	W79
235	08	W35	380	08	W80
236	08	W36	381	08	W81
237	08	W37	382	08	W82
238	08	W38	383	08	W83
239	08	W39	384	08	W84
240	08	W40	385	08	W85
241	08	W41	386	08	W86
242	08	W42	387	08	W87
243	08	W43	426	08	X26
244	08	W44	427	08	X27
272	08	V72	428	08	X28
429	08	X29	524	08	524
430	08	X30	527	08	527
512	08	512	529	0A	529

CLI Code	SQLSTATE Code		CLI Code	SQLSTATE Code	
	Class Value	Subclass Value		Class Value	Subclass Value
513	08	513	530	2C	530
521	08	521	532	0A	532

SQL Stored Procedure Command Function Codes

Command Function Code Values

The following table lists the supported SQL stored procedure `COMMAND_FUNCTION` names and their `COMMAND_FUNCTION_CODE` values from the Diagnostics Area.

- Positive values indicate command functions defined by the ANSI/ISO SQL:2011 Standard
- Negative values indicate command functions that are Teradata extensions to the ANSI/ISO SQL:2011 Standard

COMMAND_FUNCTION	COMMAND_FUNCTION_CODE
ABORT	-34
ALTER TABLE	4
ALTER TRIGGER	-28
ASSIGNMENT	5
BEGIN LOGGING	-22
BEGIN TRANSACTION	-4
CALL	7
CASE	86
CLOSE CURSOR	9
COLLECT DEMOGRAPHICS	-36
COLLECT STATISTICS (QCD Form)	-24
COLLECT STATISTICS (Optimizer Form)	-24
COMMENT (Comment-Placing Form)	-3
COMMENT (Comment-Retrieving Form)	-3
COMMIT WORK	11
CREATE CAST	52
CREATE DATABASE	-12
CREATE HASH INDEX	-38
CREATE INDEX	-11
CREATE JOIN INDEX	-39

COMMAND_FUNCTION	COMMAND_FUNCTION_CODE
CREATE MACRO	-6
CREATE ORDERING	114
CREATE PROFILE	-31
CREATE ROLE	61
CREATE TABLE	77
CREATE TRANSFORM	117
CREATE TRIGGER	80
CREATE USER	-13
CREATE VIEW	84
DEALLOCATE PREPARE	16
DECLARE VARIABLE	96
DELETE CURSOR	18
DELETE DATABASE	-19
DELETE USER	-42
DELETE WHERE	19
DROP CAST	78
DROP DATABASE	-15
DROP HASH INDEX	-40
DROP INDEX	-37
DROP JOIN INDEX	-41
DROP MACRO	-7
DROP ORDERING	115
DROP PROCEDURE	-29
DROP PROFILE	-33
DROP ROLE	29
DROP STATISTICS (QCD Form)	-25
DROP STATISTICS (Optimizer Form)	-25
DROP TABLE	32
DROP TRANSFORM	116

COMMAND_FUNCTION	COMMAND_FUNCTION_CODE
DROP TRIGGER	34
DROP USER	-43
DROP VIEW	36
DYNAMIC CLOSE	37
DYNAMIC DELETE CURSOR	38
DYNAMIC FETCH	39
DYNAMIC OPEN	40
DYNAMIC SQL	-1
DYNAMIC UPDATE CURSOR	42
END LOGGING	-23
END TRANSACTION	-5
FETCH	45
FOR	46
GET DIAGNOSTICS	-51
GIVE	14
GRANT	48
GRANT LOGON	-20
GRANT ROLE	49
IF	88
INSERT	50
MERGE	128
MODIFY DATABASE	-16
MODIFY PROFILE	-32
MODIFY USER	-35
MULTI STATEMENT	-49
OPEN	53
PREPARE	56
QUERY_BAND	-52
RENAME FUNCTION	-50

COMMAND_FUNCTION	COMMAND_FUNCTION_CODE
RENAME MACRO	-10
RENAME PROCEDURE	-30
RENAME TABLE	-8
RENAME TRIGGER	-26
RENAME VIEW	-9
REPEAT	95
REPLACE CAST	-48
REPLACE MACRO	-18
REPLACE ORDERING	-45
REPLACE TRANSFORM	-44
REPLACE TRIGGER	-27
REPLACE VIEW	-17
RESIGNAL	91
REVOKE	59
REVOKE LOGON	-21
REVOKE ROLE	129
ROLLBACK WORK	62
SELECT	65
SIGNAL	92
UPDATE CURSOR	81
UPDATE WHERE	82
WHILE	97

Performance Considerations

This section provides suggestions for improving database query performance.

Using Updatable Cursors to Optimize Query Design

In ANSI mode, you can define a cursor for the query results and for every row in the query results in order to update or delete the data row via the cursor associated with the row.

This means that update and delete operations do not identify a search condition; instead, they identify a cursor (or a pointer) to a specific row to be updated or deleted.

You can use pocketable cursors to update each row of a select result independently as it is processed.

Recommendations

To reap the full benefit from the updatable cursor feature, you should minimize:

- The size of query result and number of updates/transaction
- The length of time you hold the cursor open

Using many updates per cursor may not be optimal because:

- They block other transactions.
- The system requires longer rollbacks.

In this case, use the MultiLoad utility to do updates.

Additional Information

Teradata Links

Link	Description
https://docs.teradata.com/	Search Teradata Documentation, customize content to your needs, and download PDFs. Customers: Log in to access Orange Books.
https://support.teradata.com	One-stop source for Teradata community support, software downloads, and product information. Log in for customer access to: <ul style="list-style-type: none">• Community support• Software updates• Knowledge articles
https://www.teradata.com/University/Overview	Teradata education network
https://support.teradata.com/community	Link to Teradata community