

Teradata Vantage™ - SQL Data Definition Language

Syntax and Examples

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 Data Definition Language Syntax and Examples	8
Changes and Additions	8
Chapter 2: SQL DDL Statements	10
How the Statements are Organized	10
List of SQL Statements and Purposes	10
Chapter 3: Table Statements	21
CREATE TABLE and CREATE TABLE AS	21
CREATE TABLE (Queue Table Form)	177
CREATE GLOBAL TEMPORARY TRACE TABLE	199
CREATE FOREIGN TABLE	209
CREATE ERROR TABLE	254
ALTER TABLE	257
ALTER TABLE (Map and Colocation Form)	387
ALTER FOREIGN TABLE	390
ALTER TABLE TO CURRENT	398
RENAME TABLE	408
DROP TABLE	410
DROP ERROR TABLE	412
HELP COLUMN	413
HELP CONSTRAINT	446
HELP ERROR TABLE	449
HELP TABLE	451
HELP VOLATILE TABLE	460
SHOW TABLE	462
Chapter 4: View Statements	478
CREATE VIEW and REPLACE VIEW	478
CREATE RECURSIVE VIEW and REPLACE RECURSIVE VIEW	511
RENAME VIEW	532
DROP VIEW	533
HELP VIEW	534
Chapter 5: Index Statements	537
CREATE INDEX	537
CREATE JOIN INDEX	545
CREATE HASH INDEX	619
ALTER JOIN INDEX	635

ALTER HASH INDEX	636
DROP INDEX	637
DROP JOIN INDEX	642
DROP HASH INDEX	643
HELP INDEX	644
HELP JOIN INDEX	650
HELP HASH INDEX	651
Chapter 6: Authorization Statements for External Routines	653
CREATE AUTHORIZATION and REPLACE AUTHORIZATION	653
DROP AUTHORIZATION	659
Chapter 7: Global and Persistent (GLOP) Data Statements	661
CREATE GLOP SET	661
DROP GLOP SET	662
Chapter 8: Procedure Statements	664
CREATE PROCEDURE and REPLACE PROCEDURE (External Form)	664
CREATE PROCEDURE and REPLACE PROCEDURE (SQL Form)	690
ALTER PROCEDURE (SQL Form)	741
ALTER PROCEDURE (External Form)	747
RENAME PROCEDURE	751
DROP PROCEDURE	753
HELP PROCEDURE	754
Chapter 9: Macro Statements	759
CREATE MACRO and REPLACE MACRO	759
RENAME MACRO	767
DROP MACRO	768
HELP MACRO	769
Chapter 10: User-Defined Function Statements	772
CREATE FUNCTION and REPLACE FUNCTION (SQL Form)	772
CREATE FUNCTION and REPLACE FUNCTION (Table Form)	786
CREATE FUNCTION and REPLACE FUNCTION (External Form)	812
CREATE FUNCTION MAPPING and REPLACE FUNCTION MAPPING	847
ALTER FUNCTION	864
DROP FUNCTION	868
DROP FUNCTION MAPPING	873
HELP FUNCTION	874
RENAME FUNCTION (SQL Form)	882
RENAME FUNCTION (External Form)	885
SHOW FUNCTION MAPPING	889
Chapter 11: User-Defined Method Statements	892

CREATE METHOD	892
ALTER METHOD	905
REPLACE METHOD	908
HELP METHOD	915
Chapter 12: User-Defined Type Statements	920
CREATE TYPE (Structured Form)	920
CREATE TYPE (Distinct Form)	931
CREATE TYPE (ARRAY/VARRAY Form)	940
CREATE <i>storage_format</i> SCHEMA	952
CREATE CAST and REPLACE CAST	954
CREATE ORDERING and REPLACE ORDERING	959
CREATE TRANSFORM and REPLACE TRANSFORM	964
SET TRANSFORM GROUP FOR TYPE	973
ALTER TYPE	975
DROP TYPE	986
DROP <i>storage_format</i> SCHEMA	988
DROP CAST	988
DROP ORDERING	990
DROP TRANSFORM	991
HELP TYPE	992
HELP <i>storage_format</i> SCHEMA	1002
HELP CAST	1003
HELP TRANSFORM	1005
Chapter 13: Database Statements	1007
CREATE DATABASE	1007
DELETE DATABASE	1019
MODIFY DATABASE	1020
DATABASE	1031
DROP DATABASE	1032
HELP DATABASE	1034
LOGGING INCREMENTAL ARCHIVE ON FOR <i>object_list</i>	1036
LOGGING INCREMENTAL ARCHIVE OFF FOR <i>object_list</i>	1037
INCREMENTAL RESTORE ALLOW WRITE FOR <i>object_list</i>	1038
Chapter 14: User, Profile, and Role Statements	1040
CREATE PROFILE	1040
MODIFY PROFILE	1058
CREATE ROLE	1073
CREATE USER	1074
MODIFY USER	1103
SET ROLE	1129
DELETE USER	1131
DROP PROFILE	1132

DROP ROLE	1134
DROP USER	1135
HELP USER	1137
Chapter 15: Map Statements	1139
CREATE MAP	1139
DROP MAP	1140
SHOW MAP	1141
Chapter 16: Load Isolation Statements	1143
BEGIN ISOLATED LOADING	1143
LDILoadGroup Query Band	1145
CHECKPOINT ISOLATED LOADING	1145
END ISOLATED LOADING	1146
Chapter 17: Secure Zones Statements	1148
CREATE ZONE	1148
ALTER ZONE	1149
DROP ZONE	1151
Chapter 18: Session Statements	1152
SET SESSION	1152
SET SESSION ACCOUNT	1153
SET SESSION CALENDAR	1155
SET SESSION CHARACTER SET UNICODE PASS THROUGH	1156
SET SESSION COLLATION	1157
SET SESSION CONSTRAINT	1163
SET SESSION DATABASE	1176
SET SESSION DATEFORM	1176
SET SESSION DEBUG FUNCTION	1178
SET SESSION DOT NOTATION	1180
SET SESSION FOR ISOLATED LOADING	1181
SET SESSION FUNCTION TRACE	1182
SET SESSION JSON IGNORE ERRORS	1184
SET SESSION SEARCHUIFDBPATH	1185
SET SESSION TRANSACTION ISOLATION LEVEL	1186
SET QUERY_BAND	1189
SET TIME ZONE	1206
SET SESSION UDFSEARCHPATH	1210
HELP SESSION	1211
Chapter 19: Logging Statements	1223
BEGIN LOGGING	1223
BEGIN QUERY CAPTURE	1239
BEGIN QUERY LOGGING	1244

FLUSH QUERY LOGGING	1261
REPLACE QUERY LOGGING	1263
END LOGGING	1271
END QUERY CAPTURE	1278
END QUERY LOGGING	1279
SHOW QUERY LOGGING	1282
Chapter 20: Statistics Statements	1304
COLLECT STATISTICS (Optimizer Form)	1304
SHOW STATISTICS	1331
DROP STATISTICS (Optimizer Form)	1353
HELP STATISTICS (Optimizer Form)	1359
HELP STATISTICS (QCD Form)	1364
Chapter 21: Row-Level Security Constraint Statements	1368
CREATE CONSTRAINT	1368
ALTER CONSTRAINT	1371
DROP CONSTRAINT	1374
Chapter 22: Triggers Statements	1377
ALTER TRIGGER	1377
CREATE TRIGGER and REPLACE TRIGGER	1378
DROP TRIGGER	1399
HELP TRIGGER	1400
RENAME TRIGGER	1402
Chapter 23: Comment, Help, and Show Statements	1404
COMMENT (Comment Placing Form)	1404
HELP ONLINE	1411
SHOW <i>object</i>	1412
SHOW <i>request</i>	1433
Appendix A: Notation Conventions	1446
Appendix B: Object Data Types	1449
Appendix C: Additional Information	1451

Introduction to SQL Data Definition Language Syntax and Examples

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.

This document describes the Teradata SQL Data Definition Language (DDL) statements used to define or restructure the database. Teradata SQL is an ANSI compliant product. Teradata has its own extensions to the language.

Changes and Additions

Date	Description
July 2021	<p>New Statement: ALTER FOREIGN TABLE</p> <p>Support for Named JSON Arrays Updated JSON External Files.</p> <p>Error Tables for Foreign Tables Updated CREATE FOREIGN TABLE.</p> <p>Google Cloud Authorizations for Foreign Tables Added Example: Creating a Google Cloud Authorization.</p> <p>Fallback Subtables Not Compressed by Default</p> <ul style="list-style-type: none"> Updated ALTER TABLE. Updated CREATE TABLE and CREATE TABLE AS. Updated SET QUERY BAND. <p>FAST MODE for ALTER TABLE Statement Updated ALTER TABLE.</p> <p>Azure LOCATION Syntax for NOS</p> <ul style="list-style-type: none"> Updated LOCATION. Updated LOCATION Key Prefix Best Practices. <p>AUTHORIZATION for READ_NOS Updated CREATE AUTHORIZATION and REPLACE AUTHORIZATION.</p> <p>USING Clause for CREATE AUTHORIZATION Updated CREATE AUTHORIZATION and REPLACE AUTHORIZATION.</p> <p>INVOKER and DEFINER, which appear in the syntax of the following statements, are deprecated for NOS:</p> <ul style="list-style-type: none"> CREATE FOREIGN TABLE CREATE AUTHORIZATION and REPLACE AUTHORIZATION CREATE FUNCTION MAPPING and REPLACE FUNCTION MAPPING <p>Miscellaneous Changes</p> <ul style="list-style-type: none"> Corrected CREATE PROFILE Syntax. Corrected MODIFY PROFILE Syntax. Corrected DROP PROFILE Syntax. Corrected description of FOR USER in PASSWORD clause of MODIFY USER.

Date	Description
	<ul style="list-style-type: none"> Updated CREATE FOREIGN TABLE syntax, syntax element descriptions, and examples. Updated description of WHEN clause in CREATE TRIGGER and REPLACE TRIGGER Syntax Elements. <p>Note: 17.05 syntax is deprecated (supported but not recommended).</p>
June 2020	<p>UDF Search Path Added SET SESSION UDFSEARCHPATH</p> <p>Native Object Store Added CREATE FOREIGN TABLE, including syntax, options, usage notes, and examples. Updated DROP TABLE to add the FOREIGN keyword. Updated COLLECT STATISTICS (Optimizer Form) to add Example: Collecting Statistics on a Foreign Table Column. Added Example: SHOW TABLE for Foreign Table. Updated CREATE AUTHORIZATION and REPLACE AUTHORIZATION to include the USING clause, where you can specify security parameters for accessing remote repositories. Updated CREATE FUNCTION MAPPING and REPLACE FUNCTION MAPPING to add the EXTERNAL SECURITY clause. Updated CREATE storage_format SCHEMA to add Example: Schema for DATASET Type with CSV Storage Format.</p> <p>Incremental Restore Added LOGGING INCREMENTAL ARCHIVE ON FOR object_list, LOGGING INCREMENTAL ARCHIVE OFF FOR object_list, and INCREMENTAL RESTORE ALLOW WRITE FOR object_list.</p> <p>JSON Auto Composition and Shredding Updated CREATE TABLE and CREATE TABLE AS column attributes to add the AUTO COLUMN option. Added AUTO COLUMN and Examples: Creating a Table with a JSON Auto Column. Updated ALTER TABLE column attributes to add the AUTO COLUMN option. Added AUTO COLUMN and Examples: Adding and Removing the JSON Auto Column Option. Added Example: HELP COLUMN for a JSON Column with the Auto Column Option and Example: HELP TABLE with a JSON Auto Column. Added Example: SHOW TABLE with an Auto Column.</p> <p>Function Mapping Variable Substitution Updated CREATE FUNCTION MAPPING and REPLACE FUNCTION MAPPING.</p> <p>Function Mapping for Native Functions Updated CREATE FUNCTION MAPPING and REPLACE FUNCTION MAPPING.</p> <p>Algorithm 3 is the new default mode for DBQL Logging The Database Query Log (DBQL) is enhanced for Advanced SQL Engine 17.00 with more complete and accurate logging. DBQL now uses Algorithm 3 by default, which includes collecting statistics on aborted and parallel steps, and results in more accurate resource usage statistics. Upgraded systems that were not previously using Algorithm 3 will not use Algorithm 3 until explicitly configured to do so. See BEGIN QUERY LOGGING m and REPLACE QUERY LOGGING m.</p>

SQL DDL Statements

How the Statements are Organized

Statements are grouped according to function. In each section, statements appear in the order that you typically use them. For example, statements are ordered according to the first word as follows:

- CREATE, ADD, BEGIN, SET
- ALTER, CHANGE, FLUSH, MODIFY, RENAME, REPLACE
- DROP, DELETE, END
- SHOW
- HELP

You can also refer to the following alphabetical list of statements.

List of SQL Statements and Purposes

This table alphabetically lists SQL statements and their purpose.

Statement	Purpose
ALTER CONSTRAINT	Modifies the definition of an existing row-level security constraint object.
ALTER FUNCTION	<p>Performs either or both of the following functions.</p> <ul style="list-style-type: none"> • Controls whether an existing function can run in protected mode as a separate process or in unprotected mode as part of the database. • Recompiles C or C++ functions or relinks Java functions and redistributes them. <p>Note: Java UDFs must always run in protected mode, so you cannot use this statement to change their protection mode.</p>
ALTER HASH INDEX	Moves a hash index from one map to another or changes the colocation for a sparse map.
ALTER JOIN INDEX	Moves a join index from one map to another or changes the colocation for a sparse map.
ALTER METHOD	<p>Performs either or both of the following functions.</p> <ul style="list-style-type: none"> • Controls whether an existing method can run in protected mode as a separate process or in non-protected mode as part of the database. • Recompiles or relinks the method and redistributes it.

Statement	Purpose
ALTER PROCEDURE (External Form)	Recompiles an existing external procedure and allows changes in the following compile time attributes of the procedure. <ul style="list-style-type: none"> • Generate a new library for the recompiled procedure or not. • Toggle the protection mode between protected and unprotected states. • Change the creation time zone.
ALTER PROCEDURE (SQL Form)	Recompiles an existing SQL procedure and allows changes in the following compile time attributes of the procedure. <ul style="list-style-type: none"> • SPL option • WARNING option • AT TIME ZONE option
ALTER TABLE	Add one or more columns to a table or global temporary table, add or change attributes and options, including partitioning, constraints, and compression. You can also drop columns, change a join index, or revalidate a table.
ALTER TABLE (Map and Colocation Form)	Moves a table from one map to another. You can also change the colocation name for a sparse map.
ALTER TABLE TO CURRENT	Reconciles the row partitioning for a table or uncompressed join index to a newly resolved date or timestamp when its partitioning is based on the DATE, CURRENT_DATE, or CURRENT_TIMESTAMP functions.
ALTER TRIGGER	Enables or disables a trigger or changes its creation timestamp.
ALTER TYPE	Performs any of the following operations for a UDT. <ul style="list-style-type: none"> • Add a new attribute to a structured UDT definition. • Drop an existing attribute from a structured UDT definition. • Add a method to a distinct or structured UDT definition. • Drop a method from a distinct or structured UDT definition. • Recompile the source code for a distinct, structured, ARRAY, or VARRAY type definition.
ALTER ZONE	Add a database or user as the root to a secure zone or remove the root database or user from a secure zone.
BEGIN ISOLATED LOADING	Starts an explicit concurrent isolated load operation on a load isolated (LDI) table. You can perform concurrent read operations on committed rows while the table is being loaded.
BEGIN LOGGING	Starts the auditing of SQL requests that attempt to access data.
BEGIN QUERY CAPTURE	Starts the logging of database request information, including some database object creation and drop information.
BEGIN QUERY LOGGING	Collects demographic data for one or more columns, computes a statistical profile of the collected data, stores the synopsis in DBC.StatsTbl

Statement	Purpose
	in the data dictionary, and optionally copies the statistics for one or more columns to a duplicate target table.
CHECKPOINT ISOLATED LOADING	Sets a checkpoint for the explicit isolated load operation and commits the data that has been loaded up to that point.
COLLECT STATISTICS (Optimizer Form)	Collects demographic data for one or more columns, computes a statistical profile of the collected data, stores the synopsis in DBC.StatsTbl in the data dictionary, and optionally copies the statistics for one or more columns to a duplicate target table.
COMMENT (Comment Placing Form)	Creates a user-defined description of a user-defined database object or definition in the data dictionary.
CREATE AUTHORIZATION	Creates or replaces an authorization object.
CREATE CAST	Creates a cast operation for a UDT.
CREATE CONSTRAINT	Creates the SQL row-level security constraint definition and associates it with specific UDFs to enforce that constraint.
CREATE DATABASE	Creates a database in which other database objects can be created.
CREATE ERROR TABLE	Defines the name and containing database of a new error table and specifies the name of the associated data table.
CREATE FOREIGN TABLE	Foreign tables enable Vantage to access data in external object storage, such as semi-structured and unstructured data in Amazon S3, Azure Blob storage, and Google Cloud Storage. In-database integration of this data allows data scientists and analysts to read and process this data with Vantage, using standard SQL. You can join external data to relational data in Vantage, and process it using built-in Vantage analytics and functions.
CREATE FUNCTION (External Form)	Compiles and installs an external UDF and creates or replaces the SQL function definition used to invoke that UDF.
CREATE FUNCTION (Table Form)	Creates a table function definition.
CREATE FUNCTION (SQL Form)	Creates an SQL UDF.
CREATE FUNCTION MAPPING	Creates a function mapping.
CREATE GLOBAL TEMPORARY TRACE TABLE	Creates a global temporary trace table to support the UDF- and external SQL procedure-related trace option of the SET SESSION statement. See SET SESSION FUNCTION TRACE .
CREATE GLOP SET	Creates the definition of a global persistent memory set.
CREATE HASH INDEX	Creates a hash index on a base table.

Statement	Purpose
CREATE INDEX	Creates a secondary index on an existing data table or join index.
CREATE JOIN INDEX	Creates a join index for one or multiple tables, optionally with aggregation.
CREATE MACRO	Defines a set of statements that are frequently used or that perform a complex operation. The statements in the macro body are submitted when the macro is invoked by a subsequent EXECUTE statement.
CREATE MAP	Creates a sparse map.
CREATE METHOD	Defines the body of a method that is associated with a user-defined data type.
CREATE ORDERING	Creates a map ordering routine used to compare UDT values.
CREATE PROCEDURE (External Form)	Compiles and installs an external SQL procedure routine and creates or replaces the SQL definition used to invoke the procedure.
CREATE PROCEDURE (SQL Form)	Directs the SQL procedure compiler to create a procedure from the SQL statements in the remainder of the source text and creates the SQL definition used to invoke the procedure.
CREATE PROFILE	Creates a profile that defines a set of parameters. You can then assign the profile to multiple users.
CREATE RECURSIVE VIEW	Creates or replaces a recursive view definition.
CREATE ROLE	Creates a role for managing user access privileges on database objects. A role is a shell database object to which sets of privileges can be granted using GRANT requests.
CREATE storage_format SCHEMA	Creates a schema for a specific storage format of the DATASET type.
CREATE TABLE and CREATE TABLE AS	Defines the column names, column data types and attributes, primary and secondary indexes, column- and table-constraints, partitioning, and other attributes of a new table. The CREATE TABLE AS form copies column definitions to a new table. Optionally, data and statistics are copied to the new table.
CREATE TABLE (Queue Table Form)	Creates a queue table.
CREATE TRANSFORM	Creates transform groups for importing UDT data from a client system to Vantage and exporting UDT data from Vantage to a client system. You can use CREATE TRANSFORM to add transform groups in addition to the existing transform groups.
CREATE TRIGGER	Creates a new trigger definition.

Statement	Purpose
CREATE TYPE (ARRAY/VARRAY Form)	Creates a user-defined ARRAY or VARRAY data type that is constructed from a predefined Vantage data type, a distinct UDT data type, a structured UDT data type, or an internal UDT data type.
CREATE TYPE (Distinct Form)	Creates a user-defined data type that is constructed directly from a predefined Vantage data type.
CREATE TYPE (Structured Form)	Creates the body of a structured user-defined data type.
CREATE USER	Creates a permanent database user object, in which other database objects may be created, with a mandatory permanent space allocation and a mandatory password plus optional attributes.
CREATE VIEW	Creates a view on a set of tables or views or both.
CREATE ZONE	Defines a secure zone. You can also specify an existing database or user as the zone root.
DATABASE	Establishes a new default database for the current session for SQL requests that do not have fully-qualified table, view, or macro names.
DELETE DATABASE	Deletes all data tables, views, triggers, SQL procedures, macros, and user-installed files (UIFs) from a database.
DELETE USER	Deletes all data tables, views, triggers, SQL procedures, macros, and user-installed files (UIFs) from a user.
DROP AUTHORIZATION	Drops an authorization object.
DROP CAST	Drops a cast definition for a UDT from the data dictionary.
DROP CONSTRAINT	Drops the definition of a row-level security constraint object from the data dictionary.
DROP DATABASE	Drops the definition for an empty database from the Data Dictionary.
DROP ERROR TABLE	Deletes the specified error table object definition from the Data Dictionary and from the containing database or user.
DROP FUNCTION	Drops the definition of the specified function or specific function from the Data Dictionary and from the containing database or user.
DROP FUNCTION MAPPING	Drops a function mapping.
DROP GLOP SET	Drops the definition of the specified GLOP set from the Data Dictionary.
DROP HASH INDEX	Drops a hash index on a table.
DROP INDEX	Drops a secondary index on a table or join index.

Statement	Purpose
DROP JOIN INDEX	Drops the join index.
DROP MACRO	Drops the definition for the specified macro from the dictionary.
DROP MAP	Drop a contiguous or sparse map.
DROP ORDERING	Drops the ordering definition for a UDT.
DROP PROCEDURE	Drops the definition for the specified procedure from the Data Dictionary and from the containing database or user.
DROP PROFILE	Drops a specified profile.
DROP ROLE	Drops a specified role.
DROP storage format SCHEMA	Drops a schema.
DROP STATISTICS (Optimizer Form)	Drops the statistical data that was collected for specified columns of a table, hash index, or join index by a COLLECT STATISTICS (Optimizer Form) request. To drop the SUMMARY statistics for a database object, drop all statistics on that object.
DROP TABLE	Drops the definition for a specified table from the Data Dictionary and from the containing database or user, depending on the keyword specified.
DROP TRANSFORM	Drop transform groups for a specified UDT.
DROP TRIGGER	Drops the definition for the specified trigger from its subject table.
DROP TYPE	Drops the definition for a specified UDT.
DROP USER	Drops the definition for an empty user from the Data Dictionary.
DROP VIEW	Drops the definition of a specified view from the Data Dictionary.
DROP ZONE	Removes a secure zone.
END ISOLATED LOADING	Ends the explicit concurrent isolated load operation for the specified load group value.
END LOGGING	Ends the auditing of SQL requests that was started with a BEGIN LOGGING request.
END QUERY CAPTURE	Stops the capture of SQL requests initiated by a BEGIN QUERY CAPTURE request for the session.
END QUERY LOGGING	Stops the logging of SQL requests initiated by a BEGIN QUERY LOGGING request and commits the query log cache.

Statement	Purpose
FLUSH QUERY LOGGING	Flushes one, several, or all DBQL caches or workload management caches to disk.
HELP CAST	Returns the available cast operations for the specified UDT.
HELP COLUMN	Displays the attributes of a column, including whether it is a single-column primary or secondary index and, if so, whether it is unique.
HELP CONSTRAINT	Displays the attributes for a specific named constraint on a table. Use the SHOW TABLE statement to obtain unnamed constraint information. See “SHOW object.”
HELP DATABASE	Displays the attributes, sorted by object name, for all tables, views, join indexes, hash indexes, SQL procedures, user-defined functions, and macros contained by a specified database.
HELP ERROR TABLE	Displays the attributes for a specified error table.
HELP FUNCTION	Reports the specific function name, list of parameters, the data types of the parameters, whether the function is used to compress or decompress character or graphic data, and any comments associated with the parameters for SQL, scalar, aggregate, and table functions.
HELP HASH INDEX	Displays the data types of the columns defined by a particular hash index.
HELP INDEX	Displays the attributes for the primary and secondary indexes defined for a base data table, hash index, or join index.
HELP JOIN INDEX	Displays the attributes of the columns defined by a particular join index.
HELP MACRO	Displays the attributes for the specified macro.
HELP METHOD	Displays the parameter list of the specified method.
HELP ONLINE	Displays syntax help for any SQL statement or client utility command.
HELP PROCEDURE	Displays the attribute and format parameters for each parameter of a procedure or just the creation time attributes for the specified procedure.
HELP SESSION	Displays attribute information for the user of the current session or just the row-level constraint attribute information for the user of the current session.
HELP STATISTICS (Optimizer Form)	Displays the attribute and format parameters for each parameter of a procedure or just the creation time attributes for the specified procedure.
HELP STATISTICS (QCD Form)	Displays summary or detailed attributes for the statistics that have been collected in the TableStatistics table of a specified query capture database (QCD) for a specified table.

Statement	Purpose
HELP storage format SCHEMA	Displays the name of a schema, the storage format of the DATASET type associated with the schema, and the length of the schema. The length is listed in UNICODE characters.
HELP TABLE	Displays the attributes for a specified base data table.
HELP TRANSFORM	Reports the fromsql and tosql transform routines for a specified UDT or predefined data type.
HELP TRIGGER	Displays the attributes for a specified trigger.
HELP TYPE	Displays the attributes for a UDT and, optionally, all the attributes of a structured UDT or all the methods associated with the UDT.
HELP USER	Displays the attributes, sorted by object name, for all tables, views, join indexes, hash indexes, SQL procedures, user-defined functions, and macros contained by the specified user.
HELP VIEW	Displays the attributes for the specified view or recursive view.
HELP VOLATILE TABLE	Displays the attributes for the requested volatile table.
HELP ONLINE	Displays syntax help for any SQL statement or client utility command.
INCREMENTAL RESTORE ALLOW WRITE FOR object_list	Enables read and write access for the databases, users, or tables you specify after an incremental restore. Incremental restore sets tables to read-only access.
LOGGING INCREMENTAL ARCHIVE OFF FOR object_list	Disables incremental restore for databases and tables you specify.
LOGGING INCREMENTAL ARCHIVE ON FOR object_list	Enables incremental restore for databases and tables you specify.
MODIFY DATABASE	Changes the parameters for the specified database.
MODIFY PROFILE	Changes the parameters for the specified profile.
MODIFY USER	Changes parameters assigned to the specified user.
RENAME FUNCTION (External Form)	Renames either the overloaded calling function name or specific function name for an external function.
RENAME FUNCTION (SQL Form)	Renames either the overloaded calling function name or specific function name for an SQL function.
RENAME MACRO	Renames an existing macro.
RENAME PROCEDURE	Renames an existing SQL procedure.

Statement	Purpose
RENAME TABLE	Renames an existing table.
RENAME TRIGGER	Renames an existing trigger.
RENAME VIEW	Renames an existing view.
REPLACE AUTHORIZATION	Creates or replaces an authorization object.
REPLACE CAST	Replaces a cast operation for a UDT.
REPLACE FUNCTION (External Form)	Compiles and installs an external UDF and creates or replaces the SQL function definition used to invoke that UDF.
REPLACE FUNCTION (SQL Form)	Creates or replaces an SQL UDF.
REPLACE FUNCTION MAPPING	Replaces the definition of an existing function mapping or, if the specified function mapping does not exist, creates a new function mapping by that name.
REPLACE MACRO	REPLACE MACRO redefines an existing macro. If the specified macro does not exist, REPLACE MACRO creates a new macro with that name.
REPLACE METHOD	Creates or replaces an existing method definition.
REPLACE ORDERING	Creates or replaces a map ordering routine used to compare UDT values.
REPLACE PROCEDURE (External Form)	Compiles and installs an external SQL procedure routine and creates or replaces the SQL definition used to invoke the procedure.
REPLACE PROCEDURE (SQL Form)	Directs the SQL procedure compiler to create or replace a procedure from the SQL statements in the remainder of the source text and creates the SQL definition used to invoke the procedure.
REPLACE QUERY LOGGING	Creates a rule or replaces the current rule with the rule you specify.
REPLACE RECURSIVE VIEW	Creates or replaces a recursive view definition.
REPLACE TRANSFORM	Creates or replaces transform groups for importing UDT data from a client system to Vantage and exporting UDT data from Vantage to a client system. You can use CREATE TRANSFORM to add transform groups in addition to the existing transform groups. REPLACE TRANSFORM drops all of the existing transform groups and adds the transform groups you specify.
REPLACE TRIGGER	Creates a new trigger or replaces a trigger definition.
REPLACE VIEW	Creates or replaces a view on a set of tables or views or both.
SET QUERY_BAND	Sets or removes a query band for the current session or transaction.

Statement	Purpose
SET ROLE	Sets the current role for a session. It does not distinguish between directory- and database-managed roles.
SET SESSION	Allows the setting of various session parameters for the entire session or individual requests within the session.
SET SESSION ACCOUNT	Dynamically changes your account or account priorities for the duration of a session or for one SQL request only.
SET SESSION CALENDAR	Sets the default calendar for the session to a system-defined calendar.
SET SESSION CHARACTER SET UNICODE PASS THROUGH	Enables or disables Unicode Pass Through processing for the session. Pass Through Characters (PTCs) include the Unicode characters that are not currently supported and character codes reserved for future use.
SET SESSION COLLATION	Changes the collation sequence for the current session.
SET SESSION CONSTRAINT	Overrides the default constraints assigned to a user for the current session.
SET SESSION DATABASE	Changes the default database for the session.
SET SESSION DATEFORM	Changes the default DATE format in field mode and the default format for importing and exporting DATE values for the session.
SET SESSION DEBUG FUNCTION	Identifies the UDF or stored procedure to be run in debug mode the next time the UDF is invoked.
SET SESSION DOT NOTATION	Sets the session response for dot notation query results that return a list of values.
SET SESSION FOR ISOLATED LOADING	Enables or disables isolated loading for DML operations for the session.
SET SESSION FUNCTION TRACE	Enables function trace output for debugging external user-defined functions and external SQL procedures.
SET SESSION JSON IGNORE ERRORS	Enables or disables the validation of JSON data on INSERT operations.
SET SESSION SEARCHUIFDBPATH	Sets the database search path for the SCRIPT execution in the SessionTbl.SearchUIFDBPath column.
SET SESSION TRANSACTION ISOLATION LEVEL	Changes the default transaction isolation level read-only semantics for the current session.
SET TIME ZONE	Changes the default time zone displacement for a session.
SET TRANSFORM GROUP FOR TYPE	Sets the active transform group for Complex Data Types (CDTs) that have multiple transform groups.

Statement	Purpose
SHOW FUNCTION MAPPING	Displays the SQL data definition text for the function mapping.
SHOW MAP	Returns the equivalent CREATE MAP statement in SQL or, optionally, XML.
SHOW object	For tables, macros and views, displays the SQL data definition text for the original create text from DBC.TVM.RequestText.
SHOW QUERY LOGGING	Returns the query logging rule set applied to the specified user, database, user:account set, application set, or all users from the rules cache or from DBC.DBQLRuleTbl.
SHOW request	Displays the DDL for all database objects referenced by a specified DML request.
SHOW STATISTICS	Reports the SQL text for the Optimizer and QCD forms of COLLECT STATISTICS requests that collected the statistics and optionally reports the detailed or summary statistics.
SHOW TABLE	Displays the most recent SQL create text.

Table Statements

CREATE TABLE and CREATE TABLE AS

Defines the column names, column data types and attributes, primary and secondary indexes, column- and table-constraints, partitioning, and other attributes of a new table. The CREATE TABLE AS form copies column definitions to a new table. Optionally, data and statistics are copied to the new table.

The table structure definition is stored in the data dictionary for all table types, except volatile tables. While processing a CREATE TABLE statement, an EXCLUSIVE lock is placed on the table being created.

For information about global temporary trace tables, see [CREATE GLOBAL TEMPORARY TRACE TABLE](#).

To create queue tables, see [CREATE TABLE \(Queue Table Form\)](#).

To log batch insert and update errors, you must create an error table for each data table for which you want to track errors. See [CREATE ERROR TABLE](#).

For information about temporal tables and temporal syntax, see *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

For information on creating time series tables with a Primary Time Index (PTI), see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

ANSI Compliance

CREATE TABLE is ANSI SQL:2011-compliant with extensions.

Global temporary tables are ANSI SQL:2011-compliant.

Other SQL dialects support similar non-ANSI standard statements with names such as the following.

- DECLARE GLOBAL TEMPORARY TABLE
- CREATE TEMP TABLE

Volatile tables are a Teradata extension to the ANSI SQL:2011 standard.

Other SQL dialects support similar non-ANSI standard statements with names such as the following:

- DECLARE GLOBAL TEMPORARY TABLE

Required Privileges

No privileges are required to create, access, modify, or drop volatile tables. For all other table types, you must have the CREATE TABLE privilege on the database or user in which the table is created.

If you specify the JOURNAL option, then you must also have INSERT privilege on the journal table.

To access a table that contains UDT columns, you must have at least one of the following privileges:

- UDTUSAGE on the specified UDT

- UDTUSAGE on the SYSUDTLIB database
- UDTTYPE on the SYSUDTLIB database
- UDTMETHOD on the SYSUDTLIB database

You must have the CONSTRAINT ASSIGNMENT privilege to create a table that includes one or more row-level security columns.

Privileges Granted Automatically

No privileges are granted on newly created volatile tables, because no privileges are needed to access the table.

For other table types, the creator receives all of the following privileges WITH GRANT OPTION on the newly created table:

- CREATE TRIGGER
- DELETE
- DROP TABLE
- DROP TRIGGER
- DUMP
- INDEX
- INSERT
- REFERENCES
- RESTORE
- SELECT
- STATISTICS
- UPDATE

CREATE TABLE Syntax

```
CREATE table_kind TABLE table_specification
  [ , table_option [,...] ] ( column_partition_definition )
  [ index [,...] ]
  [ table_preservation ][;]
```

table_kind

```
[ SET | MULTISSET ] [ GLOBAL TEMPORARY | VOLATILE ]
```

table_specification

```
[ database_name. | user_name. ] table_name
```

table_option

```
{ MAP = map_name [ COLOCATE USING colocation_name |
  [NO] FALLBACK [PROTECTION] |
  WITH JOURNAL TABLE = table_specification |
  [NO] LOG |
  [ NO | DUAL ] [BEFORE] JOURNAL |
  [ NO | DUAL | LOCAL | NOT LOCAL ] AFTER JOURNAL |
  CHECKSUM = { DEFAULT | ON | OFF } |
  FREESPACE = integer [PERCENT] |
  mergeblockratio |
  datablocksize |
  blockcompression |
  isolated_loading
}
```

column_partition_definition

```
column_name data_type [ column_data_type_attribute [,...] ] |

[ COLUMN | ROW ] ( column_name data_type [column_attributes] [,...] )
  [ [NO] AUTO COMPRESS] |

PERIOD FOR period_name ( period_begin_column , period_end_column ) |

normalize_option |

table_constraint
][,...]
```

index

```
[UNIQUE] PRIMARY INDEX [index_name] ( index_column_name [,...] ) |
NO PRIMARY INDEX |
PRIMARY AMP [INDEX] [index_name] ( index_column_name [,...] ) |
PARTITION BY { partitioning_level | ( partitioning_level [,...] ) } |
```

```

UNIQUE INDEX [ index_name ] [ ( index_column_name [,...] ) ]
[loading] |
INDEX [index_name] [ALL] ( index_column_name [,...] )
[ordering] [loading]

```

table_preservation

```

ON COMMIT { DELETE | PRESERVE } ROWS

```

mergeblockratio

```

{ DEFAULT MERGEBLOCKRATIO |
  MERGEBLOCKRATIO = integer [PERCENT] |
  NO MERGEBLOCKRATIO
}

```

datablocksize

```

DATABLOCKSIZE = {
  data_block_size [ BYTES | KBYTES | KILOBYTES ] |
  { MINIMUM | MAXIMUM | DEFAULT } DATABLOCKSIZE
}

```

blockcompression

```

BLOCKCOMPRESSION = { AUTOTEMP | MANUAL | ALWAYS | NEVER | DEFAULT }
[, BLOCKCOMPRESSIONALGORITHM = { ZLIB | ELZS_H | DEFAULT } ]
[, BLOCKCOMPRESSIONLEVEL = { value | DEFAULT } ]

```

isolated_loading

```

WITH [NO] [CONCURRENT] ISOLATED LOADING [ FOR { ALL | INSERT
| NONE } ]

```

column_data_type_attribute

```

{ { UPPERCASE | UC } |
  [NOT] { CASESPECIFIC | CS } |
  FORMAT quotestring |

```



```

TITLE quotestring |
NAMED name |
DEFAULT { number | USER | DATE | TIME | NULL } |
WITH DEFAULT |
CHARACTER SET server_character_set |
[NOT] NULL |
[NOT] AUTO COLUMN |
compression_attribute |
column_constraint_attribute |
identity_column
}

```

normalize_option

```

NORMALIZE [ ALL BUT (normalize_ignore_column_name [,...]) ]
ON normalize_column
[ ON { MEETS OR OVERLAPS | OVERLAPS [OR MEETS]} ]

```

table_constraint

```

CONSTRAINT constraint_name
{ { UNIQUE | PRIMARY KEY } (column_name [,...]) |
  CHECK (boolean_condition) |
  FOREIGN KEY (referencing_column [,...]) references }

```

partitioning_level

```

{ partitioning_expression |
  COLUMN [ [NO] AUTO COMPRESS |
  COLUMN [ [NO] AUTO COMPRESS ] [ ALL BUT ] column_partition ]
} [ ADD constant ]

```

loading

```

WITH [NO] LOAD IDENTITY

```

ordering

```

ORDER BY [ VALUES | HASH ] [ ( order_column_name ) ]

```

compression_attribute

```

{ NO COMPRESS |

  COMPRESS [ constant | ( { constant | NULL } [,...] ) ] |

  COMPRESS USING compress_UDF_name DECOMPRESS
    USING decompress_UDF_name
}

```

column_constraint_attribute

```

[ CONSTRAINT constraint_name ]
{ UNIQUE | PRIMARY KEY | CHECK ( boolean_condition ) | references } |
[ row_level_security_constraint_column_name [,...] ] CONSTRAINT

```

identity_column

```

GENERATE {ALWAYS | BY DEFAULT} AS IDENTITY
[ ( START WITH constant |
  INCREMENT BY constant |
  MINVALUE constant |
  NO MINVALUE |
  MAXVALUE constant |
  NO MAXVALUE |
  [ NO ] CYCLE ) ]

```

references

```

REFERENCES [ WITH [NO] CHECK OPTION ] referenced_table_name
[ ( referenced_column_name [,...] ) ]

```

column_partition

```

( [ COLUMN | ROW ] { column_name | ( column_name [,...] ) }
  [[NO] AUTO COMPRESS]
) [,...]

```

CREATE TABLE AS Syntax

```
{ CREATE table_kind TABLE | CT } table_specification
  [ table_option [,...] ]
  ( attribute [,...] )
  AS_clause
  [ , index [[,]...] ]
  [ table_preservation ][;]
```

table_kind

```
[ SET | MULTISSET ] [ GLOBAL TEMPORARY | VOLATILE ]
```

table_specification

```
[ database_name. | user_name. ] table_name
```

table_option

```
{ MAP = map_name [COLOCATE USING colocation_name |
  [NO] FALLBACK [PROTECTION] |
  WITH JOURNAL TABLE = table_specification |
  [NO] LOG |
  [ NO | DUAL ] [BEFORE] JOURNAL |
  [ NO | DUAL | LOCAL | NOT LOCAL ] AFTER JOURNAL |
  CHECKSUM = { DEFAULT | ON | OFF } |
  FREESPACE = integer [PERCENT] |
  mergeblockratio |
  datablocksize |
  blockcompression |
  isolated_loading
}
```

attribute

```
{ column_specification |
  [ COLUMN | ROW ] ( column_specification [,...] ) [ [NO]
  AUTOCOMPRESS ] |
```

```

    table_constraint
}

```

AS_clause

```

AS source_table [ subquery_clause ] WITH [NO] DATA [ AND
[NO] STATISTICS ]

```

index

```

[UNIQUE] PRIMARY INDEX [index_name] ( index_column_name [,...] ) |
NO PRIMARY INDEX |
PRIMARY AMP [INDEX] [index_name] ( index_column_name [,...] ) |
PARTITION BY { partitioning_level | ( partitioning_level [,...] ) } |
UNIQUE INDEX [ index_name ] [ ( index_column_name [,...] ) ]
[loading] |
INDEX [index_name] [ALL] ( index_column_name [,...] )
[ordering] [loading]
[,...]

```

table_preservation

```

ON COMMIT { DELETE | PRESERVE } ROWS

```

mergeblockratio

```

{ DEFAULT MERGEBLOCKRATIO |
  MERGEBLOCKRATIO = integer [PERCENT] |
  NO MERGEBLOCKRATIO
}

```

datablocksize

```

DATABLOCKSIZE = {
  data_block_size [ BYTES | KBYTES | KILOBYTES ] |
  { MINIMUM | MAXIMUM | DEFAULT } DATABLOCKSIZE
}

```

blockcompression

```
BLOCKCOMPRESSION = { AUTOTEMP | MANUAL | ALWAYS | NEVER | DEFAULT }
[ , BLOCKCOMPRESSIONALGORITHM = { ZLIB | ELZS_H | DEFAULT } ]
[ , BLOCKCOMPRESSIONLEVEL = { value | DEFAULT } ]
```

isolated_loading

```
WITH [NO] [CONCURRENT] ISOLATED LOADING [ FOR { ALL | INSERT
| NONE } ]
```

subquery_clause

```
AS source_table WITH [NO] DATA [ AND [NO] STATISTICS ]
```

column_specification

```
column_name [ column_data_type_attribute [...] ]
```

partitioning_level

```
{ partitioning_expression |
  COLUMN [ [NO] AUTO COMPRESS |
  COLUMN [ [NO] AUTO COMPRESS ] [ ALL BUT ] column_partition ]
} [ ADD constant ]
```

loading

```
WITH [NO] LOAD IDENTITY
```

ordering

```
ORDER BY [ VALUES | HASH ] [ ( order_column_name ) ]
```

column_data_type_attribute

```
{ { UPPERCASE | UC } |
  [NOT] { CASESPECIFIC | CS } |
  FORMAT quotestring |
```

```

TITLE quotestring |
NAMED name |
DEFAULT { number | USER | DATE | TIME | NULL } |
WITH DEFAULT |
CHARACTER SET server_character_set |
[NOT] NULL |
[NOT] AUTO COLUMN |
compression_attribute |
column_constraint_attribute |
identity_column
}

```

compression_attribute

```

{ NO COMPRESS |

COMPRESS [ constant | ( { constant | NULL } [,...] ) ] |

COMPRESS USING compress_UDF_name DECOMPRESS
  USING decompress_UDF_name
}

```

column_constraint_attribute

```

[ CONSTRAINT constraint_name ]
{ UNIQUE | PRIMARY KEY | CHECK ( boolean_condition ) | references } |
[ row_level_security_constraint_column_name [,...] ] CONSTRAINT

```

identity_column

```

GENERATE {ALWAYS | BY DEFAULT} AS IDENTITY
[ ( START WITH constant |
  INCREMENT BY constant |
  MINVALUE constant |
  NO MINVALUE |
  MAXVALUE constant |
  NO MAXVALUE |
  [ NO ] CYCLE ) ]

```

Syntax Elements

table_kind

The kind of table determines duplicate row control. See *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for details. The table can be created as a global temporary table or a volatile table. If you do not specify global temporary or volatile, then the table is defined as a persistent user data table, also referred to as base tables. Hash and join index tables are also considered base tables.

If you do not explicitly specify SET or MULTiset, the table kind assignment depends on the session mode:

Session Mode	Default
ANSI	MULTiset
Teradata	SET

The session mode default is in effect, except when you:

- Copy a table definition using the non-subquery form of the CREATE TABLE .. AS syntax. The default table kind is the table kind of the source table, regardless of the current session mode.
- Create a column-partitioned table. The default table kind is always MULTiset, regardless of the session mode or the setting of the DBS Control parameter PrimaryIndexDefault.

MULTiset

Duplicate rows are permitted, in compliance with the ANSI SQL:2011 standard. If there are uniqueness constraints on any column or set of columns in the table definition, then the table cannot have duplicate rows even if it is declared as MULTiset. Vantage creates NoPI and column-partitioned tables as MULTiset tables by default.

Some client utilities have restrictions regarding MULTiset tables. See the appropriate documentation:

- *Teradata® FastLoad Reference*, B035-2411
- *Teradata® Parallel Data Pump Reference*, B035-3021

SET

Duplicate rows are not permitted. You cannot create the following kinds of tables as SET tables:

- Temporal
- Column-partitioned
- NoPI

GLOBAL TEMPORARY

A temporary table definition is created and stored in the data dictionary for future materialization. You can create global temporary tables by copying a table WITH NO DATA, but not by copying a table WITH DATA.

You cannot create a column-partitioned global temporary table.

You cannot create a global temporary table with row-level security constraint columns.

VOLATILE

Create a volatile table. The definition of a volatile table is retained in memory only for the duration of the session in which it is defined. Space usage is charged to the login user spool space. Because volatile tables are private to the session that creates them, the system does not check the creation, access, modification, and drop privileges. A single session can materialize up to 1,000 volatile tables.

The contents and the definition of a volatile table are dropped when a system reset occurs.

If you frequently reuse particular volatile table definitions, consider writing a macro that contains the CREATE TABLE text for those volatile tables.

You cannot create a column-partitioned volatile table or normalized volatile table.

You cannot create secondary, hash, or join indexes on a volatile table.

You cannot create a volatile table with row-level security constraint columns.

For more information about volatile tables, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

*table_specification***database_name**

The name of the database in which it is to be contained if different from the current default database.

If the name is not fully qualified, then the system assigns the name of the default database for the current session.

user_name

The name of the user in which it is to be contained if different from the current default database.

If the name is not fully qualified, then the system assigns the name of the default database for the current session.

table_name

The name of the new table.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

table_option**MAP**

You can specify an existing contiguous or sparse map for the table.

You must have been granted the specified map, except when you specify the same map determined to be the default map.

Specifying a map is optional. If you do not specify a map, the default map is determined according to the following order of precedence:

- If the immediate owner is not the creator:
 - Default map, if defined, for the profile of the immediate owner.
 - Default map, if defined, for the immediate owner.
 - System-default map.
- Default map, if defined, for the profile of the creator.
- Default map, if defined, for the creator.
- System-default map.

See GRANT MAP in *Teradata Vantage™ - SQL Data Control Language*, B035-1149. Also, see the CREATE PROFILE [DEFAULT MAP](#) option and the CREATE USER [DEFAULT MAP](#) option.

map_name

Name of an existing contiguous or sparse map.

You cannot specify TD_DataDictionaryMap or TD_GlobalMap.

COLOCATE USING *colocation_name*

Optionally, you can specify a colocation name so that the tables reside on the same AMPs to avoid a redistribution of the rows when the tables are joined on a primary index or primary AMP index. For example, you can colocate two tables, then join the tables on the primary index or primary AMP index columns.

You can only specify this option for a sparse map. For a contiguous map, the *colocation_name* is not needed for colocation and is set to NULL.

If you do not specify a colocation name, the name defaults to *database_table*, where *database* is the name of the database followed by an underscore (_) and *table* is the name

of the table. If *database* exceeds 63 characters, *database* is truncated to 63 characters. If *table* exceeds 64 characters, *table* is truncated to 64 characters.

For a CREATE TABLE ... AS *source_table* statement, where the map for the created table defaults to the sparse map of the source table, *colocation_name* defaults to the colocation name of the source table.

Usage Notes

Default Map for the User, Database, or Profile

The immediate owner of the table can be a user or a database. See the CREATE USER [DEFAULT MAP](#) option, the MODIFY USER [DEFAULT MAP](#) option, the CREATE DATABASE [DEFAULT MAP](#) option, the MODIFY DATABASE [DEFAULT MAP](#) option, the CREATE PROFILE [DEFAULT MAP](#) option, or the MODIFY PROFILE [DEFAULT MAP](#) option.

User, Database, or Profile DEFAULT MAP OVERRIDE ON ERROR Option

If the map you specify is not valid for any reason, (such as it does not exist, has not been granted to you, or is not in the same secure zone), Vantage either substitutes a default map or returns an error, subject to the values of the DEFAULT MAP settings for your PROFILE, USER, or DATABASE.

Default Map for a CREATE TABLE Statement

If you do not specify the MAP option when creating a table, the system determines a default map for the table in following order of precedence:

- If the immediate owner is not the creator:
 - Default map, if defined, for the profile of the immediate owner.
 - Default map, if defined, for the immediate owner.
 - System-default map.
- Default map, if defined, for the profile of the creator.
- Default map, if defined, for the creator.
- System-default map.

Default Map for a CREATE TABLE AS Statement

For a CREATE TABLE AS *source_table* statement, the map for the source table is used as the default map if you have been granted this map, or the map is one of the following as listed in order of precedence:

- If the immediate owner is not the creator:
 - Default map, if defined, for the profile of the immediate owner.
 - Default map, if defined, for the immediate owner.
 - System-default map.

- Default map, if defined, for the profile of the creator.
- Default map, if defined, for the creator.
- System-default map.

For a CREATE TABLE AS *query* statement, regardless of whether the query references 0, 1, 2, or more tables, the system determines a default map for table in the following order of precedence:

- If the immediate owner is not the creator:
 - Default map, if defined, for the profile of the immediate owner.
 - Default map, if defined, for the immediate owner.
 - System-default map.
- Default map, if defined, for the profile of the creator.
- Default map, if defined, for the creator.
- System-default map.

Secure Zones and Sparse Maps

For a sparse map, you must be in the same secure zone as the sparse map.

AMPs and Sparse Maps

For sparse maps, the system determines on which AMPs of the parent contiguous map the table resides based on the number of AMPs defined for the sparse map and the colocation name for the table. See [CREATE MAP](#).

Renaming a Table and Colocation Name

Renaming a table does not change the colocation name for the table.

Error Tables and Sparse Maps

Error tables use the same sparse map and the same colocation name as the associated data table.

Map for Global Temporary Tables and Volatile Tables

You can specify an existing contiguous or sparse map for global temporary tables and volatile tables.

Volatile tables in DBC are always in TD_DataDictionaryMap.

Permanent Journaling and Maps

If permanent journaling is enabled for a table, the map for the table must be a contiguous map and the map must be the same as the contiguous map of the journal table for the table. A journal table must have a contiguous map.

Examples

Specifying a Sparse Map and Colocation Name for a Table

In this example, the table creator has been granted the sparse maps FourAMP_Map, with AMPCount = 4, and TenAMP_Map, with AMPCount = 10. The colocation name defaults to MyDatabase_Table1.

```
CREATE TABLE MyDatabase.Table1, MAP=FourAMP_Map
(a1 INTEGER, b1 INTEGER) UNIQUE PRIMARY INDEX (a1);
```

In the example below, the colocation name is explicitly specified. Even though this table has the same colocation name as the previous table, the tables are not colocated because each table definition specifies a different sparse map. To avoid confusion, you should specify a different colocation name for tables that use different sparse maps.

```
CREATE TABLE AnotherTable,
Map=TenAMP_Map COLOCATE USING MyDatabase_Table1
(a3 INTEGER, d3 INTEGER) PRIMARY INDEX (a3);
```

Specifying a Colocation Name for Tables

The table creator has been granted the sparse map, TenAMP_Map, with AMPCOUNT=10. The colocation name is MyGroups_Group4.

```
CREATE TABLE MyDatabase1.Orders
, MAP=TenAMP_Map COLOCATE USING MyGroups_Group4
(a1 INTEGER, b1 INTEGER) UNIQUE PRIMARY INDEX (a1);
```

The colocation name is also MyGroups_Group4 for this table. Since the above two tables have the same map, colocation name, and primary index data type, they can be directly joined on their primary indexes without redistribution.

```
CREATE TABLE MyDatabase2.LineItems
, Map=TenAMP_Map COLOCATE USING MyGroups_Group4
(a2 INTEGER, c2 INTEGER) PRIMARY INDEX (a2);
```

FALLBACK

Duplicate copy protection for the table.

When you specify FALLBACK, Vantage creates and stores duplicate copies of rows in the table.

The default for this option is set by a CREATE DATABASE, CREATE USER, MODIFY DATABASE, or MODIFY USER request for the database in which the table is to be created.

When a hardware read error occurs, the file system reads the fallback copy of the data and reconstructs the rows in memory on their home AMP. Support for Read From Fallback is limited to the following cases:

- Requests that do not attempt to modify data in the bad data block
- Primary subtable data blocks
- Reading the fallback data in place of the primary data. In some cases, Active Fallback can repair the damage to the primary data dynamically. In situations where the bad data block cannot be repaired, Read From Fallback substitutes an error-free fallback copy of the corrupt rows each time the read error occurs.

NO

Duplicate copy protection is not provided for the table.

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

PROTECTION

Optional keyword that can be specified after the FALLBACK keyword.

WITH JOURNAL TABLE

This clause is required if the CREATE TABLE statement specifies some level of journaling, but a default journal table was not defined for the database in which the new table is being created.

If you specify permanent journaling for this table, but do not specify a journal table with this clause, then you must define a default permanent journal table for the containing database.

If a default permanent journal table was defined for the database, then this clause can be used to override the default.

You cannot add or modify journal options for a NoPI table or for a column-partitioned table.

Journal options are not supported for tables with row sizes greater than 64KB.

table_name

The name of the permanent journal table to be used for the data table being created.

table_name can be contained within the same database as the table being created or in a different database.

database_name

If you specify a database name, then that name must exist and *table_name* must have been defined as its default permanent journal name.

If you do not specify a database name, then the default database for the current session is assumed and *table_name* must have been defined as the default permanent journal table.

user_name

If you specify a user name, then that name must exist and *table_name* must have been defined as its default permanent journal name.

If you do not specify a user name, then the default user for the current session is assumed and *table_name* must have been defined as the default permanent journal table.

LOG

Transient journaling is performed for global temporary and volatile tables. This option only pertains to global temporary and volatile tables.

Update, insert, or delete operations on the global temporary or volatile table are logged in the transient journal. This is the default.

NO

Transient journal logging of rows is not performed, reducing the system overhead of logging.

If an error or restart occurs and the table is defined as NO LOG, then any update, insert, or delete operations on the global temporary or volatile table cannot be recovered.

If the table is defined as NO LOG, a transient journal is generated for the transaction and the content of any materialized global temporary table or volatile table is emptied when a transaction aborts.

BEFORE JOURNAL

The number of before change images to be maintained.

If the JOURNAL keyword is specified without NO or DUAL, then a single copy of the image is maintained unless FALLBACK is in effect or is also specified.

If journaling is requested for a table that uses fallback protection, DUAL images are maintained automatically.

Permanent journaling is not supported for NoPI tables, column-partitioned tables, global temporary tables, or volatile tables.

DUAL

Dual images are maintained.

NO

A journal image is not maintained.

AFTER JOURNAL

Type of after-image to be maintained for the table. Any existing images are not affected until the table is updated.

Permanent journaling is not supported for NoPI tables, column-partitioned tables, global temporary tables, or volatile tables.

NO

After-change images are not maintained for the table.

DUAL

Two after-change images are maintained for the table.

LOCAL

Single after-image journal rows for non-fallback data tables are written on the same virtual AMP as the changed data rows.

NOT LOCAL

Single after-image journal rows for non-fallback data tables are written on another virtual AMP in the cluster.

JOURNAL

Type of image to be maintained for the table.

This option can appear twice in the same request: once to specify a BEFORE or AFTER image, and again to specify the alternate type.

Permanent journaling is not supported for NoPI tables, column-partitioned tables, global temporary tables, or volatile tables.

The default for this option is established by a CREATE DATABASE, CREATE USER, or MODIFY USER request for the database in which the table is to be created.

You cannot add or modify journal options for a NoPI or column-partitioned table.

Journal options are not supported for tables with row sizes greater than 64KB.

BEFORE JOURNAL**AFTER JOURNAL**

If you specify either BEFORE or AFTER, Vantage maintains only the default journal image for that type of journal.

For example, if you specify AFTER, before-image journaling remains at the default setting.

If you specify both BEFORE and AFTER, Vantage maintains both journal images, but the two specifications must not conflict with one another.

If you do not specify BEFORE or AFTER, Vantage maintains both journal images.

CHECKSUM

A table-specific disk I/O integrity checksum for detection of hardware read errors. The checksum level setting applies to primary data rows, fallback data rows, and all secondary index rows for the table.

For a system-defined join index, the integrity checking setting you specify for a base table also applies to any system-defined join indexes defined on that table. You cannot specify integrity checking for a system-defined join index directly.

If you do not specify a value, the system assumes the system-wide default value for this table type. This is equivalent to specifying DEFAULT.

ON

Calculate checksums using the entire disk block. Sample 100% of the disk blocks to generate a checksum.

OFF

Disables checksum disk I/O integrity checks.

DEFAULT

The default setting is the current DBS Control checksum setting specified for this table type.

FREESPACE

The percentage of free space to reserve on a cylinder during some loading operations.

The following operations use the FREESPACE setting you specify:

- FastLoad data loads.
- MultiLoad data loads into empty tables.
- Table Rebuild.
- System reconfiguration

- Ferret PACKDISK.
- MiniCylPack.

If few cylinders are available and storage space is limited, MiniCylPack might not be able to honor the specified FREESPACE value.

- ALTER TABLE request that adds fallback protection to a table.
- CREATE INDEX request that defines or redefines a secondary index on a populated table.
- Creation of a fallback table during an INSERT ... SELECT operation into an empty table that is defined with fallback.
- Creation of a secondary index during an INSERT ... SELECT operation into an empty table that is defined with a secondary index.

The following operations do not use the FREESPACE setting you specify:

- SQL INSERT operations.
- Teradata Parallel Data Pump INSERT operations.
- MultiLoad data loads into populated tables.

NOTICE

You cannot use PACKDISK to change this attribute after you have created a table. Instead, submit an ALTER TABLE request to change the free space percent value for the table and then immediately afterward submit a PACKDISK command using the same free space percent value. See [ALTER TABLE](#). For more information and instructions for running PACKDISK, see the description of the Ferret utility in *Teradata Vantage™ - Database Utilities*, B035-1102.

integer

Value from 0 through 75, representing a percentage. DECIMAL or FLOATING POINT constants are converted to an integer by truncating the value.

PERCENT

An optional keyword you can type following the value for *integer*.

mergeblockratio

The merge block ratio to be used for this table when Vantage combines smaller data blocks into a single larger data block.

You can only specify a numeric merge block ratio for permanent base tables and permanent journal tables. You cannot specify a numeric merge block ratio for global temporary or volatile tables. You can specify either the DEFAULT MERGEBLOCKRATIO or the NO MERGEBLOCKRATIO options for global temporary and volatile tables.

If you do not specify this option, Vantage uses the value that is specified for the MergeBlockRatio parameter in the DBS Control record at the time a data block merge operation on the table begins.

This value of the merge block ratio for a table does not affect the resulting block size when only a single block is modified. Setting the merge block ratio to too high a value can cause the resulting merged block to require being split during subsequent modifications.

The system uses the merge block ratio you specify, depending on the setting for the DBS Control parameter `DisableMergeBlocks`. See *Teradata Vantage™ - Database Utilities*, B035-1102.

DisableMergeBlocks	Description
FALSE	The value you specify for MERGEBLOCKRATIO overrides the system-wide default setting for the DBS Control parameter MergeBlockRatio.
TRUE	The system ignores all table-level settings for the merge block ratio and does not merge data blocks for any table in the system.

DEFAULT MERGEBLOCKRATIO

Vantage uses the value for MergeBlockRatio that is defined by the DBS Control record at the time a data block merge operation on this table begins.

MERGEBLOCKRATIO = *integer*

Vantage uses the value specified by *integer* as the merge block ratio when a data block merge operation occurs on this table.

The valid range for *integer* is from 1 through 100. The default value is 60.

PERCENT

Optional keyword to indicate that the value is a percentage.

NO MERGEBLOCKRATIO

NO MERGEBLOCKRATIO means that Vantage does not merge small data blocks for this table.

Example: Specifying Various MERGEBLOCKRATIO Settings

These examples demonstrate different specifications for the MERGEBLOCKRATIO option. If the DBS Control parameter `DisableMergeBlocks` is set to TRUE, Vantage ignores the setting for this option globally and disables all data block merges for all tables in the system. See *Teradata Vantage™ - Database Utilities*, B035-1102.

This example defines `emp_table` so that the system uses the value for the DBS Control parameter `MergeBlockRatio` as the merge block ratio when merging data blocks for the table.

```
CREATE TABLE emp_table, DEFAULT MERGEBLOCKRATIO (
  emp_no INTEGER);
```

This example sets the value of the merge block ratio for emp_table to 25%.

```
CREATE TABLE emp_table, MERGEBLOCKRATIO=25 PERCENT (
  emp_no INTEGER);
```

This example defines emp_table in a way that disables all data block merges.

```
CREATE TABLE emp_table, NO MERGEBLOCKRATIO (
  emp_no INTEGER);
```

data_block_size

The maximum data block size for blocks that contain multiple rows.

data_block_size

You can express the value either as a decimal or integer number or using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

This specification is optional. If you do not specify DATABLOCKSIZE, then data blocks default to the sizes set in the PermDBSize and JournalDBSize fields of the DBS Control record. For more information about DBS Control settings, see *Teradata Vantage™ - Database Utilities*, B035-1102.

For systems without large cylinders, the minimum data block size you can specify is 9,216 bytes (18 sectors). You can specify a value of 8,960 bytes (17.5 sectors), but Vantage rounds that value up to 9,216 bytes internally, and that is the actual data block size the system uses.

For systems with large cylinders, the minimum data block size you can specify is 21,504 bytes (42 sectors). You can specify a value of 21,248 bytes (41.5 sectors), but Vantage rounds that value up to 21,504 bytes internally, and that is the value the system actually uses.

The default values for data block types are determined by DBS Control settings. For detailed information about setting default data block sizes, see *Teradata Vantage™ - Database Utilities*, B035-1102.

- For a permanent data block, the default size is determined by the PermDBSize parameter.
- For transient and permanent journals, the default data block size is determined by the JournalDBSize parameter.

BYTES

For BYTES, the value of the data block size is the result of rounding *n* to the nearest multiple of the sector size in bytes. This is the default.

Rounding is upward when n is exactly halfway between two consecutive multiples.

KBYTES KILOBYTES

For KILOBYTES, the value of the data block size is the result of rounding $1024n$ to the nearest multiple of the sector size in bytes.

MINIMUM

The smallest data block size for this table.

MINIMUM DATABLOCKSIZE sets the maximum data block size for blocks that contain multiple rows to the minimum value of 21,504 bytes (42 sectors) for systems with large cylinders or 9,216 bytes (18 sectors) for systems without large cylinders.

You can abbreviate MINIMUM as MIN.

For details about minimum data block sizes, see DATABLOCKSIZE for CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

MAXIMUM

The largest data block size for this table.

Systems that do not support block sizes 128KB and larger use a value of 127.5KB.

Systems that allow blocks larger than 127.5KB use a maximum of 1,048,064 bytes (2047 sectors) for large cylinders or 262,144 bytes (512 sectors) for small cylinders.

The value can be expressed either as a decimal number or integer number or using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

You can abbreviate MAXIMUM as MAX.

DEFAULT

The default data block size for this table.

If you specify DEFAULT or do not specify a value, Vantage uses the system-wide default data block size specified by the DBS Control setting PermDBSize. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Example: Specifying DATABLOCKSIZE, CHECKSUM, and FREESPACE

This example includes options to specify a maximum data block size of 16,384 bytes (32 sectors), a disk I/O checksum of OFF, and 10 percent free space:

```
CREATE TABLE employee, DATABLOCKSIZE = 16384 BYTES, CHECKSUM = OFF,
FREESPACE = 10 PERCENT (
```

```

empno    SMALLINT FORMAT '9(5)'
         CHECK (empno >= 10001 AND empno <= 32001) NOT NULL,
name     VARCHAR(12) NOT NULL,
deptno   SMALLINT FORMAT '999'
         CHECK (deptno >= 100 AND deptno <= 900),
jobtitle VARCHAR(12),
salary   DECIMAL(8,2) FORMAT 'ZZZ,ZZ9.99'
         CHECK (salary >= 1.00 AND salary <= 999000.00),
yrsexp   BYTEINT FORMAT 'Z9'
         CHECK (yrsexp >= -99 AND yrsexp <=99),
dob      DATE FORMAT 'MMbDDbYYYY' NOT NULL,
sex      CHARACTER UPPERCASE NOT NULL,
race     CHARACTER UPPERCASE,
mstat    CHARACTER UPPERCASE,
edlev    BYTEINT FORMAT 'Z9'
         CHECK(edlev >=0 AND edlev <= 22) NOT NULL,
hcap     BYTEINT FORMAT 'Z9'
         CHECK (hcap >= -99 AND hcap <= 99)
UNIQUE PRIMARY INDEX (empno),
INDEX (name);

```

blockcompression

Table data is compressed at the block level.

You cannot specify this option to modify the definitions of global temporary tables or volatile tables.

For details, see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184. For information about DBS Control fields, see *Teradata Vantage™ - Database Utilities*, B035-1102.

AUTOTEMP

Block-level compression is applied based on the temperature of the cylinders on which the data is stored. The file system determines the block-level compression setting for the table based on its Teradata Virtual Storage temperature. The definitions of the various thresholds are determined by the DBS Control field TempBLCThresh. You can adjust the temperature values for data being loaded using SET QUERY_BAND. See [Example: Setting BLOCKCOMPRESSION and TVSTEMPERATURE Query Bands](#).

MANUAL

Block-level compression is applied based on the default for the table at the time the table is created. The defaults for the table are determined by the settings in the Compression group of DBS Control fields. You can override these values using SET QUERY_BAND. See [Example: Using the BLOCKCOMPRESSION Reserved Query Band](#).

Tables can be compressed or uncompressed at any time after loading by using the Ferret COMPRESS and UNCOMPRESS commands.

ALWAYS

The table and its subtables are always block-level compressed, even if a query band or the applicable DBS Control block-level compression settings indicate otherwise. The DBS Control field BlockLevelCompression must be enabled.

NEVER

The table and its subtables are not block-level compressed, even if a query band or the applicable DBS Control block-level compression settings indicate otherwise.

DEFAULT

The table uses the block-level compression setting in the DBS Control field DefaultTableMode.

BLOCKCOMPRESSIONALGORITHM

Specifies the algorithm to use for block-level compression (BLC).

This option only applies when the effective BLOCKCOMPRESSION is MANUAL, AUTOTEMP, or ALWAYS.

ZLIB

Block-level compression using the zlib software algorithm.

ELZS_H

Block-level compression using a hardware compression engine board in every system node.

DEFAULT

Uses the setting of the DBS Control field CompressionAlgorithm.

BLOCKCOMPRESSIONLEVEL

Specifies a value to indicate a preference for compression speed or compression effectiveness. This option only applies when the BLOCKCOMPRESSIONALGORITHM option is set to ZLIB. Although this option is accepted in combination with BLOCKCOMPRESSIONALGORITHM = ELZS_H, this option is ignored. If BLOCKCOMPRESSIONALGORITHM is altered to ZLIB, this option takes effect.

value

Integer from 1 through 9, where 1 specifies the least processor-intensive compression speed with the lowest compression ratio and 9 specifies the most processor-intensive compression speed with highest compression ratio.

DEFAULT

The table uses the compression level setting of the DBS Control field CompressionLevel.

Example: Specifying Block-Level Compression Set to ALWAYS

This example creates a table that is always block-level compressed using the zlib compression algorithm with a compression level of 8.

```
CREATE SET TABLE t_blc, NO FALLBACK, NO BEFORE JOURNAL,
                    NO AFTER JOURNAL, CHECKSUM = DEFAULT,
                    BLOCKCOMPRESSION = ALWAYS
                    BLOCKCOMPRESSIONALGORITHM=ZLIB
                    BLOCKCOMPRESSIONLEVEL=8 (
    c1    INTEGER FORMAT '-(10)9' )
UNIQUE PRIMARY INDEX (c1)
PARTITION BY(RANGE_N(c1 BETWEEN 1
                    AND    500
                    EACH 5));
```

Example: Specifying Block-Level Compression Set to AUTOTEMP

This example creates a table with its BLOCKCOMPRESSION option set to AUTOTEMP, so that Vantage can change the compressed state of the data in the table at any time based on its temperature.

```
CREATE SET TABLE t_blc, NO FALLBACK, NO BEFORE JOURNAL,
                    NO AFTER JOURNAL, CHECKSUM = DEFAULT,
                    BLOCKCOMPRESSION = AUTOTEMP(
    c1    INTEGER FORMAT '-(10)9' )
UNIQUE PRIMARY INDEX (c1)
PARTITION BY(RANGE_N(c1 BETWEEN 1
                    AND    500
                    EACH 5));
```

isolated_loading

Defines the table for load isolation (LDI). Load isolation enables concurrent read operations on committed rows while the table is being loaded.

The following types of tables cannot be defined as load isolated:

- Volatile table
- Error table
- Queue table
- Temporary table
- Global Temporary table
- Column partitioned table

NO

Defines the table as a non-load isolated table. This is equivalent to creating the table without this clause.

CONCURRENT ISOLATED LOADING

Enables concurrent read operations on committed rows while the table is being modified.

FOR ALL

All modifications can be concurrent load isolated.

FOR INSERT

Only INSERT operations can be concurrent load isolated.

FOR NONE

Concurrent load operations are disabled.

Example: Define an LDI Table

All operations can be concurrent load isolated on this table.

```
CREATE TABLE ldi_table1, WITH CONCURRENT ISOLATED LOADING (
    c1 INTEGER,
    c2 INTEGER,
    c3 INTEGER)
PRIMARY INDEX (c1)
INDEX (c2)
;
```

Example: Define an LDI Table Where Only INSERT Operations are Concurrent Load Isolated

Only INSERT operations can be concurrent load isolated on this table.

```
CREATE TABLE ldi_table3, WITH ISOLATED LOADING FOR INSERT (
    c1 INTEGER,
    c2 INTEGER,
```



```

        c3 INTEGER)
PRIMARY INDEX (c1)
INDEX (c2)
;

```

Example: Define an LDI Table Where All Operations are Nonconcurrent LDI

All operations are nonconcurrent LDI because of the FOR NONE specification.

```

CREATE TABLE ldi_table3, WITH ISOLATED LOADING FOR NONE (
        c1 INTEGER,
        c2 INTEGER,
        c3 INTEGER)
PRIMARY INDEX (c1)
INDEX (c2)
;

```

Example: Define a Non-LDI table

All operations on this table are non-load-isolated. This is equivalent to creating a table without this clause.

```

CREATE TABLE ldi_table3, WITH NO ISOLATED LOADING (
        c1 INTEGER,
        c2 INTEGER,
        c3 INTEGER)
PRIMARY INDEX (c1)
INDEX (c2)
;

```

column_partition_definition

The format to be used for the storage of a column partition. A column partition consists either of a series of containers or a series of subrows.

You can only specify this option for a column-partitioned table.

If you do not specify either COLUMN or ROW, then Vantage determines which format to use based on the width of the column partition value.

As a general rule, Vantage assigns COLUMN format to narrow column-valued partitions and ROW format to wide column-valued partitions.

The system autocompresses data as physical rows that are inserted into a column partition of a column-partitioned table unless there are no applicable autocompression methods that reduce the size of the physical row or you specify NO AUTO COMPRESS.

column_name

The name of one or more columns, in the order in which they and their attributes are to be defined for the table. Up to 2,048 columns can be defined for a table.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

data type

One or more phrases that define the data type for the column. For a list of data types you can specify, see [Data Types Syntax](#).

You must specify a single data type for each *column_name* , except for those defined in a CREATE TABLE AS request, where you cannot specify data types.

You must define the data type before you specify attributes. Column data attribute specifications are optional.

If you do not specify explicit formatting, a column assumes the default format for the data type. The default format can be specified by a custom data formatting specification (SDF) defined by the tdlocaledef utility. See *Teradata Vantage™ - Database Utilities*, B035-1102. Explicit formatting applies to the parsing and to the retrieval of character strings.

For information on data types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

column_data_type_attribute

For information about converting data between data types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

The following column attributes are supported for UDT, Geospatial, and Period columns:

- DEFAULT NULL
- FORMAT
- NAMED
- NOT NULL
- NULL
- TITLE

The following column attributes are not supported for UDT, Geospatial, and Period columns:

- CASESPECIFIC or CS
- NOT CASESPECIFIC or NOT CS
- CHARACTER SET
- COMPRESS
- DEFAULT DATE

- DEFAULT TIME
- DEFAULT USER
- UPPERCASE or UC

For a complete list of the attributes that are supported for Vantage, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

For information about the additional column attributes that apply to the columns of temporal tables, see *Teradata Vantage™ - Temporal Table Support*, B035-1182 and *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186.

UPPERCASE

The column data is in uppercase format.

CASESPECIFIC

The column data is in case-specific format.

NOT

The column data is not in case-specific format.

***FORMAT* quotestring**

The format string that must be valid for the external type of the UDT, the external type being its fromsql transform routine as defined either by default or by user definition using the CREATE TRANSFORM statement. See [CREATE TRANSFORM and REPLACE TRANSFORM](#).

If you do not specify a format, the system automatically applies the default display format of the external type.

***TITLE* quotestring**

A title for reporting purposes.

The maximum size for a *column_name* TITLE is 256 characters.

This option is a Teradata extension to the ANSI standard.

***NAMED* name**

Default name.

DEFAULT***number***

Value to use as a default.

USER

Default user.

DATE

Default date.

TIME

Default time.

NULL

Nullable column as the default.

WITH DEFAULT

All rows initially contain the system default in the field.

CHARACTER SET server_character_set

Character column definitions use the character set assigned as the default for the user with CREATE USER and MODIFY USER. You can explicitly override the default character set definitions by using the CHARACTER SET clause. For example, if the DEFAULT CHARACTER SET defined by CREATE USER is Unicode for a user, whenever that user creates or alters a table, the declaration CHARACTER(*n*) in CREATE TABLE or ALTER TABLE is equivalent to CHARACTER(*n*) CHARACTER SET UNICODE. See the information about the CHARACTER SET phrase in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

You cannot specify a character server data set of KANJI1.

NULL

The default is NULL except when you copy a table definition using the CREATE TABLE AS syntax, in which case the system carries the specification made for the source table over to the target table definition.

NOT

You must specify NOT NULL if you do not want nulls to be valid for the column.

AUTO COLUMN

Specifies that the column accepts shredded data not designated for the other columns in the table.

You can only specify the AUTO COLUMN option for a single JSON data type column in a table.

Examples: Creating a Table with a JSON Auto Column

This example creates a table MyTable with the JSON auto column j.

```
CREATE TABLE MyTable (
  a INTEGER,
  b INTEGER,
  j JSON AUTO COLUMN);
```

This table definition for MyTable includes the JSON auto column j defined with a length of 64K bytes and Latin character set.

```
CREATE TABLE MyTable (
  a INTEGER,
  b INTEGER,
  j JSON(64000) CHARACTER SET LATIN AUTO COLUMN);
```

This table definition for MyTable includes the JSON column c, which is not an auto column, and the JSON auto column j defined with a length of 64K bytes and Latin character set.

```
CREATE TABLE MyTable (
  a INTEGER,
  b INTEGER,
  c JSON,
  j JSON(64000) CHARACTER SET LATIN AUTO COLUMN);
```

compression_attribute

Compression attributes are a Teradata extension to the ANSI SQL:2011 specification.

COMPRESS *constant*

Multivalue compression for a set of distinct values in a column.

Using COMPRESS can save space, depending on the percentage of rows for which the compressed value is assigned.

You can specify multivalue compression for all numeric types, and the following predefined data types:

- BYTE
- VARBYTE
- DATE
- CHARACTER
- VARCHAR
- GRAPHIC
- VARGRAPHIC
- TIME
- TIME WITH TIME ZONE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

You can specify multivalue compression for columns defined with distinct UDT types based on the following predefined data types:

- All numeric types
- DATE
- CHARACTER and GRAPHIC
- VARCHAR and VARGRAPHIC
- BYTE
- VARBYTE

Multivalue compression is not supported for columns defined with the following data types:

- BLOB
- CLOB
- ARRAY/VARRAY
- Structured UDTs
- Period
- XML
- Geospatial
- JSON
- DATASET

Multivalue compression is not supported for:

- Any column that is a member of the primary index column set for a table.
- Row-level security constraint columns.
- Partitioning columns.

Source and target tables with the same compress attributes can use fast path INSERT ... SELECT. Columns defined with non-matching COMPRESS attributes cannot participate in fast path INSERT ... SELECT operations. If you perform an INSERT ... SELECT on a target table containing compressed

columns defined with COMPRESS attributes that do not match, the Optimizer does not specify fast path optimization for the access plan it creates.

Usually, the performance advantages of multivalue compression offset the cost of not being able to use fast path INSERT ... SELECT.

See [ALTER TABLE](#).

You can specify multivalue compression and algorithmic compression in either order.

For a detailed description of multivalue compression, see *Teradata Vantage™ - Database Design*, B035-1094.

For more information about the COMPRESS attribute, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

constant

The specified value or list of values are compressed.

For information about limits for this value, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

NULL

Nulls are compressed for the column.

The following rules apply:

- You cannot specify NULL if the column is defined as NOT NULL.
- LOB-based UDT nulls cannot be compressed.
- You can only specify NULL once per column.

If a column is constrained as NOT NULL, then none of the specifications in the multivalue compression list can be the literal NULL.

Example: Specifying Multivalue Compression

The following example compresses all emp_name values of Smith, Wong, and Rodriguez and all DOB values of 1972-02-29, 1976-02-29:

```
CREATE TABLE employee (
  emp_no    INTEGER      NOT NULL,
  emp_name  CHARACTER(30) NOT NULL COMPRESS ('Smith', 'Wong',
                                             'Rodriguez')
  ...
  dob       DATE         COMPRESS (NULL, DATE '1972-02-29',
                                   DATE '1976-02-29')
  ...);
```

Example: Discrepancy Between Explicit Column Data Type and Implicit Data Type of Multivalue Compression

This example demonstrates the problem with specifying compressed numeric data values in an implicit data type that differs from the explicit type specified for the column containing the value to be compressed. The problem is restricted to bidirectional conversions between the DECIMAL/NUMERIC data type and the REAL/FLOAT/DOUBLE PRECISION data type.

This example shows the case for a column defined with an explicit FLOAT data type, but a compression value specified with an implicit DECIMAL data type.

```
CREATE TABLE comptest, NO FALLBACK (
  col_1 INTEGER NOT NULL,
  col_2 FLOAT COMPRESS 0.58)
PRIMARY INDEX (col_1);
```

This example shows the case for a column defined with an explicit DECIMAL data type, but a compression value specified with an implicit FLOAT data type:

```
CREATE TABLE comptest3, NO FALLBACK (
  col_1 INTEGER NOT NULL,
  col_2 DECIMAL(3,2) COMPRESS 0.07E0 )
PRIMARY INDEX (col_1);
```

This example returns an error because the data type of the column (DECIMAL) differs from the implicit data type of the compression value specified (FLOAT).

COMPRESS USING

Algorithmic compression (ALC) for the column.

You can combine multivalue compression, algorithmic compression, and block-level compression on a table for better compression.

Note:

To avoid performance impact on other workloads, you should not use algorithmic compression with block-level compression.

For guidelines on creating algorithmic compression UDFs for UDT, BLOB, CLOB, XML, and Geospatial columns, see CREATE FUNCTION (External Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

The rules that apply to multivalue compression also apply to algorithmic compression.

Algorithmic compression is not supported for row-level security constraint columns.

If you specify COMPRESS USING, you must also specify DECOMPRESS USING. You can specify COMPRESS USING and DECOMPRESS USING in either order.

The default containing database for *compress_udf_name* is SYSUDTLIB. If *compress_udf_name* is an embedded services UDF, then its default containing database is TD_SYSFNLIB. If Vantage cannot find *compress_udf_name* in either database, the system returns a message to the requestor.

You can specify multivalue compression and algorithmic compression for the same column. Algorithmic compression is only applied to values that are not specified for multivalue compression.

You can specify multivalue compression and algorithmic compression in either order.

For database object naming rules, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

For more information about algorithmic compression, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 and *Teradata Vantage™ - Database Design*, B035-1094.

You can also specify COMPRESS USING for a column in a volatile table.

compress_udf_name

Name of the UDF to be used to algorithmically compress data in this column. For a list of data types that can be algorithmically compressed, see the COMPRESS and DECOMPRESS phrases in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

database_name

Default containing database for *compress_udf_name* is SYSUDTLIB. If *compress_udf_name* is an embedded services UDF, then its default containing database is TD_SYSFNLIB. If *compress_udf_name* is not contained in SYSUDTLIB or TD_SYSFNLIB, the system returns an error to the requestor.

Example: Specifying Algorithmic Compression

The first example in this set specifies algorithmic compression (ALC) only for *col_2*. The UDF *compress_udf* compresses all values of *col_2* and the UDF *decompress_udf* decompresses them.

```
CREATE TABLE Pendants (
  col_1 INTEGER,
  col_2 CHARACTER(10) COMPRESS USING compress_udf
                        DECOMPRESS USING decompress_udf);
```

The following example specifies algorithmic compression and multivalue compression for *col_2*, with multivalue compression specified first. The UDF *compress_udf* compresses only those values that are not specified in the multivalue list. The UDF *decompress_udf* decompresses the values compressed by *compress_udf*.

```
CREATE TABLE Pendants (
  col_1 INTEGER,
  col_2 CHARACTER(10) COMPRESS ('amethyst', 'amber')
```

```
COMPRESS USING compress_udf
DECOMPRESS USING decompress_udf);
```

The following example specifies algorithmic compression and multivalue compression for *col_2*. Algorithmic compression is performed using the UDF *compress_udf* and algorithmic decompression is performed using the UDF *udf_decompress*.

```
CREATE TABLE Pendants (
  col_1 INTEGER,
  col_2 CHARACTER(100) COMPRESS ('amethyst', 'amber')
                        COMPRESS USING compress_udf
                        DECOMPRESS USING decompress_udf);
```

This example copies the compression specified for table *t1* to table *t2*.

```
CREATE TABLE t2 AS t1 WITH NO DATA;
```

Example: Specifying Algorithmic Compression for a Column with the TIMESTAMP Data Type

The following example specifies algorithmic compression (ALC) and decompression for the column *start_time* with the *TIMESTAMP* data type using UDFs. For information on compression and decompression functions, see the compression and decompression functions in *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

```
CREATE TABLE BillDateTime
(item_ID INTEGER,
start_time TIMESTAMP(0)
COMPRESS USING TD_SYSFNLIB.ts_compress
DECOMPRESS USING TD_SYSFNLIB.ts_decompress);
```

DECOMPRESS USING

Algorithmically decompress data in this column.

For information about the rules you must follow to create valid algorithmic decompression UDFs for UDT columns, see “CREATE FUNCTION (External Form)/ REPLACE FUNCTION (External Form)” in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

If you specify *DECOMPRESS USING*, you must also specify *COMPRESS USING*. You can specify *COMPRESS USING* and *DECOMPRESS USING* in either order.

decompress_UDF_name

The name of the UDF to be used to algorithmically decompress data in this column for the following data types:

- ARRAY/VARRAY
- BLOB
- BLOB-related UDT
- BYTE
- CHARACTER
- CLOB
- CLOB-related UDT
- Geospatial
- GRAPHIC
- TIME
- TIME WITH TIME ZONE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- Distinct UDTs
- XML

Note:

Structured UDTs cannot be algorithmically compressed.

database_name

The default containing database for *decompress_UDF_name* is SYSUDTLIB. If *decompress_UDF_name* is an embedded services UDF, the default containing database is TD_SYSFNLIB. If *decompress_UDF_name* does not reside in SYSUDTLIB or TD_SYSFNLIB, the system returns an error.

column_constraint_attribute

You can specify the following column constraint forms:

- UNIQUE
- PRIMARY KEY
- CHECK (*boolean_condition*)
- REFERENCES *referenced_table_name*

You cannot specify column-level or table-level constraints for UDT columns. You cannot specify the following constraints for UDT, Geospatial, or Period columns:

- BETWEEN ... AND
- UNIQUE (*boolean_condition*)
- PRIMARY KEY
- CHECK

- REFERENCES *referenced_table_name*

CONSTRAINT

Unnamed constraints are not assigned system-generated names.

constraint_name

Optional name for a constraint.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

The CONSTRAINT *constraint_name* specification is mandatory for named constraints for UNIQUE, PRIMARY KEY, FOREIGN KEY, and CHECK when those constraints are named, and applies to column-level and table-level constraints.

It does not apply to row-level security constraints. See *row_level_security_constraint_name* CONSTRAINT later in this table.

Constraint names must conform to the rules for Vantage object names and be unique among all other constraint, primary index, and secondary index names specified in the table definition. For a description of Vantage identifiers and a list of the characters they can contain, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

See *Teradata Vantage™ - Temporal Table Support*, B035-1182 and *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 for information about the constraints that can be defined for temporal tables.

UNIQUE

A column or column set is unique, that is, any two rows in the table cannot have the same value in the uniquely-constrained column. The specified columns must be defined as NOT NULL.

You can specify a UNIQUE constraint as a:

- column attribute on a single column or column set
- table attribute, except for volatile tables

You cannot specify UNIQUE constraints on columns with the following data types:

- BLOB
- BLOB UDT
- CLOB
- CLOB UDT
- VARIANT_TYPE
- ARRAY
- VARRAY
- Period
- XML
- Geospatial

- JSON
- DATASET

The UNIQUE constraint uses a unique secondary index for nontemporal tables and a single-table join index for most temporal tables. For information about temporal tables and temporal syntax, see *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

Any system-defined secondary or single-table join indexes used for a UNIQUE constraint count toward the collective maximum of 32 secondary, hash, and join indexes per table. This includes the system-defined secondary indexes used for UNIQUE constraints.

UNIQUE constraints ensure that the uniqueness of alternate keys is enforced by the system. Columns with UNIQUE constraints can be used to create referential integrity relationships with other tables.

If a row-level security table is defined with a UNIQUE constraint, enforcement of the constraint does not execute any security policy defined for the table. UNIQUE constraints are applicable to all rows in a row-level security table, not just to user-visible rows.

You cannot define a UNIQUE constraint on a row-level security constraint column of a row-level security table.

UNIQUE constraints are valid for nontemporal and temporal tables.

For a complete list of the rules for implicitly defined unique indexes, see the discussion of primary index defaults in *Teradata Vantage™ - Database Design*, B035-1094.

The implicitly defined index is a unique primary index if all of the following conditions are true.

- An explicit primary index is not specified.
- An explicit primary key is not specified.
- This is the first unique constraint defined for the table.

If none of the previous conditions are true, the implicitly defined index is a unique secondary index.

For a temporal table, the implicitly defined index is a system-defined single-table join index. For more information, see *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

When a UNIQUE constraint is defined for a normalized table, Vantage validates the constraint with normalized rows. If the normalized row violates the UNIQUE constraint, the system returns an error message to the requestor.

For an unnamed UNIQUE column constraint, use this syntax:

```
UNIQUE
```

CONSTRAINT *constraint_name*

For a named UNIQUE column constraint, use this syntax:

```
CONSTRAINT constraint_name UNIQUE
```

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

(*column_name*)

If you specify more than one *column_name*, the unique constraint is based on the combined values of the column set.

- For an unnamed UNIQUE table constraint, use this syntax:

```
CONSTRAINT UNIQUE (column_name)
```

- For a named UNIQUE table constraint, use this syntax:

```
CONSTRAINT constraint_name UNIQUE (column_name)
```

When you specify UNIQUE as a table constraint, the constraint for the table can be defined on a maximum of 64 columns.

PRIMARY KEY

A column or column set is the primary key for *table_name*. The defined column set makes each row in the table unique. The primary key is also used to enforce referential constraints.

The column must be defined as NOT NULL.

You can specify a PRIMARY KEY constraint as a:

- column attribute for a single column or a column set
- table attribute, except for volatile tables

You cannot specify PRIMARY KEY constraints on columns with these data types:

- BLOB
- BLOB UDT
- CLOB
- CLOB UDT
- VARIANT_TYPE
- ARRAY
- VARRAY
- Period
- XML
- Geospatial
- JSON
- DATASET

A table can only have one primary key. To specify candidate primary keys for referential integrity relationships with other tables, use the UNIQUE column attribute. This is not necessary for Referential Constraints, but is required for standard referential integrity constraints and batch referential integrity constraints.

PRIMARY KEY constraints are valid for nontemporal and temporal tables. For information about temporal tables and temporal syntax, see *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

For a PRIMARY KEY constraint, nontemporal tables use a system-defined secondary index and temporal tables use a single-table join index. System-defined secondary or single-table join indexes used for this constraint count toward a maximum of 32 secondary, hash, and join indexes per table.

Like UNIQUE constraints, PRIMARY KEY constraints ensure that the uniqueness of alternate keys is enforced when they are specified in a referential integrity relationship.

When a PRIMARY KEY constraint is defined for a normalized table, Vantage validates the constraint with normalized rows. If the normalized row violates the PRIMARY KEY constraint, the system returns an error to the requestor.

For an unnamed PRIMARY KEY column constraint, use this syntax:

```
PRIMARY KEY
```

CONSTRAINT *constraint_name*

For a named PRIMARY KEY column constraint, use this syntax:

```
CONSTRAINT constraint_name PRIMARY KEY
```

Use the PRIMARY KEY column attribute to apply the constraint to a single column.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

Example: Specifying Table-Level Named CHECK, PRIMARY KEY, UNIQUE, and FOREIGN KEY Constraints

The request in this example names constraints at the table level.

```
CREATE TABLE good_2 (
  column_1 INTEGER NOT NULL,
  column_2 INTEGER NOT NULL,
  column_3 INTEGER NOT NULL,
  column_4 INTEGER NOT NULL,
  column_5 INTEGER,
  column_6 INTEGER,
  CONSTRAINT primary_1
  PRIMARY KEY (column_1, column_2),
  CONSTRAINT unique_1
  UNIQUE (column_3, column_4),
  CONSTRAINT check_1
  CHECK (column_3 > 0 OR column_4 IS NOT NULL),
```

```

CONSTRAINT reference_1
FOREIGN KEY (column_5, column_6)
REFERENCES parent_1 (column_2, column_3));

```

Example: Specifying a Mix of Column-Level and Table-Level Named and Unnamed PRIMARY KEY, UNIQUE, and FOREIGN KEY Constraints

This statement defines named and unnamed constraints at the column and table levels.

```

CREATE TABLE good_3 (
  column_1 INTEGER NOT NULL
  CONSTRAINT primary_1
  PRIMARY KEY,
  column_2 INTEGER NOT NULL
  CONSTRAINT unique_1 UNIQUE
  CONSTRAINT check_1
  CHECK (column_2 <> 3)
  CONSTRAINT reference_1
  REFERENCES parent_1
  CHECK (column_2 > 0)
  REFERENCES parent_1 (column_4),
  column_3 INTEGER NOT NULL,
  column_4 INTEGER NOT NULL,
  column_5 INTEGER,
  CONSTRAINT unique_2 UNIQUE (column_3),
  CONSTRAINT check_2
  CHECK (column_3 > 0 AND column_3 < 100),
  CONSTRAINT reference_2
  FOREIGN KEY (column_3)
  REFERENCES parent_1 (column_5), UNIQUE (column_4),
  CHECK (column_4 > column_5),
  FOREIGN KEY (column_4, column_5)
  REFERENCES parent_1 (column_6, column_7));

```

CHECK Constraint Column Attribute

An optionally named simple boolean conditional expression used to constrain the values that can be inserted into, or updated for, a column.

A column attribute CHECK constraint cannot reference other columns in the same table or another table.

When you specify multiple CHECK constraints on a single column:

- Multiple unnamed column-level CHECK constraints are combined into a single column-level CHECK constraint.
- Multiple named column-level CHECK constraints are processed individually.

You can specify column-level CHECK constraints for nontemporal and temporal tables. See *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

CHECK (*boolean_condition*)

The *boolean_condition* must reference a *column_name*.

For an unnamed CHECK column constraint, use this syntax:

```
CHECK (boolean_condition)
```

CONSTRAINT *constraint_name*

For a named CHECK column constraint, use this syntax:

```
CONSTRAINT constraint_name CHECK (boolean_condition)
```

Example: Specifying a Mix of Column-Level and Table-Level Named and Unnamed CHECK Constraints

The request in this example combines the three unnamed CHECKs for *column_1*. Constraint *check_0* and each of the named CHECKs for *column_2* are treated as table constraints.

```
CREATE TABLE good_4 (
  column_1 INTEGER
    CHECK (column_1 > 0)
    CHECK (column_1 < 999)
    CHECK (column_1 NOT IN (100,200,300))
  CONSTRAINT check_0
    CHECK (column_1 IS NOT NULL),
  column_2 INTEGER
  CONSTRAINT check_1
    CHECK (column_2 > 0)
    CHECK (column_2 < 999));
```

Example: Specifying Column-Level Named CHECK Constraints

The request in this example names constraints at the column level.

```
CREATE TABLE good_1 (
  column_1 INTEGER NOT NULL
    CONSTRAINT primary_1 PRIMARY KEY,
  column_2 INTEGER NOT NULL
    CONSTRAINT unique_1 UNIQUE,
  column_3 INTEGER
    CONSTRAINT check_1 CHECK (column_3 > 0);
```

REFERENCES Constraint Column Attribute

REFERENCES as a column attribute. The syntax varies depending on whether or not the constraint is named:

- For an unnamed REFERENCES column constraint.

```
REFERENCES referenced_table_name (referenced_column_name)
```

- For a named REFERENCES column constraint.

```
CONSTRAINT constraint_name REFERENCES referenced_table_name  
(referenced_column_name)
```

The following rules apply to column attribute REFERENCES constraints only.

- Do not specify the SQL text FOREIGN KEY (*referencing_column*) for a column attribute foreign key constraint. The referencing column is implicit in a column foreign key constraint and the FOREIGN KEY keywords are only used for table constraint foreign key specifications.
- If you do not specify *column_name*, the referenced table must have a simple primary key, and the specified foreign key column references that primary key column in the referenced table.
- If you specify *column_name*, it must refer to the simple primary key of the referenced table or to a simple alternate key in the referenced table that is defined as UNIQUE.

This rule does not apply for Referential Constraints. In such cases, the candidate key acting as the primary key in the referenced table need not be explicitly declared to be unique using the PRIMARY KEY or UNIQUE keywords or by declaring it to be a USI in the table definition.

However, the candidate key in the relationship actually must always be unique even if it is not explicitly declared to be unique. Otherwise, you can produce incorrect results and corrupt your databases in some situations when appropriate care is not taken to ensure that data integrity is maintained. This is true by definition. For more information, see *Teradata Vantage™ - Database Design*, B035-1094.

As is always true when you specify Referential Constraints, you must assume responsibility for ensuring that any candidate key in a referential integrity relationship is unique, just as you must assume responsibility for ensuring that the referential relationship it anchors holds true in all cases, because Vantage enforces neither constraint.

referenced_table_name

The name of the table that contains the primary key or alternate key referenced by the specified *referencing_column*.

(referenced_column_name)

The name of the column in *referenced_table_name* that contains the primary key or alternate key referenced by *referencing_column*.

CONSTRAINT *constraint_name*

The name of the column attribute foreign key constraint.

Example: Specifying Referential Integrity Constraints

In this example, table_1 is created with column-level and table-level foreign key constraints that specify the referential integrity constraints described in the following table.

This specific reference in the example ...	Is this level constraint ...
<p>explicit foreign key columns (column_1, column_2) in table_1 defining a foreign key to table_2.</p> <p>For this to work, there must be columns identically typed to the (column_1,column_2) columns in table_1 (call them upi_column_1 and upi_column_2) that constitute the unique primary index columns for table_2.</p> <p>In other words, columns upi_column_1 and upi_column_2 in table_2 must both be defined as type INTEGER and must constitute the unique primary index for table_2. Otherwise, an error is returned.</p>	Table
<p>implicit foreign key column_1 in table_1 explicitly referencing column_1 in table_3. table_1 (column_1) and table_3 (column_1) must both be typed INTEGER, but table_3 (column_1) need not be the unique primary index for table_3.</p>	Column

```
CREATE TABLE table_1
  (column_1 INTEGER NOT NULL REFERENCES table_3(column_1),
   column_2 INTEGER,
  FOREIGN KEY (column_1, column_2) REFERENCES table_2)
PRIMARY INDEX (column_1);
```

Example: Specifying a Column-Level Foreign Key Constraint

The first request creates a column-level referential integrity constraint on column a1 with column b1 in table b. Referential integrity is enforced for this relationship.

The second request specifies a column-level Referential Constraint relationship on column a1 with column b1 in table b. Referential integrity is not enforced for this relationship.

```
CREATE TABLE a (
  a1 INTEGER REFERENCES b(b1),
  a2 CHARACTER(10))
PRIMARY INDEX (a1);

CREATE TABLE a (
  a1 INTEGER REFERENCES WITH NO CHECK OPTION b(b1),
```

```
a2 CHARACTER(10))
PRIMARY INDEX (a1);
```

identity_column

You cannot specify identity columns for nonpartitioned NoPI tables.

If you specify an identity column for a table that is used with Teradata Unity, you must enable Change Data Multicast (CDM) in Unity and Vantage. See *Teradata® Unity™ User Guide*, B035-2520.

Identity columns are ANSI SQL:2011-compliant.

UDT, Geospatial, and Period columns are not supported as Identity columns.

See [Identity Column Parameters](#) and [column_constraint_attributes](#).

GENERATED ALWAYS AS IDENTITY

Identity column values are always system-generated.

You cannot insert values into, or update, an identity column defined as GENERATED ALWAYS AS IDENTITY.

GENERATED BY DEFAULT AS IDENTITY

Identity column values can be system-generated or user-inserted:

- When you insert a null into the column, the system generates an identity column value to replace it.
- When you insert a value into the column, the system inserts the value into the identity column.

Identity Column Parameters

The identity column parameters are optional and can be specified in any order.

START WITH *constant*

The lowest integer value in the system-generated numeric series for an identity column. The default is 1.

The value you specify can be any exact negative or positive integer within the range of the data type for the column as long as it is less than MAXVALUE for an incremental series or greater than MINVALUE for a decremental series.

INCREMENT BY *constant*

The interval on which to increment system-generated numbers.

The value you specify can be a negative integer. The default is 1.

The value can be any integer (except 0) less than or equal to the value of DBS Control parameter IdCol Batch Size.

MINVALUE *constant*

The minimum value to which a system-generated numeric series can decrement. MINVALUE applies only to system-generated numbers.

You can specify any integer value with an absolute value less than the value specified for START WITH.

The default is the minimum integer number for the data type defined for the column.

When cycling is not enabled, the sum of the specified values for START WITH and INCREMENT BY must be greater than MINVALUE. If they are not, then Vantage generates only one number before the minimum limit is exceeded.

NO MINVALUE

Numbering restarts its cycle from the minimum value that can be expressed for the data type of the column when the MAXVALUE limit is reached.

You can only specify NO MINVALUE when the INCREMENT BY interval is a negative number. The default is the minimum value for the data type specified for the identity column.

The following rules apply to INCREMENT BY and CYCLE specifications for MINVALUE.

- If INCREMENT BY is a positive number and you also specify CYCLE, then renumbering begins from MINVALUE when MAXVALUE is reached.
- If INCREMENT BY is a negative number, then MINVALUE, if specified, must be a whole number such that $\text{MINVALUE} \leq \text{START WITH}$.
- If you do not specify INCREMENT BY, but do specify NO CYCLE, then MINVALUE is not applicable for positive increments.

No warning or error is returned if you specify a MINVALUE with NO CYCLE.

MAXVALUE *constant*

The maximum value to which a system-generated numeric series can increment. MAXVALUE applies only to system-generated numbers. Its value can be any integer with a value that is greater than the value specified for START WITH.

The default is the maximum number for the data type defined for the column.

When cycling is not enabled, the sum of the specified values for START WITH and INCREMENT BY must be less than MAXVALUE. If it is not, then Vantage generates only one number before the maximum limit is exceeded.

The following rules apply to INCREMENT BY and CYCLE specifications for MAXVALUE.

- If INCREMENT BY is a positive number, but you do not specify CYCLE, then MAXVALUE, if specified, must be a whole number such that MAXVALUE ³ START WITH.
MAXVALUE cannot be larger than the maximum value for the data type assigned to the identity column.
- If INCREMENT BY is a negative number and you also specify CYCLE, then renumbering begins with MAXVALUE when MINVALUE is reached.
- If you do not specify INCREMENT BY, but do specify NO CYCLE, then MAXVALUE is not applicable for negative increments.

No warning or error is returned if you specify a MAXVALUE with NO CYCLE.

NO MAXVALUE

Numbering restarts its cycle from the minimum value that can be expressed for the data type of the column when the maximum value for the type is reached.

You can only specify NO MAXVALUE when the INCREMENT BY interval is a positive number. The default is the maximum value for the data type specified for the identity column.

CYCLE

System-generated values can be recycled when their minimum or maximum is reached.

NO CYCLE

This is the default.

COLUMN

If you precede a column grouping that is specified in the column list of the table definition with the keyword COLUMN, then Vantage stores the column partition, which consists of the columns in the specified grouping, using COLUMN format.

ROW

If you precede a column grouping that is specified in the column list of the table definition with the keyword ROW, then Vantage stores the column partition, which consists of the columns in the grouping, using ROW format.

(column_name)

Group the values from the specified column set of 1 or more columns into the same partition of a column-partitioned table. You can specify a single column in a partition.

This option is only valid for column-partitioned tables.

You should consider grouping columns into a column partition when either of the following conditions occur:

- The columns are frequently accessed together by queries.
- The columns are infrequently accessed or autocompression on the individual columns or subsets of columns is not effective.

You cannot specify a column to be in more than one partition.

If you do not specify a column grouping for a COLUMN specification in the PARTITION BY clause, Vantage defines a column partition for each individual column and column group specified in the column list for *table_name*.

AUTO COMPRESS

You can only specify this option for columns or column partitions of a column-partitioned table. Vantage applies autocompression for a physical row on a per container basis. For efficiency, the system may use the autocompression method chosen for the previous container, including not using autocompression, if that is more effective.

AUTO COMPRESS

Apply auto compression to values inserted into this column partition.

NO AUTO COMPRESS

Do not apply auto compression to values inserted into this column partition.

PERIOD FOR

A derived period column.

A DATE or timestamp column can be part of only one derived period column definition and can only be used as the start of the period or end of the period, not both. The data type of begin and end columns in a derived period column must match, including the same properties, the same DATE or timestamp data type, and identical format column attribute.

A table can have more than one derived period column. For each derived period column you add, the maximum limit of 2048 non-derived period columns is reduced by one.

You cannot perform a period arithmetic operation on a derived period column, except in a WHERE or ON clause.

A derived period column cannot:

- be projected
- be part of a primary or secondary index
- be specified as part of a PARTITION BY COLUMN clause
- contain column attributes

period_name

Name of the derived period column and cannot match the name of any other column in the table.

period_begin_column

DATE or timestamp column to use as the start of the period.

period_end_column

DATE or timestamp column to use as the end of the period.

You can specify UNTIL_CHANGED.

Example: Creating a Table with a Derived Period Column

This example creates a table, `employee`, with a derived period column, `jobduration`, where `jobstart` is the period begin column name and `jobend` is the period end column name.

```
CREATE TABLE employee (
  eid INTEGER NOT NULL,
  name VARCHAR(100) NOT NULL,
  deptno INTEGER NOT NULL,
  jobstart DATE NOT NULL,
  jobend DATE NOT NULL,
  PERIOD FOR jobduration(jobstart, jobend)
) PRIMARY INDEX(eid);
```

normalize_option

A normalize condition for *normalize_column*.

To coalesce 2 values from a Period column or derived Period column that meet or overlap. The result of a normalization operation on 2 input values is a third Period value that is the union of the 2 input Period values.

You can only specify the NORMALIZE option once for a table.

A normalized table does not have two rows with the same data values whose normalize column values meet or overlap. Because of this, if you insert a new row or update an existing row in the table, it is coalesced with all existing rows whose data values are the same as the new row and whose normalize column values overlap or meet.

You cannot specify the NORMALIZE option for a volatile table.

The NORMALIZE option specifies a mandatory single normalize column and normalization condition with an optional ALL BUT column list to exclude the specified columns from the normalization process.

ALL BUT (*normalize_ignore_column_name*)

Exclude the specified column set from the normalization process.

To normalize a table defined with columns that have a BLOB, CLOB, XML, Geospatial, JSON, or DATASET data type, you must explicitly specify those columns as members of the *normalize_ignore_column_name* column set.

ON *normalize_column*

The Period or derived Period column on which the table is to be normalized.

This cannot be a TRANSACTIONTIME column, but can be a VALIDTIME or other period column.

ON OVERLAPS

Normalize the table based on when the two input Period or derived Period columns overlap in time.

ON MEETS OR OVERLAPS

ON OVERLAPS OR MEETS

Normalize the table based on when the two input Period or derived Period columns either overlap in time or are temporally continuous, but do not overlap.

You cannot specify ON MEETS as a normalize condition.

Example: NORMALIZE

This CREATE TABLE request defines a table that normalizes on the *duration* column. Rows are normalized only when values of the columns *emp_id*, *project_name*, and *dept_id* are the same and the Period values for *duration* meet or overlap.

```
CREATE TABLE project (
  emp_id      INTEGER,
  project_name VARCHAR(20),
  dept_id     INTEGER,
```

```
duration    PERIOD(DATE),
NORMALIZE ON duration);
```

The following CREATE TABLE request defines a similar *project* table that also normalizes on the *duration* column, but specifies *dept_id* as a column to ignore for normalization. For such a table, a row is normalized only when the values of the columns *emp_id* and *project_name* are the same and Period values for *duration* meet or overlap.

```
CREATE TABLE project (
  emp_id      INTEGER,
  project_name VARCHAR(20),
  dept_id     INTEGER,
  duration    PERIOD(DATE),
  NORMALIZE ALL BUT(dept_id) ON duration);
```

table_constraint

You can specify these attributes for a table.

PRIMARY KEY (*column_name*)

A column or column set as the PRIMARY KEY constraint for the table. Columns specified in a PRIMARY KEY constraint must all be defined as NOT NULL.

For nontemporal tables, the PRIMARY KEY constraint implicitly uses a USI.

For temporal tables, the PRIMARY KEY constraint implicitly uses a single-table join index or, in certain cases, a USI. For details, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

For an unnamed PRIMARY KEY table constraint, use this syntax:

```
CONSTRAINT PRIMARY KEY (column_name)
```

CONSTRAINT *constraint_name*

For a named PRIMARY KEY table constraint, use this syntax:

```
CONSTRAINT constraint_name PRIMARY KEY (column_name)
```

If you specify multiple columns, the primary key constraint is based on the combined values of the specified columns. The primary key for the table can consist of up to 64 columns.

CHECK (boolean_condition)

If you specify CHECK as a composite table attribute, the syntax varies depending on whether the constraint is named or not.

The following rules apply only to table attribute CHECK constraints:

- A table-level CHECK constraint must reference at least 1 column from its table.
- A maximum of 100 table-level constraints can be defined for a given table.

A table attribute CHECK constraint can compare any columns defined for its table, both against each other and against constants.

For an unnamed CHECK column constraint, use this syntax:

```
CHECK (boolean_condition)
```

You can specify CHECK constraints as column attributes or as table attributes.

You cannot include subqueries, aggregate, or ordered analytic functions in a CHECK constraint.

You cannot specify CHECK constraints for identity columns or for the columns of a volatile table.

Note:

You can create a constraint expression that is too large to be parsed for INSERT and UPDATE requests by specifying a combination of table-level, column-level, and FOREIGN KEY WITH CHECK OPTION constraints on a table that underlies a view.

You cannot specify CHECK constraints on columns with the following data types:

- BLOB
- CLOB
- UDT
- ARRAY/VARRAY
- Period
- XML
- Geospatial
- JSON
- DATASET

Vantage tests CHECK constraints for character columns using the current session collation. As a result, a CHECK constraint might be met for one session collation, but not met for another even though the identical data is inserted or updated for both.

Unnamed CHECK constraints with identical boolean conditions and case are considered duplicates, and the system returns an error when you specify them.

For example, the following CREATE TABLE request is valid because the case of f1 and F1 is different:

```
CREATE TABLE t1 (
  f1 INTEGER, CHECK (f1 > 0), CHECK (F1 > 0));
```

The following CREATE TABLE request is not valid because the case is the same for both f1 specifications.

```
CREATE TABLE t1 (
  f1 INTEGER, CHECK (f1 > 0), CHECK (f1 > 0));
```

For column attribute and table attribute CHECK constraints, you can specify any simple boolean search condition. CHECK constraint definitions cannot include subqueries, aggregate expressions, or ordered analytic expressions.

You cannot specify CHECK constraints for:

- Volatile table columns.
- Identity columns.

The search condition for a CHECK constraint cannot specify SET operators.

Vantage supports the following non-ANSI SQL constraint syntax for table attribute CHECK constraints.

```
BETWEEN value_1
AND value_2
```

The system processes the syntax like the following ANSI-compliant constraint.

```
CHECK (column_name
BETWEEN value_1
AND value_2)
```

You can use the BETWEEN ... AND operator as a form of CHECK constraint except for volatile table columns, identity columns, UDT columns, ARRAY, VARRAY, Geospatial columns, Period columns, BLOB columns, or CLOB columns.

For information about using CHECK constraints to construct domains, see *Teradata Vantage™ - Database Design*, B035-1094.

For a normalized table, Vantage validates CHECK constraints on the input row or new row.

You cannot define a CHECK constraint on a row-level security constraint column of a row-level security table.

If a row-level security table is defined with 1 or more CHECK constraints, the enforcement of those constraints does not execute any UDF security policies that are defined for the table. The enforcement of the CHECK constraint applies to the entire table. This means that CHECK constraints apply to all of the rows in a table, not just to the rows that are user-visible.

CONSTRAINT *constraint_name*

An optionally named simple boolean conditional expression used to constrain the values that can be inserted into, or updated for, a column. For a named CHECK column constraint, use this syntax:

```
CONSTRAINT constraint_name CHECK (boolean_condition)
```

FOREIGN KEY

A foreign key reference from a column set in this table to the primary key or an alternate key in another table.

You can specify a foreign key REFERENCES constraint as a column attribute or as a table attribute.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

The following rules apply to foreign key REFERENCES constraints.

- You must either have the REFERENCES privilege on the referenced table or on all specified columns of the referenced table.
- A maximum of 64 foreign keys can be defined for a table and a maximum of 64 referential constraints can be defined for a table.

Similarly, a maximum of 64 other tables can reference a single table. Therefore, there is a maximum of 128 reference indexes that can be stored in the table header per table, but only 64 of these, the reference indexes that map the relationship between the table and its child tables, are stored per reference index subtable. The table header limit on reference indexes includes both references to and from the table.

For information about temporal tables and foreign key REFERENCES constraints, see *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

Although you can create a child table before the parent table exists, a foreign key REFERENCES constraint that makes a forward reference to a table that has not yet been created cannot qualify the parent table name with a database name.

The forward-referenced parent table that has not yet been created is assumed to be in the same database as the child table currently being created.

Each column in the foreign key *referenced_column_name* list must correspond to a column of *referenced_table_name* in REFERENCES *referenced_table_name*, and you cannot specify the same column name more than once.

The foreign key column list must contain the same number of column names as the referenced primary or alternate key in *table_name*. The *i*th column of the referencing list corresponds to the *i*th column identified in the referenced list.

The data type of each foreign key referencing column must be the same as the data type of the corresponding REFERENCES referenced column.

Each individual foreign key can be defined on a maximum of 64 columns.

A maximum of 100 table-level constraints can be defined for any table.

You cannot specify foreign key REFERENCES constraints on columns with the following data types:

- BLOB
- CLOB
- UDT
- ARRAY/VARRAY
- Period
- XML
- Geospatial
- JSON
- DATASET

You cannot specify a foreign key REFERENCES constraint on an identity column.

Foreign key REFERENCES constraints can be null.

Though foreign key REFERENCES constraints can be unique, this is rare.

An example of when a foreign key would be unique is a vertical partitioning of a logical table into multiple tables.

You cannot specify foreign key REFERENCES constraints for global temporary trace, volatile, or queue tables. See [CREATE TABLE \(Queue Table Form\)](#).

Foreign key REFERENCES constraints cannot be copied to a new table using the copy table syntax.

You can specify a mix of standard referential integrity constraints, batch referential integrity constraints, and referential constraints for the same table, but not for the same column sets. For a description, see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

Example: Specifying Batch Referential Integrity Constraints

This example creates a table-level batch referential integrity constraint on column d1 of child table drs.t2, which refers to column c1 in parent table drs.t1. Referential integrity is enforced for this constraint.

```
CREATE SET TABLE drs.t1, NO FALLBACK,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL
(
    c1 INTEGER NOT NULL,
    c2 INTEGER NOT NULL,
    c3 INTEGER NOT NULL)
```

```

UNIQUE PRIMARY INDEX (c1);
CREATE SET TABLE drs.t2, NO FALLBACK,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL
(
    d1 INTEGER,
    d2 INTEGER,
    d3 INTEGER,
    FOREIGN KEY (d1) REFERENCES WITH CHECK OPTION drs.t1 (c1));

```

Example: Specifying a Table-Level Foreign Key Constraint

The first request creates a table-level standard referential integrity constraint on foreign key column a2 with column d1 in table d. Referential integrity is enforced for this relationship.

The second request creates a table-level Referential Constraint relationship on column a2 with column d1 in table d. Referential integrity is not enforced for this relationship.

```

CREATE TABLE a (
    a1 INTEGER,
    a2 CHARACTER(10),
    a3 INTEGER,
    FOREIGN KEY (a2) REFERENCES d(d1))
PRIMARY INDEX (a1);

CREATE TABLE a (
    a1 INTEGER,
    a2 CHARACTER(10),
    a3 INTEGER,
    FOREIGN KEY (a2) REFERENCES WITH NO CHECK OPTION d(d1))
PRIMARY INDEX (a1);

```

WITH NO CHECK OPTION

Referential integrity is not to be enforced for the specified primary key-foreign key relationship.

For a column attribute foreign key constraint, specify this clause as follows.

```

REFERENCES WITH NO CHECK OPTION referenced_table_name
                        (referenced_column_name)

```

For a table attribute foreign key constraint, specify this clause as follows.

```
FOREIGN KEY (referencing_column_name) REFERENCES WITH NO CHECK
OPTION (referenced_table_name referenced_column_name)
```

row_level_security_constraint_column_name **CONSTRAINT**

The name of a constraint object that you want to define as a row-level security constraint column for *table_name*. The constraint object must exist in the database. See [CREATE CONSTRAINT](#).

Each specified row-level security constraint name must be unique among column names for the table.

You cannot create a row-level security constraint column for a global temporary table or for a volatile table.

A security constraint object definition includes a name, data type, and nullability specification when it is created. You cannot assign any additional attributes to a security constraint column when creating a table.

You cannot compress a row-level security constraint column using either multivalue compression or algorithmic compression. However, there are no restrictions on using auto-compression or block compression for a row-level security table with column partitions.

You can define a maximum of 5 row-level security constraint columns for a table.

Note:

When using referential integrity (RI) on tables that define security constraint columns, the system does not recognize security constraints when checking referential integrity for either the parent and child RI table. Execution of any request that access such RI tables continues as if the tables had no row-level security constraints.

index

Keyword that defines a nonunique secondary index (NUSI) for the table.

You can define a secondary index on a column that has a UDT data type.

You can define a nonunique secondary index on a column that has a Geospatial data type.

If you specify INDEX without the preceding keyword UNIQUE, the index is a nonunique secondary index (NUSI).

The INDEX list is an extension to ANSI SQL.

Unlike the indexes created by the UNIQUE and PRIMARY KEY constraint definitions, indexes defined by the index list can have nullable columns.

You cannot define a secondary index on a column defined with any of the following data types:

- BLOB
- CLOB

- LOB UDT
- VARIANT_TYPE
- ARRAY
- VARRAY
- Period
- XML
- JSON
- DATASET

index_name

Optional name for the index.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

ALL

Ignore the assigned case specificity for a column. This property enables a NUSI defined with the ALL option to do the following:

- Include case-specific values.
- Cover a table or join index on a NOT CASESPECIFIC column set.

ALL enables a NUSI to cover a query, enhancing performance by eliminating the need to access the base table itself when all columns needed by a query are contained in the NUSI.

Be aware that specifying the ALL option might also require additional index storage space.

You cannot specify multiple NUSIs that differ only by the presence or absence of the ALL option.

You cannot specify the ALL option for primary or unique secondary indexes.

index_column_name

Column set whose values are to be used as the basis for a secondary index.

Columns in the list cannot have any of the following data types: BLOB, CLOB, Period, XML, Geospatial, JSON, or DATASET.

If you specify more than one column name, the index is created on the combined values of each column named. A maximum of 64 columns can be specified for an index, and a maximum of 32 secondary indexes can be created for one table.

PRIMARY INDEX

The primary index definition.

The primary index is used by the hashing algorithm to distribute table rows across the AMPs.

Definitions that do not explicitly specify PRIMARY INDEX (*column_list*) or NO PRIMARY INDEX use the setting of the DBS Control parameter PrimaryIndexDefault, except for column-partitioned tables, which are always defined by default with NO PRIMARY INDEX regardless of the PrimaryIndexDefault setting. For information about DBS Control parameters, see *Teradata Vantage™ - Database Utilities*, B035-1102.

For column-partitioned tables, the default table kind is MULTISSET. This cannot be changed for a column-partitioned table.

For information about primary index defaults, see *Teradata Vantage™ - Database Design*, B035-1094.

index_column_name

A column in the column set that defines a partitioned primary index.

If you specify more than one column name, the index is created on the combined values of each column named. A maximum of 64 columns can be specified for an index, and a maximum of 32 secondary indexes can be created for one table. A multicolumn NUSI defined with an ORDER BY clause counts as two consecutive indexes in this calculation.

You can define a primary index on a column that has a UDT data type.

You cannot define a primary index on a column defined with any of the following data types:

- BLOB
- CLOB
- LOB UDT
- VARIANT_TYPE
- ARRAY
- VARRAY
- Period
- XML
- Geospatial
- JSON
- DATASET

You cannot define a primary index on a column that is a row-level security constraint.

index_name

Optional name for the index.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

UNIQUE

The primary index is to be unique.

Any secondary indexes and the primary index can be defined to be unique with the exception of a PI whose definition does not include all of the partitioning columns, if any.

Example: Specifying a UPI and a USI on the Same Table

In this example, a unique primary index is defined on the empno column, and a unique secondary index is defined on the name column. Because the empno and name columns must always contain a value, they are assigned the NOT NULL attribute.

For display purposes, the values for sex , race , and mstat are defined using the UPPERCASE column attribute.

```
CREATE TABLE employee (
  emp_no      SMALLINT FORMAT '9(5)'
              CHECK (emp_no >= 10001 AND emp_no <= 32001) NOT NULL,
  name        VARCHAR(12) NOT NULL,
  dept_no     SMALLINT FORMAT '999'
              CHECK (deptno >= 100 AND dept_no <= 900),
  job_title   VARCHAR(12),
  salary      DECIMAL(8,2) FORMAT 'ZZZ,ZZ9.99'
              CHECK (salary >= 1.00 AND salary <= 999000.00),
  yrs_exp     BYTEINT FORMAT 'Z9'
              CHECK(yrs_exp >= -99 AND yrs_exp <=99),
  dob         DATE FORMAT 'MMbDDbYYYY' NOT NULL,
  sex         CHARACTER UPPERCASE NOT NULL
              CHECK (sex IN ('M','F')),
  race        CHARACTER UPPERCASE,
  m_stat      CHARACTER UPPERCASE
              CHECK (m_stat IN ('S','M','D','U')),
  edlev       BYTEINT FORMAT 'Z9'
              CHECK (ed_lev >=0 AND ed_lev <=22) NOT NULL,
  h_cap       BYTEINT FORMAT 'Z9'
              CHECK (h_cap >= -99 AND h_cap <= 99)
  UNIQUE PRIMARY INDEX (emp_no),
  UNIQUE INDEX (name);
```

Example: Specifying an Identity Column as a Unique Primary Index

This example creates an identity column that is used as the primary index for the table. Values for idnum are generated starting with 1,000 and incrementing by 10 until the specified MAXVALUE of 300,000 is reached.

The numbers generated for rows inserted into the table are the following: 1,000, 1,010, 1,020, ..., 300,000 and then an error is returned when the maximum value is reached.

```
CREATE TABLE t (
  idnum INTEGER GENERATED ALWAYS AS IDENTITY
    (START WITH 1000
     INCREMENT BY 10
     MINVALUE 0
     MAXVALUE 300000),
  phone INTEGER)
UNIQUE PRIMARY INDEX(idnum);
```

This example does not specify a CYCLE parameter.

CYCLE	Description
Not specified.	Specification defaults to NO CYCLE.
Specified.	Numbering sequence generated is as follows: 1,000, 1,010, 1,020 ... 300,000, 0, 10, 20, 30 ...

No warning message is returned when numbers are cycled. However, if an idnum value is still in existence when an attempt is made to recycle it, the request aborts because idnum is the UPI for the table, so all its values must be unique.

If there is a system restart, the numbering sequence might not be in strict increments of 10 from the last generated number. The next available number stored in DBC.IdCol.AvailValue for that particular table is retrieved and numbering begins from there. Unassigned numbers last reserved and stored in the cache are lost.

Example: Specifying an Identity Column as a Unique Primary Index Using CYCLE

This example also creates an identity column that is used as the primary index for the table. Values for idnum are generated starting with 1,000 and decremented by 10 until the specified MINVALUE of 1 is reached. Because CYCLE is specified, no error is returned and numbering proceeds as follows: 1,000, 990, 980 ... 10, 100,000, 99,990, 99,980 ...

However, if an idnum value is still in existence when an attempt is made to recycle it, the request aborts because idnum is the UPI for the table, so all its values must be unique.

```
CREATE TABLE t (
  idnum INTEGER GENERATED ALWAYS AS IDENTITY
    (START WITH 1000
     INCREMENT BY -10
     MINVALUE 1
     MAXVALUE 100000 CYCLE),
  phone INTEGER)
UNIQUE PRIMARY INDEX(idnum);
```

Example: Specifying a Primary Index and a Primary Key

The request in this example specifies both a primary index and primary key.

The Primary Key (column_1) is mapped to a unique secondary index. The UNIQUE constraint on column_3 also is mapped to a USI. These two unique secondary indexes cannot be null. The Unique Primary Index (column_2) can be null; however, no more than one row in the table can have a null UPI value.

```
CREATE TABLE good_5 (
  column_1 INTEGER NOT NULL PRIMARY KEY,
  column_2 INTEGER,
  column_3 INTEGER NOT NULL UNIQUE,
  column_4 INTEGER)
UNIQUE PRIMARY INDEX (column_2);
```

Example: Primary Index with Column Partitioning

Following is an example of a CREATE TABLE statement for a CP table with a primary index:

```
CREATE TABLE Sales4 (
  storeid      INTEGER NOT NULL,
  productid    INTEGER NOT NULL,
  salesdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  totalrevenue DECIMAL(13,2),
  note         VARCHAR(256)
)
PRIMARY INDEX (storeid, productid)
PARTITION BY COLUMN;
```

By default for this table, each column is in a separate column partition and each of the column partitions have COLUMN format with autocompression. For the following query, this allows a single-AMP step with hashed access on this AMP and column partition elimination. In step 1, for a single hash value, three user column partitions and the delete column partition are accessed on this single AMP. While this can provide efficient access, row header compression and autocompression might not be as effective as a CP table with a primary AMP index, especially if the primary index is nearly unique.

```
EXPLAIN
SELECT SUM(totalrevenue) FROM Sales4 s
WHERE s.storeid = 37 AND s.productid = 1466;
```

Explanation

```
-----
1) First, we do a single-AMP SUM step to aggregate from 4 column
   partitions of PLS.s by way of the primary index "PLS.s.storeid =
```

- 37, PLS.s.productid = 1466" with no residual conditions. Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 1 row (23 bytes). The estimated time for this step is 0.02 seconds.
- 2) Next, we do a single-AMP RETRIEVE step from Spool 3 (Last Use) by way of the hash value of "PLS.s.storeid = 37, PLS.s.productid = 1466" into Spool 1 (one-amp), which is built locally on that AMP. The size of Spool 1 is estimated with high confidence to be 1 row (36 bytes). The estimated time for this step is 0.02 seconds.
 - 3) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.04 seconds.

Example: Unique Primary Index with Column Partitioning

This example defines a column-partitioned table with a unique primary index. Each column is in its own partition with system-determined ROW format. A 2-byte partitioning is defined. By default, a column-partitioned table is a multiset table. MULTiset can be explicitly specified but SET must not be specified. The number of bytes per row is increased by 86 due to the row headers per column partition. This example assumes autocompression is the default for column partitions.

```
CREATE TABLE Orders (
  o_orderkey INTEGER NOT NULL,
  o_custkey INTEGER,
  o_orderstatus CHAR(1) CASESPECIFIC,
  o_totalprice DECIMAL(13,2) NOT NULL,
  o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_shippriority INTEGER,
  o_comment VARCHAR(79) )
UNIQUE PRIMARY INDEX (o_orderkey) PARTITION BY COLUMN;
```

NO PRIMARY

The table is defined without a primary index or primary AMP index.

You cannot specify a column name list following a NO PRIMARY INDEX specification.

If the preceding item in the index list is a partitioning clause that is not part of an index clause, you must specify a COMMA character preceding NO PRIMARY INDEX. Otherwise, the comma is optional.

For the complete set of rules that control primary index defaults, see *Teradata Vantage™ - Database Design*, B035-1094.

INDEX

Optional keyword you can include for readability.

PRIMARY AMP

Use a primary AMP index (PA).

Rows are hash-distributed to AMPs for a column-partitioned (CP) table or join index. Column partition values are ordered on each AMP by an internal partition number and a row hash for a column-partitioned table or join index.

If you specify a PRIMARY AMP clause, you must specify a PARTITION BY clause that includes a column-partitioning level, either in the PRIMARY AMP clause or by itself in the index list.

The default Table Kind for a table with a PA is MULTiset. This cannot be changed for a table with a PA.

INDEX

Optionally, the INDEX keyword can be specified with AMP for readability.

index_name

Name of the primary AMP index. For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

(index_column_name)

The primary AMP index columns. For a composite primary index, *index_column_name* indicates a comma-separated list of all the index columns in parenthesis. You cannot specify the begin or end columns of a derived period column in a primary AMP index.

A row-level security constraint column cannot be a component of the primary AMP index.

You cannot define a primary AMP index on a column defined with the JSON or DATASET data type.

Example: Primary AMP Index with Column Partitioning

Following is an example of a CREATE TABLE statement for a column-partitioned (CP) table with a primary AMP index:

```
CREATE TABLE Sales1 (
  storeid      INTEGER NOT NULL,
  productid    INTEGER NOT NULL,
  salesdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  totalrevenue DECIMAL(13,2),
  note         VARCHAR(256)
)
```

```
PRIMARY AMP INDEX (storeid, productid)
PARTITION BY COLUMN;
```

By default for this table, each column is in a separate column partition and each of the column partitions have COLUMN format with autocompression.

For the following query, this allows a single-AMP step with column partition elimination. If this was a CP table with no primary index, all-AMP steps would be required. In step 1, three user column partitions and the delete column partition are accessed on this single AMP.

```
EXPLAIN
SELECT SUM(totalrevenue) FROM Sales1 s
WHERE s.storeid = 37 AND s.productid = 1466;
```

Explanation

-
- 1) First, we do a single-AMP SUM step to aggregate from 4 column partitions of PLS.s by way of the primary AMP index "PLS.s.storeid = 37, PLS.s.productid = 1466" with a condition of "(PLS.s.productid = 1466) AND (PLS.s.storeid = 37)". Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 1 row (23 bytes). The estimated time for this step is 0.02 seconds.
 - 2) Next, we do a single-AMP RETRIEVE step from Spool 3 (Last Use) by way of the hash value of "PLS.s.storeid = 37, PLS.s.productid = 1466" into Spool 1 (one-amp), which is built locally on that AMP. The size of Spool 1 is estimated with high confidence to be 1 row (36 bytes). The estimated time for this step is 0.02 seconds.
 - 3) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.04 seconds.

Example: Primary AMP Index with Column and Row Partitioning

The following is an example of a CREATE TABLE statement for a CP table with row partitioning and a primary AMP index:

```
CREATE TABLE Sales2 (
  storeid      INTEGER NOT NULL,
  productid    INTEGER NOT NULL,
  salesdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  totalrevenue DECIMAL(13,2),
  note         VARCHAR(256)
)
```



```
PRIMARY AMP INDEX (storeid, productid)
PARTITION BY (
    COLUMN,
    RANGE_N(salesdate BETWEEN DATE '2007-01-01' AND DATE '2014-12-31'
            EACH INTERVAL '1' MONTH)
);
```

By default for this table, each column is in a separate column partition and each of the column partitions has COLUMN format with autocompression. For the following query, this allows a single-AMP step with column and row partition elimination. If this was a CP table with no primary index, all-AMP steps would be required. In step 1, for one row partition, four user column partitions and the delete column partition are accessed on this single AMP.

```
EXPLAIN
SELECT SUM(totalrevenue) FROM Sales2 s
WHERE s.storeid = 37 AND s.productid = 1466 AND s.salesdate =
DATE '2012-05-21';
```

Explanation

-
- 1) First, we do a single-AMP SUM step to aggregate from 5 combined partitions (5 column partitions) of PLS.s by way of the primary AMP index "PLS.s.storeid = 37, PLS.s.productid = 1466" with a condition of ("(PLS.s.storeid = 37) AND ((PLS.s.productid = 1466) AND (PLS.s.salesdate = DATE '2012-05-21'))"). Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 1 row (23 bytes). The estimated time for this step is 0.02 seconds.
 - 2) Next, we do a single-AMP RETRIEVE step from Spool 3 (Last Use) by way of the hash value of "PLS.s.storeid = 37, PLS.s.productid = 1466" into Spool 1 (one-amp), which is built locally on that AMP. The size of Spool 1 is estimated with high confidence to be 1 row (36 bytes). The estimated time for this step is 0.02 seconds.
 - 3) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.04 seconds.

Primary AMP Index with Column Partitioning and ROW Format

Following is an example of a CREATE TABLE statement for a CP table with a primary AMP index that uses ROW format for the first and third column partitions (without applicable autocompression). The first column partition is a multicolumn partition. Autocompression is applied to the second column partition for totalrevenue.

```
CREATE TABLE Sales3 (
  ROW(storeid          INTEGER NOT NULL,
       productid       INTEGER NOT NULL,
       salesdate        DATE FORMAT 'yyyy-mm-dd' NOT NULL),
  COLUMN(totalrevenue  DECIMAL(13,2)),
  ROW(note             VARCHAR(256))
)
PRIMARY AMP INDEX (storeid, productid)
PARTITION BY COLUMN;
```

For the following query, this allows a single-AMP step with column partition elimination. If this was a CP table with no primary index, all-AMP steps would be required. In step 1, two user column partitions are accessed since the partition being scanned has ROW format. The delete column partition does not need to be accessed in this case since the partition being scanned has ROW format.

```
EXPLAIN
SELECT SUM(totalrevenue) FROM Sales3 s
WHERE s.storeid = 37 AND s.productid = 1466;
```

Explanation

-
- 1) First, we do a single-AMP SUM step to aggregate from 2 column partitions of PLS.s by way of the primary AMP index "PLS.s.storeid = 37, PLS.s.productid = 1466" with a condition of "(PLS.s.productid = 1466) AND (PLS.s.storeid = 37)". Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with high confidence to be 1 row (23 bytes). The estimated time for this step is 0.02 seconds.
 - 2) Next, we do a single-AMP RETRIEVE step from Spool 3 (Last Use) by way of the hash value of "PLS.s.storeid = 37, PLS.s.productid = 1466" into Spool 1 (one-amp), which is built locally on that AMP. The size of Spool 1 is estimated with high confidence to be 1 row (36 bytes). The estimated time for this step is 0.02 seconds.
 - 3) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.04 seconds.

Example: Primary AMP Index with Column Partitioning and a USI

This example defines a column-partitioned table with a primary AMP index. Note that a primary AMP index cannot be defined as unique but a unique (or nonunique) secondary index may be defined on the same columns as the primary AMP index. Each column is in its own partition with system-determined COLUMN format and autocompression except for o_comment which is in its own

partition with user-specified ROW format and no autocompression. A 2-byte partitioning is defined. Row header compression and autocompression reduce the size of the table. This example assumes autocompression is the default for column partitions.

```
CREATE TABLE Orders (
  o_orderkey INTEGER NOT NULL,
  o_custkey INTEGER,
  o_orderstatus CHAR(1) CASESPECIFIC,
  o_totalprice DECIMAL(13,2) NOT NULL,
  o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_shippriority INTEGER,
  ROW(o_comment VARCHAR(79)) NO AUTO COMPRESS )
PRIMARY AMP (o_orderkey) PARTITION BY COLUMN,
UNIQUE INDEX (o_orderkey);
```

Example: Primary AMP Index with Column Partitioning and USI (alternate syntax 1)

This example uses a different syntax to define the same table as the previous example. This example assumes autocompression is the default for column partitions.

```
CREATE TABLE Orders (
  o_orderkey INTEGER NOT NULL,
  o_custkey INTEGER,
  o_orderstatus CHAR(1) CASESPECIFIC,
  o_totalprice DECIMAL(13,2) NOT NULL,
  o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_shippriority INTEGER,
  o_comment VARCHAR(79) )
PRIMARY AMP INDEX (o_orderkey)
PARTITION BY COLUMN ALL BUT (ROW(o_comment) NO AUTO COMPRESS),
UNIQUE INDEX (o_orderkey);
```

Example: Primary AMP Index with Column Partitioning and USI (alternate syntax 2)

This example uses a different syntax to define the same table as the previous example. Note that columns not listed in the grouping clause are grouped together into a column partition by default but, since o_shippriority is the only one not listed in the COLUMN grouping clause, each column is in its own single-column partition as in the previous example.

```
CREATE TABLE Orders (
  o_orderkey INTEGER NOT NULL,
  o_custkey INTEGER,
  o_orderstatus CHAR(1) CASESPECIFIC,
```

```

o_totalprice DECIMAL(13,2) NOT NULL,
o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
o_shippriority INTEGER,
o_comment VARCHAR(79) )
PRIMARY AMP (o_orderkey)
PARTITION BY COLUMN (
    o_orderkey, o_custkey, o_orderstatus, o_totalprice,
    o_orderdate, ROW o_comment NO AUTO COMPRESS ),
UNIQUE INDEX (o_orderkey);

```

PARTITION BY

Specifies that a table is partitioned by one or more partitioning levels.

You can define a variety of partition types with a wide range in the number of combined partitions. However, you must consider the usefulness of defining a particular partitioning and its impact, both positive and negative, on performance and disk storage.

You cannot specify a partitioning level that includes a row-level security constraint column.

You cannot specify a character partitioning level for columns or constants that use the Kanji1 or KanjiSJIS server character sets.

You cannot define column partitioning for any of the following table types:

- Global temporary
- Volatile

A partitioned NoPI or PA table must be column partitioned and may also be row partitioned.

A partitioning expression cannot specify external or SQL UDFs or columns having any of the following data types:

- UDT
- ARRAY/VARRAY
- BLOB
- CLOB
- Period
- XML
- Geospatial
- JSON
- DATASET

However, you can reference Period columns indirectly through the use of the BEGIN and END bound functions. See [Example: CASE_N Partitioning Expression Using the END Bound Function](#).

Example: Partitioned NUPI Specification Defined on a CURRENT_DATE Function

This example partitions an insurance customer table into historical (expired) policies and current policies using a CURRENT_DATE function in the partitioning expression.

```
CREATE TABLE customer (
  cust_name          CHARACTER(8),
  policy_number      INTEGER,
  policy_expiration_date DATE FORMAT 'YYYY/MM/DD')
PRIMARY INDEX (cust_name, policy_number)
PARTITION BY CASE_N(policy_expiration_date>= CURRENT_DATE,
                    NO CASE);
```

Example: Specifying a Partitioned NUPI With a USI Defined on the Same Column Set and Partitioned by a RANGE_N Function

This example creates a PPI on o_orderkey and a USI on the same column set. The request bases its partitioning expression on the RANGE_N function on o_orderdate.

The PPI cannot be defined as unique because its partitioning expression is based on o_orderdate and that column is not included in the primary index column set.

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk         CHARACTER(16),
  o_shippriority  INTEGER,
  o_comment       VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_orderdate BETWEEN DATE '1992-01-01'
                    AND DATE '1998-12-31'
                    EACH INTERVAL '1' MONTH)

UNIQUE INDEX (o_orderkey);
```

Example: Partitioned Unique PI Specification Over a Narrow Range Defined on a RANGE_N Function

This example creates a unique PI on store_id , product_id , and sales_date and bases its partitioning expression on the RANGE_N function on sales_date. No secondary indexes are defined.

The example provides partitions that are one day wide over a 5 month interval.

The PI can be defined as unique because the partitioning expression is based on sales_date and that column is included in the primary index column set.

```
CREATE TABLE sales (
  store_id      INTEGER NOT NULL,
  product_id    INTEGER NOT NULL,
  sales_date    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  total_revenue DECIMAL(13,2),
  total_sold    INTEGER,
  note          VARCHAR(256))
UNIQUE PRIMARY INDEX (store_id, product_id, sales_date)
PARTITION BY RANGE_N(sales_date BETWEEN DATE '2001-01-01'
                      AND      DATE '2001-05-31'
                      EACH INTERVAL '1' DAY);
```

Example: Partitioned NUPI Specification Without a USI and Partitioned by a RANGE_N Function

This example creates a nonunique PI on l_orderkey and bases the partitioning expression on the RANGE_N function on l_shipdate. No secondary indexes are defined.

```
CREATE TABLE lineitem (
  l_orderkey      INTEGER NOT NULL,
  l_partkey       INTEGER NOT NULL,
  l_suppkey       INTEGER,
  l_linenumbers   INTEGER,
  l_quantity      INTEGER NOT NULL,
  l_extendedprice DECIMAL(13,2) NOT NULL,
  l_discount      DECIMAL(13,2),
  l_tax           DECIMAL(13,2),
  l_returnflag    CHARACTER(1),
  l_linestatus    CHARACTER(1),
  l_shipdate      DATE FORMAT 'yyyy-mm-dd',
  l_commitdate    DATE FORMAT 'yyyy-mm-dd',
  l_receiptdate   DATE FORMAT 'yyyy-mm-dd',
  l_shipinstruct  VARCHAR(25),
  l_shipmode      VARCHAR(10),
  l_comment       VARCHAR(44))
PRIMARY INDEX (l_orderkey)
PARTITION BY RANGE_N(l_shipdate BETWEEN DATE '1992-01-01'
                      AND      DATE '1998-12-31'
                      EACH INTERVAL '1' MONTH);
```

Example: Partitioned UPI Specification Defined on an EXTRACT Function

This example creates a unique PI on `store_id`, `product_id`, and `sales_date` and bases the partitioning expression on a simple EXTRACT function on `sales_date`. No secondary indexes are defined on `sales_by_month`.

```
CREATE TABLE sales_by_month (
  store_id      INTEGER NOT NULL,
  product_id    INTEGER NOT NULL,
  sales_date    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  total_revenue DECIMAL(13,2),
  total_sold    INTEGER,
  note          VARCHAR(256))
UNIQUE PRIMARY INDEX (store_id, product_id, sales_date)
PARTITION BY EXTRACT(MONTH FROM sales_date);
```

Example: Partitioned UPI Specification Over a Broad Range Defined on a RANGE_N Function

This example provides partitions that are one week wide over a 4 year interval:

```
CREATE TABLE sales_history (
  store_id      INTEGER NOT NULL,
  product_id    INTEGER NOT NULL,
  sales_date    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  total_revenue DECIMAL(13,2),
  total_sold    INTEGER,
  note          VARCHAR(256))
UNIQUE PRIMARY INDEX (store_id, product_id, sales_date)
PARTITION BY RANGE_N(sales_date BETWEEN DATE '1997-01-01'
                     AND      DATE '2000-12-31'
                     EACH INTERVAL '7' DAY);
```

Example: PI Specification Defined with CASE_N Partitioning

The following example creates a NUPI on `store_id`, `product_id`, and `sales_date` and bases the CASE_N partitioning expression on `total_revenue`. Because `total_revenue` is not defined in the primary index column set, the primary index on this table cannot be defined as unique.

```
CREATE TABLE store_revenue (
  store_id      INTEGER NOT NULL,
  product_id    INTEGER NOT NULL,
  sales_date    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  total_revenue DECIMAL(13,2),
  total_sold    INTEGER,
```

```

    note          VARCHAR(256))
PRIMARY INDEX (store_id, product_id, sales_date)
PARTITION BY CASE_N(total_revenue < 10000,
                    total_revenue < 100000,
                    total_revenue < 1000000,
                    NO CASE, UNKNOWN);

```

If you need to ensure the uniqueness of the column set (store_id , product_id , sales_date), you can define a USI on that set.

Example: Illegal Specification of UPI Without All Partitioning Columns Included in Primary Index Definition

The following example attempts to create a unique PI on store_id, product_id, and sales_date, but fails because total_revenue on which the table is partitioned, is not included in the unique PI definition. A PI cannot be unique unless all its partitioning columns are included in the index definition.

```

CREATE TABLE store_revenue (
    store_id      INTEGER NOT NULL,
    product_id    INTEGER NOT NULL,
    sales_date    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    total_revenue DECIMAL(13,2)
    total_sold    INTEGER
    note          VARCHAR(256))
UNIQUE PRIMARY INDEX (store_id, product_id, sales_date)
PARTITION BY CASE_N(total_revenue < 10000,
                    total_revenue < 100000,
                    total_revenue < 1000000,
                    NO CASE, UNKNOWN);

```

Example: Partitioned UPI Specification Using a Simple Arithmetic Expression

The following example creates a unique PPI on store_id , product_id , and sales_date and bases its partitioning expression on a simple arithmetic expression on store_id and product_id. No secondary indexes are defined.

```

CREATE TABLE store_product (
    store_id      INTEGER NOT NULL BETWEEN 0 AND 64,
    product_id    INTEGER NOT NULL BETWEEN 0 and 999,
    sales_date    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    total_revenue DECIMAL(13,2),
    total_sold    INTEGER,
    note          VARCHAR(256))

```



```
UNIQUE PRIMARY INDEX (store_id, product_id, sales_date)
PARTITION BY store_id*1000 + product_id + 1;
```

partitioning_level

partitioning_level

A partitioning level can be defined using a single COLUMN keyword, a partitioning expression, or a combination of both.

Multiple partitioning levels must be separated by commas and the entire set must be enclosed in parentheses.

The maximum number of partitioning levels that you can specify for:

- 2-byte partitioning is 15
- 8-byte partitioning is 62

If a partitioning expression is not a RANGE_N or CASE_N function, it is only allowed if there is only level of partitioning and its result must always be eligible to be implicitly cast to an INTEGER, if it is not already an INTEGER.

COLUMN

Column partition container is stored with COLUMN format.

partitioning_expression

Use a CASE_N function to define a mapping between conditions to INTEGER or BIGINT numbers.

The maximum number of partitions for a CASE_N partitioning level is typically limited to 4,000 or less.

Use a RANGE_N function to define a mapping of ranges of INTEGER, CHARACTER, or DATE values to INTEGER numbers or to define a mapping of ranges of BIGINT or TIMESTAMP values to BIGINT numbers.

The maximum number of ranges, not including the NO RANGE, NO RANGE OR UNKNOWN, and UNKNOWN partitions for a RANGE_N partitioning level, is 9,223,372,036,854,775,805.

You cannot specify an EACH clause if the RANGE_N function specifies a character or graphic test value.

See *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145 for documentation of the CASE_N and RANGE_N functions.

(COLUMN *column_name*)

The column partition is stored with COLUMN format.

Column grouping in the COLUMN clause enables more flexibility in specifying which columns go into which partitions while still being able to specify the display order (that is, when selecting the columns from the table using an asterisk) in the table element list. Column grouping in the column list for a table enables a simpler, but less flexible, specification of column groupings than you can specify in a partitioning level.

(ROW *column_name*)

The column partition is stored with ROW format. A ROW format means that only one column-partition value is stored in a physical row as a subrow.

If you specify neither COLUMN nor ROW for a column partition, Vantage determines whether COLUMN or ROW format is used for the column partition. For information about COLUMN format and ROW format, see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

AUTO COMPRESS

Vantage automatically determines and applies the best available compression method if it can reduce the size of physical rows. AUTO COMPRESS is the default for COLUMN format column partitions.

ADD *constant*

Specifies that the maximum number of partitions for a partitioning level is the number of partitions it defines plus the value of the BIGINT constant value specified by *constant*.

The value for *constant* must be an unsigned BIGINT constant and cannot exceed 9,223,372,036,854,775,807.

ALL BUT (*column_name_list*)

Specifies that a single-column partition with autocompression and a system-determined COLUMN or ROW format is defined for each column, if any, that is not specified in the column group list.

You can only specify this option for column-partitioned tables.

NO AUTO COMPRESS

Disables autocompression for the physical rows of a column partition of a column-partitioned table. Vantage does not automatically determine or apply the best available compression method to physical rows in COLUMN format column partitions.

Vantage does apply any user-specified compression and, for column partitions with COLUMN format, row header compression.

Example: Multilevel Partitioned UPI With no USI Defined on the Primary Index

The following example creates a two-level partitioning using the RANGE_N function as the basis for both partitioning expressions. Because both partitioning columns are components of the primary index, the index can be defined as a UPI.

```
CREATE TABLE sales (
  store_id      INTEGER NOT NULL,
  product_id    INTEGER NOT NULL,
  sales_date    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  total_revenue DECIMAL(13,2),
  total_sold    INTEGER,
  note          VARCHAR(256))
UNIQUE PRIMARY INDEX (store_id, product_id, sales_date)
PARTITION BY (RANGE_N(store_id  BETWEEN 1
                        AND      300
                        EACH      1),
              RANGE_N(sales_date BETWEEN DATE '2006-01-01'
                        AND      DATE '2006-05-31'
                        EACH INTERVAL '1' DAY));
```

Example: Multilevel Partitioned NUPI With no USI Defined on the Primary Index

The following example creates a two-level partitioning using the RANGE_N function as the basis for both partitioning expressions. Because neither partitioning column is a component of the primary index, that index must be defined as a NUPI, and because no USI is defined on o_orderkey, it is not unique.

```
CREATE TABLE lineitem (
  l_orderkey    INTEGER NOT NULL,
  l_partkey     INTEGER NOT NULL,
  l_suppkey     INTEGER,
  l_linenumber  INTEGER,
  l_quantity    INTEGER NOT NULL,
  l_extendedprice DECIMAL(13,2) NOT NULL,
  l_discount    DECIMAL(13,2),
  l_tax         DECIMAL(13,2),
  l_returnflag  CHARACTER(1),
  l_linestatus  CHARACTER(1),
  l_shipdate    DATE FORMAT 'yyyy-mm-dd',
  l_commitdate  DATE FORMAT 'yyyy-mm-dd',
  l_receiptdate DATE FORMAT 'yyyy-mm-dd',
  l_shipinstruct VARCHAR(25),
```

```

    l_shipmode      VARCHAR(10),
    l_comment       VARCHAR(44))
PRIMARY INDEX (l_orderkey)
PARTITION BY (RANGE_N(l_supkey BETWEEN 0
                        AND 4999
                        EACH 10),
              RANGE_N(l_shipdate BETWEEN DATE '2000-01-01'
                        AND DATE '2006-12-31'
                        EACH INTERVAL '1' MONTH));

```

Example: Multilevel Partitioned UPI

The following example creates a two-level partitioning using the RANGE_N function as the basis for both partitioning expressions. Because both partitioning columns are components of the primary index, the index can be defined as a UPI.

```

CREATE TABLE sales_history (
    store_id      INTEGER NOT NULL,
    product_id    INTEGER NOT NULL,
    sales_date    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    total_revenue DECIMAL(13,2),
    total_sold    INTEGER,
    note          VARCHAR(256))
UNIQUE PRIMARY INDEX (store_id, product_id, sales_date)
PARTITION BY (RANGE_N(store_id BETWEEN 1
                        AND 300
                        EACH 1),
              RANGE_N(sales_date BETWEEN DATE '2004-01-01'
                        AND DATE '2006-12-31'
                        EACH INTERVAL '1' MONTH));

```

Example: Multilevel Partitioned NUPI With a Maximum Number of Partitions

The following example specifies the maximum of 65,535 ($3 \times 5 \times 17 \times 257$) partitions allowed for a 2-byte partition number combined partitioning expression. Because none of the partitioning columns is a component of the primary index, that index cannot be defined as a UPI.

```

CREATE TABLE markets (
    product_id    INTEGER NOT NULL,
    region        BYTEINT NOT NULL,
    activity_date DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    revenue_code  BYTEINT NOT NULL,
    business_sector BYTEINT NOT NULL,
    note          VARCHAR(256))

```

```

PRIMARY INDEX (product_id, region) PARTITION BY (
  RANGE_N(region          BETWEEN 1
            AND            9
            EACH          3),
  RANGE_N(business_sector BETWEEN 0
            AND            49
            EACH          10),
  RANGE_N(revenue_code    BETWEEN 1
            AND            34
            EACH          2),
  RANGE_N(activity_date    BETWEEN DATE '1986-01-01'
            AND            DATE '2007-05-31'
            EACH INTERVAL '1' MONTH));

```

Example: Using Different Syntax to Create the Same Column-Partitioned Table Definition or to Change the Partitioning

This example presents three different syntaxes that all define the identical table.

The first example defines a column-partitioned table with a USI on column *o_orderkey*. Each column except for *o_comment* is contained in its own partition, is stored using system-determined COLUMN format, and is autocompressed. Column *o_comment* is in its own partition and is stored using ROW format without autocompression.

Because its maximum combined partition number (including 2 column partitions for internal use and 1 spare) is less than 65,535, the partition number of each row in this table uses a 2-byte partition number field in its row header.

```

CREATE TABLE orders (
  o_orderkey    INTEGER NOT NULL,
  o_custkey     INTEGER,
  o_orderstatus CHARACTER(1) CASESPECIFIC,
  o_totalprice  DECIMAL(13,2) NOT NULL,
  o_orderdate   DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_shippriority INTEGER,
  ROW(o_comment VARCHAR(79)) NO AUTO COMPRESS)
PARTITION BY COLUMN,
UNIQUE INDEX (o_orderkey);

```

The difference for the second example is the absence of the ROW specification for *o_comment* in the column list, replaced by an ALL BUT (ROW(*o_comment*)) specification in the PARTITION BY COLUMN clause.

```

CREATE TABLE orders (
  o_orderkey    INTEGER NOT NULL,

```

```

o_custkey      INTEGER,
o_orderstatus  CHARACTER(1) CASESPECIFIC,
o_totalprice   DECIMAL(13,2) NOT NULL,
o_orderdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
o_shippriority INTEGER,
o_comment      VARCHAR(79))
PARTITION BY COLUMN ALL BUT (ROW(o_comment) NO AUTO COMPRESS),
UNIQUE INDEX (o_orderkey);

```

The difference for the third example is that the columns that are not listed in the grouping clause of the COLUMN clause are grouped together into a column partition by default, but because *o_shippriority* is the only column that is not listed in the COLUMN grouping clause, each column is in its own single-column partition.

```

CREATE TABLE orders (
  o_orderkey    INTEGER NOT NULL,
  o_custkey     INTEGER,
  o_orderstatus CHARACTER(1) CASESPECIFIC,
  o_totalprice   DECIMAL(13,2) NOT NULL,
  o_orderdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_shippriority INTEGER,
  o_comment      VARCHAR(79))
PARTITION BY COLUMN (o_orderkey, o_custkey, o_orderstatus,
                    o_totalprice, o_orderdate,
                    ROW o_comment NO AUTO COMPRESS),
UNIQUE INDEX (o_orderkey);

```

This example again defines a column-partitioned *orders* table, but the partitioning is different from that used in the previous example. Each column is in its own partition with system-determined COLUMN format and autocompression except for the *o_totalprice* and *o_comment* columns, which are grouped together in a partition that has ROW format and autocompression.

Because the maximum number of combined partitions is 65,534 and the maximum column partition number is 65,535, the partition number for each row of this table uses 2-byte partitioning.

You could specify an ADD clause for the partitioning level of this table with a value as large as 65,526 and the syntax would still define the same table.

```

CREATE TABLE orders (
  o_orderkey    INTEGER NOT NULL,
  o_custkey     INTEGER,
  o_orderstatus CHARACTER(1) CASESPECIFIC,
  o_totalprice   DECIMAL(13,2) NOT NULL,
  o_orderdate    DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_shippriority INTEGER,

```

```

    o_comment      VARCHAR(79))
PARTITION BY COLUMN ALL BUT (ROW(o_totalprice, o_comment)),
UNIQUE INDEX (o_orderkey);

```

This example again defines a column-partitioned *orders* table, but the partitioning is different from the previous examples. Each column is in its own partition. This request defines 7 partitions plus the 2 internal partitions used by all column-partitioned tables.

Because the maximum number of column partitions for this table is 9,223,372,036,854,775,806 and the maximum column partition number is 9,223,372,036,854,775,807, each row of this table requires 8-byte partitioning, which also distinguishes this example from the preceding examples of column-partitioned tables.

```

CREATE TABLE orders (
    o_orderkey      INTEGER NOT NULL,
    o_custkey       INTEGER,
    o_orderstatus   CHARACTER(1) CASESPECIFIC,
    o_totalprice    DECIMAL(13,2) NOT NULL,
    o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    o_shippriority  INTEGER,
    o_comment       VARCHAR(79))
NO PRIMARY INDEX PARTITION BY COLUMN ADD 68000,
UNIQUE INDEX (o_orderkey);

```

This example defines the identical *orders* table as the previous example, the only difference being the value specified for the ADD option in the partitioning expression. Each column is in its own partition. This request defines 7 partitions plus the 2 internal partitions used by all column-partitioned tables.

Because the maximum number of column partitions for this table is 9,223,372,036,854,775,806 and the maximum column partition number is 9,223,372,036,854,775,807, the partition number for each row of this table requires 8-byte partitioning, which also distinguishes this example from the preceding examples of column-partitioned tables.

```

CREATE TABLE orders (
    o_orderkey      INTEGER NOT NULL,
    o_custkey       INTEGER,
    o_orderstatus   CHARACTER(1) CASESPECIFIC,
    o_totalprice    DECIMAL(13,2) NOT NULL,
    o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
    o_shippriority  INTEGER,
    o_comment       VARCHAR(79))
PARTITION BY COLUMN ADD 9223372036854775797,
UNIQUE INDEX (o_orderkey);

```

Example: Creating a Column-Partitioned Table With ROW Format

This example defines a column-partitioned table with 3 column partitions, all having ROW format either explicitly specified or system-determined.

```
CREATE MULTISET TABLE sensordata (
  id VARCHAR (128),
  m1 VARBYTE(10000),
  m2 VARBYTE(10000),
  m3 VARBYTE(10000),
  t1 TIMESTAMP(6) WITH TIME ZONE,
  t2 TIMESTAMP(6) WITH TIME ZONE,
  t3 TIMESTAMP(6) WITH TIME ZONE,
  t4 TIMESTAMP(6) WITH TIME ZONE )
PARTITION BY COLUMN (ROW(id), (m1, m2, m3),
                     ROW(ts1, ts2, ts3, ts4));
```

Example: Using the ADD Option for a Column-Partitioned Table

This example defines a maximum of 9,999 column partitions (3 user-specified, 2 for internal use, and 9,994 for adding more partitions) and a maximum column partition number of 10,000.

The maximum combined partition number is the product of the maximum number of partitions at each level. For table *t23*, the maximum combined partition number is calculated as follows.

Maximum combined partition number = 10000 x 10000000 x 92233720 = 9223372000000000000

This is a valid maximum because it does not exceed the limit of 9,223,372,036,854,775,807. Even though there are excess combined partitions, there are not enough of them to be able to add a partition to level 2 or to level 3, so the number of partitions at levels 2 and 3 cannot be increased using an ALTER TABLE request.

```
CREATE MULTISET TABLE t23 (
  a INTEGER,
  b INTEGER,
  c INTEGER)
PARTITION BY (COLUMN ADD 9994,
              RANGE_N(a BETWEEN      1
                      AND 10000000
                      EACH 1),
              RANGE_N(b BETWEEN      1
                      AND 92233720
                      EACH 1));
```

Suppose you change the RANGE_N *end_expression* value for level 3 to 92,233,721, which is only 1 larger than the previous *end_expression* value for level 3 of 92,233,720.


```

CREATE MULTISET TABLE t23a (
  a INTEGER,
  b INTEGER,
  c INTEGER)
PARTITION BY (COLUMN ADD 9994,
              RANGE_N(a BETWEEN      1
                      AND 10000000
                      EACH 1),
              RANGE_N(b BETWEEN      1
                      AND 92233721
                      EACH 1));

```

The table definition for *t23a*, like the definition for *t23*, defines a maximum of 9,999 column partitions (3 user-specified, 2 for internal use, and 9,994 for adding more partitions), and also has a maximum column partition number of 10,000.

But notice what happens now when you calculate the maximum combined partition number for the table.

Maximum combined partition number = 10000 x 10000000 x 92233721 = 9223372100000000000

The maximum combined partition number now exceeds the system maximum value of 9,223,372,036,854,775,807, so Vantage returns an error to the requestor.

Example: Assigning Combined Partitions for a Column-Partitioned Table to Partitioning Levels

If there are remaining unassigned combined partitions after the other partitions have been assigned to their respective levels, Vantage attempts to assign those unassigned combined partitions to the previously defined partition levels, as this example illustrates.

The CREATE TABLE SQL text for this example follows.

```

CREATE TABLE t38 (
  a INTEGER,
  b INTEGER,
  c INTEGER)
PARTITION BY (COLUMN,          -- 3 specified partitions+2 internal=5 defined partitions.
              RANGE_N(c BETWEEN 1
                      AND 10), -- 1 defined partition. Note
              RANGE_N(b BETWEEN 1
                      AND 1000
                      EACH 1) ADD 5);
              -- 1000 partitions.

```

Note the following preliminary properties of the partitioning levels for this table.

- Partitioning Level 1

Table *t38* is defined so that partitioning level 1 has 3 user-specified partitions: 1 each for column a, column b, and column c; plus the 2 internal partitions for a total of 5 defined partitions. By default, the ADD constant for partitioning level 1 is 10.

The maximum number of column partitions for partitioning level 1 is 15, which is derived by adding the initial 5 defined partitions to the default ADD constant of 10.

The maximum column partition number for level 1 is 16. This is derived by adding 1 to the maximum number of column partitions for the level, or $15 + 1 = 16$.

Preliminary number of defined partitions	5
Default ADD constant value	10
Final ADD constant value	10
Default maximum number of column partitions	15
Final maximum number of column partitions	15
Preliminary maximum column partition number	15
Final maximum column partition number	16

- Partitioning Level 2

Because partitioning level 2 is defined using a `RANGE_N` function without specifying an `EACH` clause, its number of defined partitions is only 1. With an initial maximum number of column partitions for level 2 being only 1 because it does not specify an ADD constant. However, the maximum partitioning number for a partitioning level must be at least 2, so the actual maximum number of defined partitions for level 2 is 2, as is the maximum number of column partitions for the level.

Preliminary number of defined partitions	1
Default ADD constant value	1
Final ADD constant value	3
Default maximum number of defined column partitions	2
Final maximum number of defined column partitions	4
Preliminary maximum column partition number	4
Final maximum column partition number	4

- Partitioning Level 3

Level 3 has 1,000 defined partitions and a preliminary maximum partition number of 1005. This is derived by adding the ADD constant value, 5, to the initially defined 1,000 partitions.

Preliminary number of defined partitions	1,000
Default ADD constant value	5
Final ADD constant value	23
Default maximum number of defined column partitions	1,005
Final maximum number of defined column partitions	1,023
Preliminary maximum column partition number	1,005
Final maximum column partition number	1,023

Because the product of the default maximum column partition numbers for each level, $16 \times 2 \times 1005 = 32,160$, this table defines a 2-byte partitioning because $32,160 \leq 65,335$.

DefinedCombinedPartitions	= product of defined partitions at each level	= $5 \times 1 \times 1,000 = 5,000$
MaxCombinedPartitions	= product of maximum partitions at each level	= $15 \times 4 \times 1,023 = 61,380$
Maximum Combined Partition Number	= product of maximum partition number at each level	= 65,472

While Maximum Combined Partition Number $\leq 65,535$, increasing the maximum partition number for any of the partitioning levels by only 1 would cause the value of Maximum Combined Partition Number $> 65,535$, which would not be valid.

The following SELECT request reports the values for table *t38* in DBC.TableConstraints after Vantage has made its adjustments to the number of combined partitions assigned to each partitioning level of *t38*.

```
SELECT *
FROM DBC.TableConstraints
WHERE TVMId IN (SELECT TVMId
                FROM DBC.TVM
                WHERE TVMNameI = 't38'
                AND   DataBaseId IN (SELECT DataBaseId
                                     FROM DBC.DBASE
                                     WHERE DatabaseNameI = 'test'))
ORDER BY TVMId;
```

This request returns the following result set.

TVMId	000017070000
Name	?

DBaseId	0000F303
TableCheck	CHECK (/*03 02 01*/ PARTITION#L1 /*1 5+10*/ =1 AND RANGE_N(c BETWEEN 1 AND 10) /*2 1+3*/ IS NOT NULL AND RANGE_N(b BETWEEN 1 AND 1000 EACH 1) /*3 1000+23*/ IS NOT NULL)
CreateUID	00000100
CreateTimeStamp	2010-12-12 18:16:38
LastAccessTimeStamp	?
AccessCount	?
ConstraintType	Q
IndexNumber	1
ConstraintCollation	U
CollName	?
CharSetID	?
SessionMode	?
VTCheckType	?
TTCheckType	?
ResolvedCurrent_Date	?
ResolvedCurrent_TimeStamp	?
DefinedCombinedPartitions	5000
MaxCombinedPartitions	61380
PartitioningLevels	3
ColumnPartitioningLevel	1

UNIQUE INDEX

Define a unique secondary index (USI) for the table.

index_name

Optional name for the index.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

index_column_name

Column set whose values are to be used as the basis for a secondary index.

Columns in the list cannot have any of the following data types: BLOB, CLOB, Period, XML, Geospatial, JSON, or DATASET.

If you specify more than one column name, the index is created on the combined values of each column named. A maximum of 64 columns can be specified for an index, and a maximum of 32 secondary indexes can be created for one table.

WITH LOAD IDENTITY

Records the RowLoadID for the index created on a load isolated table.

You can specify this option for indexes created on load isolated tables.

NO

Does not record RowLoadID with the ROWIDs in a NUSI. The base row, using the qualified ROWID from the index row must be used to determine the committed property. This index may be simpler to maintain but read operations are more expensive than the regular NUSI based option, limiting the usability of index.

Example: Multilevel-Partitioned NUPI With a USI Defined on the Primary Index

The following example creates a two-level PPI table using the RANGE_N function as the basis for both partitioning expressions. Because neither partitioning column is a component of the primary index, that index must be defined as a NUPI. To enforce uniqueness, a USI is defined on o_orderkey.

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk         CHARACTER(16),
  o_shippriority  INTEGER,
  o_comment       VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY (RANGE_N(o_custkey  BETWEEN 0
                        AND 49999
                        EACH 100),
              RANGE_N(o_orderdate BETWEEN DATE '2000-01-01'
                        AND      DATE '2006-12-31'
```

```

                                EACH INTERVAL '1' MONTH))
UNIQUE INDEX (o_orderkey);

```

ORDER BY

Row ordering on each AMP by a single NUSI column by value or hash.

If you specify ORDER BY, but do not specify VALUES or HASH, the ordering defaults to VALUES.

If you do not specify ORDER BY, the system orders the rows on their hash values.

If you specify HASH, the system orders the rows on their hash values.

In an ORDER BY clause, you cannot specify a column with an XML, Geospatial, JSON, or DATASET data type.

VALUES

Value-ordering for the ORDER BY NUSI column.

Select VALUES to optimize queries that return a contiguous range of values, especially for a covering index or a nested join.

You can specify VALUES with or without an order column name. If you do not specify *order_column_name*, the system orders the NUSI on the values of its first column.

HASH

Hash-ordering for the ORDER BY column.

Select HASH to limit hash-ordering to one column, rather than all columns. This is the default if you do not specify an ORDER BY clause.

Hash-ordering a multicolumn NUSI on one of its columns allows the NUSI to participate in a nested join where join conditions involve only that ordering column.

You can specify HASH with or without an order column name. If you do not specify *order_column_name*, the system orders the NUSI on the hash of its first column.

order_column_name

An optional column in the INDEX column list for a NUSI that specifies the sort order used to store index rows.

If you do not specify an *order_column_name*, the system orders the NUSI rows using the first column in the index definition by default.

A value-ordered *order_column_name* can have any of the following data types:

- BYTEINT
- DATE
- DECIMAL

- INTEGER
- SMALLINT

Values for *order_column_name* are limited to 4 or fewer bytes.

WITH LOAD IDENTITY

Records the RowLoadID for the index created on a load isolated table.

You can specify this option for indexes created on load isolated tables.

NO

Does not record RowLoadID with the ROWIDs in a NUSI. The base row, using the qualified ROWID from the index row must be used to determine the committed property. This index may be simpler to maintain but read operations are more expensive than the regular NUSI based option, limiting the usability of index.

table_preservation

Action to perform when a transaction completes.

ON COMMIT

Delete or preserve the contents of an instance of a global temporary or volatile table when a transaction completes.

This option is valid for global temporary and volatile tables only.

DELETE ROWS

Clears an instance of a global temporary or volatile table of all rows.

DELETE is the default.

PRESERVE ROWS

Retains the rows in the global temporary or volatile table after the transaction commits.

AS_clause

This clause appears only in the CREATE TABLE AS statement.

You cannot specify data types for source table columns, but you can specify column attributes for them.

You can also specify table constraints for the target table to be created.

You cannot specify *normalize_option* for a target table.

source_table

column_name

A column or list of columns from *source_table_name* to be copied to target table *table_name*.

If you create a target table from a normalized source table without specifying any new columns, Vantage creates the target table with the same NORMALIZE definition as the source table.

Note:

You cannot specify a column data type using the AS clause syntax.

column attributes

One or more new column attributes for the column set specified by *column_name* to be used for the new definition for *table_name*.

Note:

You cannot specify a foreign key constraint as a column attribute using the AS clause syntax.

table constraints

One or more new table constraints for the column set specified by *column_name* to be used for the new definition for *table_name*.

Note:

You cannot specify a foreign key constraint as a table constraint using the AS clause syntax.

database_name

user_name

The containing database or user for *source_table_name* if not the current database or user.

source_table_name

The name of the source table whose column- and table-level definitions (with some restrictions) and, optionally, data and statistics are to be copied to *table_name*.

You must specify *source_table_name* whether you use the AS *source_table_name* WITH DATA form or the AS SELECT FROM *source_table_name* subquery form.

If the source table has derived period columns, the target table will also have those derived period columns.

You cannot use the CREATE TABLE ... AS syntax, with a subquery that defines the source table, to create a table that is protected by row-level security constraints.

subquery_clause

An optional clause for selecting a subset of the column definitions or all of the column definitions from *source_table_name*.

The subquery form is only valid if you also specify a WITH DATA clause.

You cannot use the subquery form of CREATE TABLE AS to copy the NORMALIZE definition from a source table to a target table.

You cannot specify the NORMALIZE option or the EXPAND ON clause in the SELECT subquery where you specify NO PRIMARY INDEX for the target table.

SELECT *column_name* FROM *source_table_name*

Select a subset of the column definitions from *source_table_name*.

SELECT * FROM *source_table_name*

Select all of the column definitions from *source_table_name*.

WITH DATA

The data for the source table or subquery is to be copied to a new target table. You cannot create global temporary tables using the WITH DATA option.

Indicate column subsets using a subquery. Otherwise, the operation copies all column definitions and data to the target table.

Example: CREATE TABLE AS ... WITH DATA with Column Constraints

This CREATE TABLE AS ... WITH DATA request specifies column constraints. Note the absence of referential integrity constraints, which are not valid for AS clauses.

The request also copies the data from the selected columns into *target_table*.

Because the request specifies a subquery and no explicit table kind is specified, the table kind of *target_table* defaults to the session mode default, not to the table kind of *subquery_table*.

```
CREATE TABLE target_table (
  column_1 NOT NULL DEFAULT 0,
```

```

    column_2)
  AS (SELECT column_x, column_y
      FROM subquery_table )
  WITH DATA;

```

Example: CREATE TABLE AS ... WITH DATA with Explicit Column Names

This CREATE TABLE AS ... WITH DATA request specifies column names in *target_table* that differ from their names in *source_table*. The request also copies the data from the selected columns into *target_table*.

Because the request specifies a subquery and no explicit table kind is specified, the table kind of *target_table* defaults to the session mode default, not to the table kind of *subquery_table*. See the AS clause of CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184).

```

CREATE TABLE target_table (
  column_x,
  column_y)
AS (SELECT column_1, column_2
    FROM subquery_table )
WITH DATA;

```

Example: CREATE TABLE AS ... WITH DATA with Unnamed Expressions

This CREATE TABLE AS ... WITH DATA request creates *target_table* from a subquery having unnamed expressions that define both columns for *target_table*. The affected columns are named explicitly in the *target_table* definition to compensate. The request also copies the data from the selected columns into *target_table*.

Because the request specifies a subquery and no explicit table kind is specified, the table kind of *target_table* defaults to the session mode default, not to the table kind of *subquery_table*.

```

CREATE TABLE target_table (
  column_x,
  column_y)
AS (SELECT column_x + 1, column_y + 1
    FROM subquery_table )
WITH DATA;

```

Example: CREATE TABLE AS ... WITH DATA with Unnamed Expression

This CREATE TABLE AS ... WITH DATA request creates *target_table* from a subquery having an unnamed aggregate expression. Note that the affected column is named explicitly in the *target_table* definition to compensate. The request also copies the data from the selected columns into *target_table*.

Because the request specifies a subquery and no explicit table kind is specified, the table kind of *target_table* defaults to the session mode default, not to the table kind of *subquery_table*.

```
CREATE TABLE target_table (
    column_x,
    column_y)
AS (SELECT MAX(column_x), column_y
    FROM subquery_table
    GROUP BY 2 )
WITH DATA;
```

Example: AS WITH DATA Where Column Attributes Differ between Source and Target Table Definitions

This example shows that when you specify explicit column descriptors that are different from those defined in the source table, the system does not copy the old column attributes, but instead defines the new table with the attributes specified in the target table definition.

Suppose you create the following simple table.

```
CREATE TABLE source (
    x INTEGER NOT NULL,
    y INTEGER DEFAULT 0);
```

You then decide to create a new table with data based on the definition of source, so you submit a CREATE TABLE AS ... WITH DATA request like the following.

```
CREATE TABLE target (
    a,
    b NOT NULL) AS (SELECT *
    FROM source)
WITH DATA;
```

You then submit a SHOW TABLE request on target to see what its column descriptors are:

```
SHOW TABLE target;
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
-----
CREATE SET TABLE user_name.target ,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT
    (
        a INTEGER,
```

```

        b INTEGER NOT NULL)
PRIMARY INDEX ( a );

```

The NOT NULL attribute is not copied from column `source.x` to column `target.a`, and the DEFAULT 0 attribute is not copied from column `source.y` to column `target.b`. Instead, `target.a` has no column attribute because none is specified for it in its table definition, and `target.b` has the column attribute NOT NULL, as specified in the table definition for `target.b`, rather than the attribute DEFAULT 0 as is specified for `source.y`.

Example: CREATE VOLATILE TABLE AS ... WITH DATA

This CREATE VOLATILE TABLE AS ... WITH DATA request creates a new volatile table using all the column definitions and data from `source_table`.

The table kind of *target_table* defaults to the table kind of `source_table`. See CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

```

CREATE VOLATILE TABLE target_table AS source_table
WITH DATA;

```

WITH NO DATA

None of the data from the source table or query expression are to be copied to a new table based on its definition.

All (or a subset you define) of the table definitions for *source_table_name*, but none of its data, are to be copied to a new table.

Indicate column subsets using a subquery. Otherwise, the operation copies all column definitions to the target table.

Example: CREATE TABLE AS ... WITH NO DATA with Explicit Column Names

This CREATE TABLE AS ... WITH NO DATA request specifies column names for *target_table* as column alias names in the subquery.

Because the request specifies a subquery and no explicit table kind is specified, the table kind of *target_table* defaults to the session mode default, not to the table kind of *subquery_table*.

```

CREATE TABLE target_table AS (
  SELECT column_x AS column_1, column_y AS column_2
  FROM subquery_table )
WITH NO DATA;

```

Example: CREATE TABLE AS ... WITH NO DATA with Explicit Column Attributes

This CREATE TABLE AS ... WITH NO DATA request specifies column attributes and uses an expression to define a column in *target_table*.

Because the request specifies a subquery and no explicit table kind is specified, the table kind of *target_table* defaults to the session mode default, not to the table kind of *subquery_table*.

```
CREATE TABLE target_table (
    column_1 TITLE 'max_x',
    column_2)
AS (SELECT MAX(column_x), column_y
    FROM subquery_table
    GROUP BY 2 )
WITH NO DATA;
```

Example: CREATE TABLE AS ... WITH NO DATA with Explicit Column Attributes

This example creates the same *target_table* as above, but codes it differently.

Because the request specifies a subquery and does not specify an explicit table kind, the table kind of *target_table* defaults to the session mode default, not to the table kind of *subquery_table*.

```
CREATE TABLE target_table AS (SELECT MAX(column_x)
                                (TITLE 'max_x' NAMED column_1),
                                column_y AS column_2
    FROM subquery_table
    GROUP BY 2)
WITH NO DATA;
```

Example: CREATE GLOBAL TEMPORARY TABLE AS ... WITH NO DATA

This CREATE GLOBAL TEMPORARY TABLE AS ... WITH NO DATA request creates a new global temporary table using all the selected column definitions with default definitions for all table options clause attributes.

Because the request specifies a subquery and no explicit table kind is specified, the table kind of *target_table* defaults to the session mode default, not to the table kind of *subquery_table*.

```
CREATE GLOBAL TEMPORARY TABLE target_table AS (SELECT *
                                                    FROM subquery_table)
WITH NO DATA;
```

AND [NO] STATISTICS

The statistics for the source table be copied to a new table based on its definition.

The system also sets up the appropriate statistical histograms in the dictionary for the target table and copies the statistics from the source table into them (see *Teradata Vantage™ - SQL Request and Transaction Processing*).

When you copy PARTITION statistics, the statistics copied to the target table might not correctly represent the data in the target table because of differences in internal partition number mapping between the source and target tables. This is true even if the table definitions returned by a SHOW TABLE request are identical and the data is the same in both tables.

If you use a CREATE TABLE AS ... WITH DATA AND STATISTICS request to create a target table, the PARTITION statistics you copy from the source table are not valid if the internal partition numbers in the target table are different than the source table.

It is critical to understand that there is no way to guarantee that a target table created using a CREATE TABLE AS ... WITH DATA AND STATISTICS request is identical to the source table from which its statistics are copied down to the level of internal partition numbers, and it is important to understand that even though the two tables might appear to be identical from comparing their definitions using the output of SHOW TABLE requests on the tables.

As a general rule, you should always recollect the PARTITION statistics for the target table when you copy them from a source table.

If you specify WITH NO DATA with the AND STATISTICS option, Vantage sets up the appropriate statistical histograms for the target table, but does not populate them with the available statistical information. This state is referred to as zeroed statistics. It is important to collect statistics once the table is populated.

Example: Copying Statistics

Vantage copies multicolumn or index statistics even if the relative order of the columns specified in the select list of the subquery is not the same as is defined for the source table.

When you copy PARTITION statistics, the statistics copied to the target table might not correctly represent the data in the target table because of differences in internal partition number mapping between the source and target tables, even if the table definitions returned by a SHOW TABLE request are identical and the data is the same in both tables.

Note:

You cannot assume a target table created using a CREATE TABLE AS ... WITH DATA AND STATISTICS request will be identical to the source table from which its statistics are copied down to the level of internal partition numbers, even though the two tables might appear to be identical from comparing their definitions using the output of SHOW TABLE requests on the tables. In general, you should always recollect the PARTITION statistics for the target table when you copy them from a source table.

Example 1: Copying Single-Column Statistics

In this example, Vantage copies single-column statistics on columns x and a as single-column statistics for columns a and c, respectively. Multicolumn statistics on (a, b) are not copied as multicolumn statistics for (b, c) because the order of columns b and c in the target table is different from the order of columns b and a in the source table.

```
CREATE TABLE t2 (a, b, c) AS (SELECT x AS colA, b, a (AS colC)
                               FROM test)

WITH DATA AND STATISTICS;
```

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
single column	x	a	Column x in source table test maps directly to column a in target table t2.
single column	y	none	Source table test column y does not have an analog in target table t2.
single-column NUPI	a	c	Column a in source table test maps directly to column c in target table t2.
multicolumn	(x,y,z)	none	Source table test column y does not have an analog in target table t2.
multicolumn	(a,b)	none	Order of columns (b,c) in target table t2 is different than the order of columns (a,b) in source table test.
multicolumn NUSI	(x,y)	none	<ul style="list-style-type: none"> Source table test column y does not have an analog in target table t2. A NUSI is not defined for target table t2.

Example 2: Copying Single-Column Statistics in Relative Order

In this example, the system copies single-column statistics on k1, y1, and z1 as single-column statistics for a, b, and c, respectively. The system does not copy index statistics on (x1, y1) as index statistics for

(a, b) because the order of a and b in the target table t2 is not the same as the order of x1 and y1 in the source table test1.

```
CREATE TABLE t2 (a, b, c) AS (SELECT y1 AS colA, x1 AS colB,
                                z1 AS colC)
                                FROM test1)

WITH DATA AND STATISTICS
INDEX(a, b);
```

Vantage copies these statistics ...	From this source table column set ...	To this target table column set ...	Because ...
single column	x1	b	columns x1 and y1 in source table test1 map directly to target table t2 columns b and a, respectively.
	y1	a	
	k1	none	there is no analog of source table test1 column k1 in target table t2.
single-column NUSI	z1	c	source table test1 column z1 maps directly to target table t2 column c.
multicolumn NUSI	none	(a,b)	the order of columns (a,b) in target table t2 is different than the order of columns (x1,y1) in source table test1. Note that their ordering in t2, if they had retained the same column names, would be (y1, x1).
multicolumn NUPI	(x1,y1)	none	

Example: Copying Statistics for All Columns

Vantage copies statistics for all of the following scenarios because the underlying index and column data is not modified when it is copied as these examples demonstrate.

Example 1

In this example, Vantage copies the statistics for columns for x1 and y1 from source table test1 to columns a and b in target table t6. The system copies the statistics on the composite index on the source table column set (x1, y1) as multicolumn statistics for the column set (a, b) in the target table t6 because no index is explicitly defined on that column set in t6.

```
CREATE TABLE t6 (a, b, c) AS (SELECT x1, y1, z1 (AS colC)
                                FROM test1)

WITH DATA AND STATISTICS;
```


The shaded cells in the table indicate where the system copies statistics when they do not correspond directly to a column set in the source table.

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
single column	x1	a	Columns x1 and y1 in source table test1 map directly to columns a and b in target table t6.
	y1	b	
single-column NUSI	z1	c	Column z1 in source table test1 maps directly to column c in target table t6.
multicolumn	none	(a, b)	Columns x1 and y1 in source table test1 map directly, and in the same order, to columns a and b in target table t6, so the multicolumn statistics on (x1,y1) in test1 can be used as multicolumn statistics on (a,b) in t6.
multicolumn NUPI	(x1, y1)	none	

Example 2

In this example, Vantage copies the following column and index statistics.

```
CREATE TABLE t6 (a, b, c) AS (SELECT x AS colA, b, a (AS colC)
                                FROM test)
WITH DATA AND STATISTICS;
```

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
single column	y	None.	Column y does not have an analog in table t6.
	x	a	Column a in table t6 is a direct analog of column x in table test.
	a	c	The index statistics collected on NUPI column a in table test can be used as column statistics for column c in table t6.
single-column NUPI	a	None. Statistics for index column a in test are copied to non-index column c in t6.	A NUPI is not defined for target table t6. However, the statistics collected for the NUPI on column a in source table test can be used as column statistics for column c in table t6.
multicolumn	(a, b)	None.	The analogs of columns a and b from source table test are defined in a different order in target

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
			table t6. Therefore, their multicolumn statistics cannot be used for the new column set.
multicolumn NUSI	(x,y)	None.	Column y does not have an analog in table t6.

Example 3

In this example, Vantage copies the following statistics.

```
CREATE TABLE t6 (a, b, c) AS (SELECT k1 AS colA, y1 AS colB,
                                z1 AS colC)
                                FROM test1)
WITH DATA AND STATISTICS;
```

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
single column	k1	a	Columns a and b in target table t6 are direct analogs of columns k1 and y1 in source table test1. The single-column statistics for table test1 columns k1, y1, and z1 are copied to t6 for the single columns a, b, and c, respectively.
	y1	b	
single-column NUSI	z1	c	A NUSI is not defined for target table t6. However, the statistics collected for the index on column z1 in source table test1 map directly to column c in target table test1.
multicolumn NUPI	(x1,y1)	none	Column x1 does not have an analog in source table test1 in target table t6.

Example 4

In this example, Vantage copies single column statistics on x1 and y1 for a and b, and it copies index statistics for the column set (a, b).

```
CREATE TABLE t6 (a, b) AS (SELECT x1 AS colB, y1 (AS colC)
                                FROM test1)
WITH DATA AND STATISTICS
PRIMARY INDEX (a, b);
```

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
single column	x1	a	Columns a and b in target table t6 are analogs of columns x1 and y1 in source table test1.
	y1	b	
single-column NUSI	z1	None.	Source table test1 column z1 does not have an analog in target table t6.
multicolumn NUPI	(x1, y1)	(a, b)	Columns a and b in target table t6 are analogs of columns x1 and y1 in source table test1 and are also defined in the same order in t6 as they were in test1.

Example 5

In this example, the system copies statistics from source table test to target table t6 only for column x.

```
CREATE TABLE t6 AS (SELECT *
                     FROM (SELECT x
                           FROM test) AS d1)
WITH DATA AND STATISTICS;
```

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
single column	x	d1	Column x from source table test is the only column that is qualified by the nested subqueries.
single-column NUPI	a	None.	Source table column a does not have an analog in target table t6.
multicolumn	<ul style="list-style-type: none"> • (a,b) • (x,y,z) 	None.	The only column in target table t6 is an analog of column x in source table test.
multicolumn NUSI	(x, y)	None.	The only column in target table t6 is an analog of column x in source table test.

Example 6

In this example, the system copies all the available single column statistics, multicolumn statistics, and index statistics from table test1 to table t6.

```
CREATE TABLE t6 AS test1
WITH DATA AND STATISTICS;
```

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
single column	x1	x1	All columns from source table test1 and their statistics are copied to the same columns (with the same names) in target table t6.
single column	y1	y1	All columns from source table test1 and their statistics are copied to the same columns (with the same names) in target table t6.
single column	k1	k1	All columns from source table test1 and their statistics are copied to the same columns (with the same names) in target table t6.
single-column NUSI	z1	z1	All columns from source table test1 and their statistics are copied to the same columns (with the same names) in target table t6.
multicolumn NUPI	(x1, y1)	(x1, y1)	All columns from source table test1 and their statistics are copied to the same columns (with the same names) in target table t6.

Example 7

In this example, the system copies all the available single column statistics, multicolumn statistics, and index statistics from table test to table t6.

The statistics that had been collected for the column set (x, y, z) in test are now used for the new multicolumn NUPI defined on the same column set in table t6.

The multicolumn statistics defined on (x, y, z) for test are copied to t6 for the same column set, which is defined as the multicolumn NUPI for that table.

```
CREATE TABLE t6 AS test
WITH DATA AND STATISTICS
PRIMARY INDEX (x, y, z);
```

The shaded cells in the table indicate where the system copies statistics when they do not correspond directly to a column set in the source table.

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
single column	x	x	Columns x and y from source table test are analogs, having the same column names in target table t6.

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
single column	y	y	Columns x and y from source table test are analogs, having the same column names in target table t6.
single column	none	a	The system copies NUPI column a from source table test as non-index column a to target table t6. Its single-column index statistics from source table test are copied as single-column non-index statistics to target table t6.
single-column NUPI	a	None.	The system copies NUPI column a from source table test as non-index column a to target table t6. Its single-column index statistics from source table test are copied as single-column non-index statistics to target table t6.
multicolumn	(a, b)	(a, b)	the source and target table columns are analogous and defined in the same order in both tables.
multicolumn	(a, b)	(x,y)	The source and target table columns are analogous and defined in the same order in both tables. The system copies the multicolumn index statistics from source table test as multicolumn statistics to target table t6.
multicolumn NUSI	(x, y)		The source and target table columns are analogous and defined in the same order in both tables. The system copies the multicolumn index statistics from source table test as multicolumn statistics to target table t6.
multicolumn	(x,y,z)		The source and target table columns are analogous and defined in the same order in both tables. The system copies the multicolumn statistics from source table test as multicolumn index statistics to target table t6.
multicolumn NUPI		(x, y, z)	The source and target table columns are analogous and defined in the same order in both tables. The system copies the multicolumn statistics from source table test as multicolumn index statistics to target table t6.

Example 8

In this example, the system copies all the available single-column statistics, multicolumn statistics, and index statistics for table test.

```
CREATE TABLE t6 AS test
WITH DATA AND STATISTICS
PRIMARY INDEX (x, y, z)
INDEX (a, b);
```

The shaded cells in the table indicate where the system copies statistics when they do not correspond directly to a column set in the source table.

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
single column	x	x	Columns x and y from source table test are analogs and have the same column names in target table t6.
single column	y	y	columns x and y from source table test are analogs and have the same column names in target table t6.
single column		a	the system copies NUPI column a from source table test as non-index column a to target table t6. Its single-column index statistics from source table test are copied as single-column non-index statistics to target table t6.
single-column NUPI	a		the system copies NUPI column a from source table test as non-index column a to target table t6. Its single-column index statistics from source table test are copied as single-column non-index statistics to target table t6.
multicolumn	(x, y, z)		the multicolumn statistics on column set (x, y, z) in source table test are copied as multicolumn index statistics for the multicolumn NUPI on the same column set in target table t6.
multicolumn	(a,b)		The multicolumn statistics on column set (a,b) in source table test are copied as multicolumn index statistics for the multicolumn NUSI on the same column set in target table t6.
multicolumn		(x,y)	The multicolumn NUSI statistics on column set (x,y) in source table test are copied as multicolumn non-index statistics on the same column set in target table t6.
multicolumn NUPI		(x, y, z)	the multicolumn statistics on column set (x, y, z) in source table test are copied as multicolumn index statistics for the multicolumn NUPI on the same column set in target table t6.
multicolumn NUSI	(x, y)	(a, b)	The multicolumn NUSI statistics on column set (x,y) in source table test are copied as multicolumn non-index statistics on the same column set in target table t6.

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
			The multicolumn statistics on column set (a,b) in source table test are copied as multicolumn index statistics for the multicolumn NUSI on the same column set in target table t6.

The multicolumn statistics (x, y, z) that had been collected in test are now used for the new multicolumn NUPI defined on the same column set in t6, while the multicolumn statistics on (a, b) that had been collected in test are now used for the new multicolumn NUSI defined on the same column set in t6.

Example 9

In this example, the system copies all the single-column statistics, multicolumn statistics, and the index statistics for the multicolumn NUSI (x, y).

```
CREATE TABLE t6 AS (SELECT *
                     FROM test)
WITH DATA AND STATISTICS
PRIMARY INDEX(x, y);
```

The shaded cells in the table indicate where the system copies statistics when they do not correspond directly to a column set in the source table.

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
single column	x	x	Column x from source table test is an analog and has the same column name in target table t6.
single column	y	y	Column y from source table test is an analog and has the same column name in target table t6.
single column		a	The system copies its single-column index statistics from source table test as single-column non-index statistics to target table t6.
single-column NUPI	a		The system copies NUPI column a from source table test as non-index column a to target table t6.
multicolumn	(x, y, z)	(x, y, z)	The system copies the multicolumn statistics on both column sets directly from source table test to target table t6 because the columns are defined identically for both.
multicolumn	(a,b)	(a,b)	The system copies the multicolumn statistics on both column sets directly from source table test to target

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
			table t6 because the columns are defined identically for both.
multicolumn NUPI		(x, y)	The system copies multicolumn index statistics on the composite NUSI column set (x,y) of source table test as multicolumn index statistics on the composite NUPI on the same columns in target table t6.
multicolumn NUSI	(x, y)		The system copies multicolumn index statistics on the composite NUSI column set (x,y) of source table test as multicolumn index statistics on the composite NUPI on the same columns in target table t6.

The statistics that had been collected for the single columns x and y and for the column sets (x, y, z) and (a, b) in test are copied straight across into the same columns sets for t6, while the statistics that had been collected for the multicolumn NUSI on (x, y) in test are now used for the new composite NUPI defined on the same column set in t6.

The multicolumn NUSI that was defined for test is not defined for the new table t6, but the system does copy its statistics for use with the new multicolumn NUPI defined on the same column set in t6.

Example 10

In this example, the system copies all the single-column statistics, multicolumn statistics, and index statistics from test to t6.

```
CREATE TABLE t6 AS test
WITH DATA AND STATISTICS;
```

Statistics Copied	Source Table Column Set	Target Table Column Set	Description
single column	x	x	The source and target tables are identical to one another.
	y	y	
single-column NUPI	a	a	
multicolumn	(x, y, z)	(x, y, z)	
	(a,b)	(a,b)	
multicolumn NUSI	(x, y)	(x, y)	

Example: Copying Zeroed Statistics

The examples in this set copy zeroed statistics. The examples are based on the following source table definitions for *test* and *test1*.

```
CREATE SET TABLE test, NO FALLBACK,  
NO BEFORE JOURNAL,  
NO AFTER JOURNAL,  
CHECKSUM = DEFAULT (  
  x INTEGER,  
  y INTEGER,  
  z CHARACTER(30),  
  a INTEGER,  
  b DATE,  
  e INTEGER)  
PRIMARY INDEX (a),  
INDEX(x, y);
```

Assume you collect the following statistics on test.

Statistics	Test Table Column Set
single column	<ul style="list-style-type: none">• x• y
single-column NUPI	a
multicolumn	<ul style="list-style-type: none">• (x, y, z)• (a, b)
multicolumn NUSI	(x, y)

```
CREATE SET TABLE test1, NO FALLBACK,  
NO BEFORE JOURNAL,  
NO AFTER JOURNAL,  
CHECKSUM = DEFAULT (  
  x1 INTEGER,  
  y1 INTEGER,  
  z1 CHARACTER(30),  
  a1 INTEGER,  
  k1 DECIMAL,  
  b1 DATE)  
PRIMARY INDEX (x1, y1),  
INDEX (z1);
```

Assume the following statistics have been collected on test1.

Statistics	Test1 Table Column Set
single column	<ul style="list-style-type: none">• x1• y1• k1
single-column NUSI	z1
multicolumn NUPI	(x1, y1)

Example1

In this example, the system copies zeroed statistics for the following column and index sets.

```
CREATE GLOBAL TEMPORARY TABLE t2 AS (SELECT *
                                         FROM test1)
WITH NO DATA AND STATISTICS;
```

The system copies these zeroed statistics ...	From this source table column set ...	To this target table column set ...	Because ...
single column	x1	x1	all the columns defined for target table t2 are exact analogs of all columns defined for source table test1.
	y1	y1	
	k1	k1	
	none	z1	
single-column NUSI	z1	none	
multicolumn NUPI	(x1, y1)	(x1, y1)	

Example 2

In this example, the system copies zeroed statistics for the following column and index sets.

```
CREATE GLOBAL TEMPORARY TABLE t2 AS (SELECT *
                                         FROM test)
WITH NO DATA AND STATISTICS;
```

System copies these zeroed statistics	From this source table column set	To this target table column set	Description
single column	x	x	all the columns defined for target table t2 are analogs of all columns defined for source table test except that the NUSI on (x,y) is not defined as an index on t2 , so its index statistics are copied as multicolumn statistics for t2.
	y	y	
single-column NUSI	a	a	
multicolumn	(x, y, z)	(x, y, z)	
	(a, b)	(a, b)	
	none	(x, y)	
multicolumn NUSI	(x, y)	none	

Example 3: Copying Zeroed Statistics for a Table

In this example, Vantage copies zeroed statistics for the source table *t1* to the target table *test*.

```
CREATE TABLE t1 AS test
WITH NO DATA AND STATISTICS;
```

Vantage copies these zeroed statistics ...	From this source table column set ...	To this target table column set ...	Because ...
single column	x	x	all the columns defined for target table t1 are exact analogs of all columns defined for source table test.
	y	y	
single-column NUSI	a	a	
multicolumn	• (a, b) • (x, y, z)	• (a, b) • (x, y, z)	
composite NUSI	(x, y)	(x, y)	

Example4

In this example, Vantage copies no statistics because of the join condition on test and test1 specified in the subquery:

```
CREATE TABLE t1 AS (SELECT *
                     FROM test, test1)
WITH NO DATA AND STATISTICS;
```

Example: Copying Zeroed Multicolumn and Composite Index Statistics

Vantage copies multicolumn zeroed statistics and zeroed composite index statistics even if the relative order of the index columns specified in the select list of the subquery is not the same as their order in the source table.

Example 1: Copying Single-Column Statistics

In this example, Vantage copies the single-column statistics on columns x and a in test as zeroed single-column statistics for columns a and c in t2. The multicolumn statistics on (a,b) are not copied as zeroed multicolumn statistics on columns (b,c) in t2 because the order of columns b and c in t2 is different than the order of columns b and a in test.

```
CREATE TABLE t2 (a, b, c) AS (SELECT x AS colA, b, a (AS colC)
                               FROM test)
WITH NO DATA AND STATISTICS;
```

Vantage copies these zeroed statistics ...	From this source table column set ...	To this target table column set ...	Because ...
single column	x	a	column a in target table t2 is the exact analog of column x in source table test.
	y	none	there is no analog of source table column y in target table t2.
single-column NUPI	a	c	column c in target table t2 is the exact analog of column a in source table test.
multicolumn	(a,b)	none	the order of columns b and c in the definition of target table t2 is different than the order of columns b and a in the definition of source table test.
	(x,y,z)		there is no multicolumn index set defined for source table t2 on an analogous column set to (x,y,z) and (x,y) in source table test.
multicolumn NUSI	(x,y)	none	there is no multicolumn index set defined for source table t2 on an analogous column set to (x,y,z) and (x,y) in source table test.

Example 2: Copying Single-Column Statistics

In this example, Vantage copies the single-column statistics from test1 as zeroed single-column statistics on columns a , b , and c in t2. The index statistics on (x1,y1) are not copied as zeroed index statistics on columns (a,b) in t2 because the order of columns a and b in t2 is different than the order of columns x 1 and y1 in test1.

```
CREATE TABLE t2 (a, b, c) AS (SELECT k1 AS colA, y1 AS colB,
                                   z1 AS colC)
                               FROM test1)
WITH NO DATA AND STATISTICS
INDEX(a, b);
```

Vantage copies these zeroed statistics ...	From this source table column set ...	To this target table column set ...	Because ...
single column	k1	a	the new columns in target table t2 are exact analogs of the existing columns in source table test1.
	y1	b	
	z1	c	
	• x1 • a1 • b1	none	there are no analogs of these source table columns defined in target table t2.
single-column NUSI	z1	none	there is no single-column NUSI defined on the analog to source table column z1, column c, in the source table; instead, there is a multicolumn NUSI defined on the column set (a,b). The NUSI on (a,b) in target table t2 is defined without statistics being copied because no statistics were collected on their analog column set, (k1,y1), in source table test1.
multicolumn NUPI	(x1,y1)	none	the order of the index columns specified in the subquery select list is different than the column order of the test1 index.

Example: Copying PARTITION Statistics

Vantage copies PARTITION statistics in the following scenarios.

When you copy PARTITION statistics, the statistics copied to the target table might not correctly represent the data in the target table because of differences in internal partition number mapping

between the source and target tables. This is true even if the table definitions returned by a SHOW TABLE request are identical and the data is the same in both tables.

It is critical to understand that there is no way to guarantee that a target table created using a CREATE TABLE AS ... WITH DATA AND STATISTICS request is identical to the source table from which its statistics are copied down to the level of internal partition numbers, and it is important to understand that even though the two tables might appear to be identical from comparing their definitions using the output of SHOW TABLE requests on the tables.

As a general rule, you should always recollect the PARTITION statistics for the target table when you copy them from a source table.

Consider this DDL definition for the examples in this set.

```
CREATE SET TABLE test3,  
NO FALLBACK, NO BEFORE JOURNAL, NO AFTER JOURNAL,  
CHECKSUM = DEFAULT (  
  a INTEGER,  
  b DATE,  
  c INTEGER,  
  e INTEGER)  
PRIMARY INDEX (c)  
PARTITION BY RANGE_N (e BETWEEN 1 AND 1000000 EACH 50000);
```

For the examples that follow, assume the following statistics are collected on test3.

Statistics	Test3 Table Column Set
single column	<ul style="list-style-type: none">• a• b
single-column index on PPI NUPI	c
single-column PARTITION	PARTITION
composite NUPI	(a, c)
composite PARTITION	(PARTITION, a)

Example1

In this example, Vantage copies all the single column, multicolumn, and index statistics as well as the single-column PARTITION statistics and the multicolumn PARTITION statistics for the column set (PARTITION, a).

```
CREATE TABLE t8 AS  
test3  
WITH DATA AND STATISTICS;
```

Vantage copies these statistics ...	From this source table column set ...	To this target table column set ...	Because ...
single column	a	a	the system copies all columns from source table test3 to target table t8 with the same column names and properties and without using a subquery.
	b	b	
single-column index on PPI NUPI	c	c	
single-column PARTITION	PARTITION	PARTITION	
composite NUPI	(a, c)	(a, c)	
composite PARTITION	(PARTITION, a)	(PARTITION, a)	

Example2

In this example, Vantage does not copy PARTITION statistics because the target table columns are specified in a subquery; however, the system does copy the single-column and multicolumn statistics from test3 to t8.

```
CREATE TABLE t8 AS (SELECT *
                     FROM test3)
WITH DATA AND STATISTICS;
```

Vantage copies these statistics ...	From this source table column set ...	To this target table column set ...	Because ...
single column	a	a	the columns in source table test3 and target table t8 are identical, so the system copies their single-column statistics.
	b	b	
single-column index on PPI NUPI	c	c	
composite NUPI	(a, c)	(a, c)	the columns in source table test3 and target table t8 are identical, so the system copies their multicolumn statistics.
single-column PARTITION	PARTITION	none	the system does not copy PARTITION statistics, either single-column or as part of a composite, if the target table columns are specified in a subquery.
composite PARTITION	(PARTITION, a)	none	

Example3

In this example, Vantage does not copy PARTITION statistics because indexes and partitioning are not copied to the target table when any index definition for the target table is specified explicitly. However, the system does copy all single-column and multicolumn statistics from test3 to t8.

```
CREATE TABLE t8 AS test3
WITH DATA AND STATISTICS
INDEX(a, c);
```

Vantage copies these statistics ...	From this source table column set ...	To this target table column set ...	Because ...
single column	a	a	the columns in source table test3 and target table t8 are identical, so the system copies their single-column statistics.
	b	b	
single-column index on PPI NUPI	c	c	
composite NUPI	(a, c)	(a, c)	the columns in source table test3 and target table t8 are identical, so the system copies their multicolumn statistics.
single-column PARTITION	PARTITION	none	the system does not copy indexes or PARTITION statistics if any index is defined explicitly for the target table.
composite PARTITION	(PARTITION, a)	none	

INDEX Definitions

You can define a new primary index or primary AMP index for a new table using the CREATE TABLE AS syntax.

Example: CREATE TABLE AS, CT AS, and Table Kind

If you use the abbreviated CT AS syntax rather than the full CREATE TABLE AS syntax when creating a copied target table definition, you cannot specify a table kind. Because of this limitation, Vantage automatically defaults the definition of the target table kind to the table kind of the source table in the operation, as the following example illustrates.

Assume you are running in Teradata session mode, where the default table kind is SET.

First you create the source table whose definition and data are to be copied to a new table under a different name.

```
CREATE MULTISET TABLE test_m (
  i INTEGER)
PRIMARY INDEX(i);
```

Run SHOW TABLE on test_m to report its definition as stored by the system:

```
SHOW TABLE test_m;
```

The complete table definition reported by SHOW TABLE, which includes defaults for journaling and checksums, is as follows.

```
CREATE MULTISET TABLE ARUN.test_m ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT
  (
    i INTEGER)
PRIMARY INDEX ( i );
```

Now copy the definition for test_m and its data to a new table, test_m2:

```
CREATE TABLE test_m2 AS test_m WITH DATA;
```

Because you used the CREATE TABLE AS syntax, you were unable to specify the MULTISET table kind.

Now run SHOW TABLE on test_m2 to report its definition as stored by the system:

```
SHOW TABLE test_m2;
```

The result is as follows:

```
CREATE MULTISET TABLE ARUN.test_m2 ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT
  (
    i INTEGER)
PRIMARY INDEX ( i );
```

Because the SHOW TABLE reports on both tables indicate their table kind is MULTISET, you can see that the default table kind was copied from the definition of test_m to the definition of test_m2.

Example: Copy Table with Data and Default Column Names

This CREATE TABLE AS ... WITH DATA request selects all columns from the subquery for inclusion in target_table and retains their names. The request also copies the data from the selected columns into target_table.

Because the request specifies a subquery and no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of subquery_table.

```
CREATE TABLE target_table AS (
  SELECT *
  FROM subquery_table )
WITH DATA;
```

Example: Copy Table without Data and Default Column Names

This CREATE TABLE AS ... WITH NO DATA request selects a subset of the columns from subquery_table for inclusion in target_table and retains their names.

Because the request specifies a subquery and no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of subquery_table.

```
CREATE TABLE target_table AS (
  SELECT column_1, column_2
  FROM subquery_table )
WITH NO DATA;
```

Example: Changing Column Grouping for Column-Partitioned Table Using CREATE TABLE AS Syntax

This request creates a column-partitioned table with columns c and d assigned to the same partition using the column list rather than the partitioning to group the columns.

```
CREATE MULTISET TABLE t37 (
  a INTEGER,
  b INTEGER,
  (c INTEGER,
  d INTEGER))
PARTITION BY (RANGE_N(a BETWEEN 1
                      AND 10
                      EACH 1),
              COLUMN);
```

This results in the following table definition for t37.

```
CREATE MULTiset TABLE t37 ,NO FALLBACK ,NO BEFORE JOURNAL ,
NO AFTER JOURNAL ,CHECKSUM = DEFAULT ,DEFAULT MERGEBLOCKRATIO (
  a INTEGER,
  b INTEGER,
  c INTEGER,
  d INTEGER)
NO PRIMARY INDEX
PARTITION BY (RANGE_N(a BETWEEN 1
                        AND    10
                        EACH 1),
              COLUMN(a,b));
```

You next create table t37a by copying the definition from table t37 without copying its data.

```
CREATE TABLE t37a AS t37
WITH NO DATA;
```

This results in the following table definition for t37a. Table t37a has the same PARTITION BY clause and the same column grouping as table t37.

```
CREATE TABLE t37a ,NO FALLBACK ,NO BEFORE JOURNAL ,
NO AFTER JOURNAL ,CHECKSUM = DEFAULT ,DEFAULT MERGEBLOCKRATIO (
  a INTEGER,
  b INTEGER,
  c INTEGER,
  d INTEGER)
NO PRIMARY INDEX
PARTITION BY (RANGE_N(a BETWEEN 1
                        AND    10
                        EACH 1),
              COLUMN(a,b));
```

You next create table t37b by copying the definition for table t37 and modifying the grouping, which you specify in the table column list as follows.

```
CREATE TABLE t37b (a, (b, c), d) AS t37
WITH NO DATA;
```

This results in a table definition for t37b as follows. Table t37b has a PARTITION BY clause as does table t37, but the column grouping is as specified in the SQL table column list of the CREATE TABLE request for t37b.

```

CREATE MULTiset TABLE t37b, NO FALLBACK, NO BEFORE JOURNAL ,
NO AFTER JOURNAL, CHECKSUM = DEFAULT, DEFAULT MERGEBLOCKRATIO (
  a INTEGER,
  b INTEGER,
  c INTEGER,
  d INTEGER)
NO PRIMARY INDEX
PARTITION BY (RANGE_N(a BETWEEN 1
                        AND    10
                        EACH 1),
              COLUMN(a,d) ADD 10);

```

Examples

Example: CHARACTER Partitioning Using a CASE_N Function

The following example specifies character partitioning with a partitioning expression based on the CASE_N function. This example does not support static row partition elimination because the CASE_N function includes LIKE operators. See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

```

CREATE TABLE accounts (
  cust_id      INTEGER,
  last_name    VARCHAR(30) CHARACTER SET UNICODE NOT CASESPECIFIC,
  first_name   VARCHAR(30),
  city         VARCHAR(50))
PRIMARY INDEX (cust_id)
PARTITION BY CASE_N (last_name LIKE 'A%',
                     last_name LIKE 'B%',
                     last_name LIKE 'C%',
                     last_name LIKE 'D%',
                     last_name LIKE 'E%',
                     last_name LIKE 'F%',
                     last_name LIKE 'G%',
                     last_name LIKE 'H%',
                     last_name LIKE 'I%',
                     last_name LIKE 'J%',
                     last_name LIKE 'K%',
                     last_name LIKE 'L%',
                     last_name LIKE 'M%',
                     last_name LIKE 'N%',
                     last_name LIKE 'O%',

```

```

last_name LIKE 'P%',
last_name LIKE 'Q%',
last_name LIKE 'R%',
last_name LIKE 'S%',
last_name LIKE 'T%',
last_name LIKE 'U%',
last_name LIKE 'V%',
last_name LIKE 'W%',
last_name LIKE 'X%',
last_name LIKE 'Y%',
last_name LIKE 'Z%',
NO CASE,
UNKNOWN);

```

The example below supports static row partition elimination because the CASE_N function does not include LIKE operators, which enhances the performance of queries on the table.

```

CREATE TABLE accounts (
  cust_id      INTEGER,
  last_name    VARCHAR(30) CHARACTER SET UNICODE NOT CASESPECIFIC,
  first_name   VARCHAR(30),
  city         VARCHAR(50))
PRIMARY INDEX (cust_id)
PARTITION BY CASE_N (last_name < 'A', last_name < 'B',
                     last_name < 'C', last_name < 'D',
                     last_name < 'E', last_name < 'F',
                     last_name < 'G', last_name < 'H',
                     last_name < 'I', last_name < 'J',
                     last_name < 'K', last_name < 'L',
                     last_name < 'M', last_name < 'N',
                     last_name < 'O', last_name < 'P',
                     last_name < 'Q', last_name < 'R',
                     last_name < 'S', last_name < 'T',
                     last_name < 'U', last_name < 'V',
                     last_name < 'W', last_name < 'X',
                     last_name < 'Y', last_name >= 'Z',
                     UNKNOWN);

```

Example: CHARACTER Partitioning Defined Using a RANGE_N Function

This example is similar to [Example: CHARACTER Partitioning Using a CASE_N Function](#), but specifies a partitioning expression based on a RANGE_N function instead of a CASE_N function and LIKE.

This definition is preferred to that of [Example: CHARACTER Partitioning Using a CASE_N Function](#) because static row partition elimination is not supported for CASE_N expressions that specify LIKE operators.

In this example, the character in position 2 of *last_name* is a SPACE, so 'B' would go into partition 2. If the character in position 2 of *last_name* collates to less than the SPACE character (for example, TAB), the system assigns the row to partition 1.

In [Example: CHARACTER Partitioning Using a CASE_N Function](#), a match on the first character is all that is required for a row to be assigned to the corresponding partition. In that case, the row would go into partition 3, because the conditions for the first two partitions would both be false: *last_name* < 'A', *last_name* < 'B'.

```
CREATE TABLE accounts2 (
  cust_id      INTEGER,
  last_name    VARCHAR(30) NOT CASESPECIFIC,
  first_name   VARCHAR(30),
  city         VARCHAR(50))
PRIMARY INDEX (cust_id)
PARTITION BY RANGE_N(last_name BETWEEN 'A','B','C','D','E','F','G',
                     'H','I','J','K','L','M','N',
                     'O','P','Q','R','S','T','U',
                     'V','W','X','Y','Z'
                     AND
                     'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ',
                     NO RANGE,UNKNOWN);
```

The partitioning in the first part of [Example: CHARACTER Partitioning Using a CASE_N Function](#) can be duplicated using the following RANGE_N partitioning expression using a CAST expression. However, this form of RANGE_N partitioning is not desirable because static row partition elimination is not available when there is a CAST on the RANGE_N test value.

```
PARTITION BY RANGE_N(CAST(last_name AS CHARACTER(1))
                     BETWEEN 'A','B','C','D','E','F','G',
                     'H','I','J','K','L','M','N',
                     'O','P','Q','R','S','T','U',
                     'V','W','X','Y','Z'
                     AND 'Z',
                     NO RANGE,
                     UNKNOWN);
```

You can enable static partition elimination for this table by creating the same table, but specifying ranges that are suffixed by 29 nulls each, because the column is typed as VARCHAR(30), so each range specifies one character suffixed with the number of nulls required to fill the remaining 29 characters allotted for the column. If the relevant column had been specified as VARCHAR(50), then you would need to suffix each character specification for the range with 49 nulls, and so on. The RANGE_N example uses hexadecimal character literals. If the session character set is Latin, 2 digits are required to represent each character (58 zeros). If the session character set is Unicode, 4 digits are to represent each character (116 zeros).

Specifying the partitioning expression in this way enables the Optimizer to use static partition elimination for requests made against the table, and by eliminating unnecessary partitions, the speed of the table scan can be significantly enhanced.

[illegible]

Example: Character Partitioning Defined Using RANGE_N Functions

This example creates a partitioned primary index based on character data specified for a RANGE_N function. Notice that no EACH clause is specified for the RANGE_N partitioning expression in level 1 of the multilevel partitioning in this case.

```
CREATE TABLE t1(
  i INTEGER,
  j CHARACTER(4),
  k INTEGER)
PRIMARY INDEX(i)
PARTITION BY (RANGE_N(j BETWEEN 'aaaa','cccc','eeee',
                        'gggg','iiii','kkkk',
                        'mmmm','oooo','qqqq',
```

```

        'ssss'
        AND    'tttt',
NO RANGE),
    RANGE_N(k BETWEEN 1
            AND    10
            EACH   1));

```

Example: Character Partitioning, Session Mode, and Case Specificity

If you create the character multilevel partitioned primary index table in the following example in an ANSI mode session, the system treats the test value (`j || 'a'`) as CASESPECIFIC because character literals are case specific by default in ANSI mode sessions. Similarly, all range boundary comparisons in the partitioning expression are case sensitive.

Note that this is not an advisable practice because specifying the `||` concatenation operator in a partitioning expression where one or both sides of the concatenation involves a column reference eliminates the possibility of the system being able to use either static or dynamic partition elimination to enhance the performance of queries made against the table.

If you create the table in a Teradata mode session, Vantage treats the test value as NOT CASESPECIFIC, and range boundary comparisons in the partitioning expression are case blind.

```

CREATE TABLE t1(
  i INTEGER,
  j CHARACTER(4) NOT CASESPECIFIC,
  k INTEGER)
PRIMARY INDEX(i)
PARTITION BY (RANGE_N(j || 'a' BETWEEN 'aaaa' AND 'bbbb',
                        'cccc' AND 'dddd',
                        'eeee' AND 'ffff',
                        'gggg' AND 'hhhh',
                        'iiii' AND 'jjjj',
                        'kkkk' AND 'llll',
                        'mmmm' AND 'nnnn',
                        'oooo' AND 'pppp',
                        'qqqq' AND 'rrrr',
                        'ssss' AND 'tttt',
NO RANGE),
    RANGE_N(k BETWEEN 1
            AND    10
            EACH   1));

```


Example: Character Partitioning and System-Derived PARTITION[#Ln] Columns

Assume you have created the following tables:

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) NOT CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_comment       VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY (RANGE_N(o_custkey BETWEEN 0
                      AND 49999
                      EACH 1000),
              RANGE_N(o_orderdate BETWEEN DATE '2000-01-01'
                      AND DATE '2006-12-31'
                      EACH INTERVAL '1' MONTH),
              CASE_N(o_orderstatus = 'S', o_orderstatus = 'C',
                    o_orderstatus = 'F'))
UNIQUE INDEX (o_orderkey);
CREATE TABLE lineitem (
  l_orderkey      INTEGER NOT NULL,
  l_partkey       INTEGER NOT NULL,
  l_suppkey       INTEGER,
  l_linenumbers   INTEGER,
  l_quantity      INTEGER NOT NULL,
  l_extendedprice DECIMAL(13,2) NOT NULL,
  l_discount      DECIMAL(13,2),
  l_tax           DECIMAL(13,2),
  l_returnflag    CHARACTER(1),
  l_linestatus    CHARACTER(1),
  l_shipdate      DATE FORMAT 'yyyy-mm-dd',
  l_commitdate    DATE FORMAT 'yyyy-mm-dd',
  l_receiptdate   DATE FORMAT 'yyyy-mm-dd',
  l_shipinstruct  VARCHAR(25),
  l_shipmode      VARCHAR(25),
  l_comment       VARCHAR(44))
PRIMARY INDEX (l_orderkey)
PARTITION BY (RANGE_N(l_suppkey BETWEEN 0
```

```

AND 4999
EACH 100),
RANGE_N(l_shipdate BETWEEN DATE '2000-01-01'
AND DATE '2006-12-31'
EACH INTERVAL '1' MONTH),
CASE_N(l_shipmode = 'USPS FirstClass',
l_shipmode = 'UPS',
l_shipmode = 'FedEx',
NO CASE OR UNKNOWN));

```

The following are examples of the usage of the system-derived PARTITION columns.

Select rows for all customer orders with a status of S. The system does not return the value of PARTITION#L3 because it was not specified explicitly in the select list for the request. Note that it is not necessary to qualify PARTITION#L3 because the request references only one table.

```

SELECT *
FROM orders
WHERE orders.PARTITION#L3=1;

```

Select rows for customers with IDs between 1000 and 1999 (partition 2 for level 1) with an order status of C (partition 2 for level 3). The system does not return the values of PARTITION#L1 and PARTITION#L3 because they were not specified explicitly in the select list for the request.

```

SELECT *
FROM orders
WHERE orders.PARTITION#L1=2
AND PARTITION#L3=2;

```

Select rows for suppliers with IDs between 3700 and 3799 (partition 38 for level 1) in January 2003 (partition 37 for level 2) where the shipping mode is FedEx (partition 3 for level 3). PARTITION is the partition number for the combined partitioning expression and is equal to $(\text{PARTITION\#L1}-1)*84*4 + (\text{PARTITION\#L2}-1)*4 + (\text{PARTITION\#L3})$ where 84 is the number of partitions defined for level 2 and 4 is the number of partitions defined for level 3 for this example. That is, 12579 is derived from $(38-1)*84*4 + (37-1)*4 + 3$.

Vantage does not return the value of PARTITION because it was not specified explicitly in the select list for the request.

```

SELECT *
FROM lineitem
WHERE lineitem.PARTITION = 12579;

```

Select rows for ordered items with an order status of F that were shipped using UPS. Note that PARTITION#L3 must be qualified with lineitem because more than one table is referenced in the request.

```
SELECT *
FROM orders, lineitem
WHERE orders.PARTITION#L3 = 3
AND   lineitem.PARTITION#L3 = 2
AND   orders.o_orderkey = lineitem.l_orderkey;
```

Example: RANGE_N Partitioning Expression Using the END Bound Function

This example creates a table whose primary index is partitioned using a RANGE_N expression and an END bound function.

```
CREATE TABLE sales_history (
  product_code      CHARACTER(8),
  quantity_sold     INTEGER,
  transaction_period PERIOD(DATE)
PRIMARY INDEX (product_code)
PARTITION BY RANGE_N(END(transaction_period)
                     BETWEEN DATE '2008-03-31'
                     AND   DATE '2015-12-31'
                     EACH INTERVAL '1' YEAR);
```

Example: CASE_N Partitioning Expression Using the END Bound Function

This example creates a table whose primary index is partitioned using a CASE_N expression and END bound functions.

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderperiod   PERIOD(DATE) NOT NULL
  o_orderpriority CHARACTER(21),
  o_comment       VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY CASE_N (END(o_orderperiod) <= DATE '2010-03-31', /*Q1*/
                     END(o_orderperiod) <= DATE '2010-06-30', /*Q2*/
                     END(o_orderperiod) <= DATE '2010-09-30', /*Q3*/
```

```
END(o_orderperiod) <= DATE '2010-12-31' /*Q4*/
);
```

Example: 15 Levels of Multilevel Partitioning

The following example demonstrates partitioning with the 2-byte partitioning maximum of 15 levels. A table with an 8-byte partitioning can have up to 62 partitioning levels.

In order not to exceed 65,535 partitions for the combined partitioning expression, each level must define a maximum of 2 partitions except for one level that can define a maximum of three partitions.

In this example, the last partitioning expression defines three partitions and the other partitioning expressions each define two partitions. Therefore, a total of 49,152 partitions are defined for the combined partitioning expression, which is fewer than the maximum of 65,535 for a 2-byte partition number.

Note that while the partitioning expressions are defined using a mix of RANGE_N and CASE_N function-based expressions, each level is defined atomically using either one or the other function, but not both. You cannot mix RANGE_N and CASE_N functions within any of the partitioning expressions, nor can a partitioning expression be based on anything but a RANGE_N or CASE_N function when you define a multilevel partitioning.

```
CREATE TABLE product (
  product_id    INTEGER NOT NULL,
  p_color       INTEGER NOT NULL,
  p_size        INTEGER NOT NULL,
  p_width       SMALLINT NOT NULL,
  p_depth       SMALLINT NOT NULL,
  p_height      SMALLINT NOT NULL,
  p_weight      SMALLINT NOT NULL,
  p_catalog     INTEGER NOT NULL,
  p_code        BYTEINT NOT NULL,
  p_subcode     SMALLINT NOT NULL,
  rating        BYTEINT NOT NULL,
  discontinued  BYTEINT NOT NULL,
  in_stock      INTEGER NOT NULL,
  back_order    INTEGER NOT NULL,
  total_sold    INTEGER NOT NULL,
  product_date  DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  note         CHARACTER(255))
PRIMARY INDEX (product_id)
PARTITION BY (RANGE_N(p_color      BETWEEN 1
                      AND 1000
                      EACH 500),
              RANGE_N(p_size      BETWEEN 1
                      AND 1000, 1001
```

```

                                AND 48000000),
RANGE_N(p_width                BETWEEN 1
                                AND 400
                                EACH 200),
RANGE_N(p_depth                BETWEEN 1
                                AND 400
                                EACH 200),
RANGE_N(p_height               BETWEEN 1
                                AND 400
                                EACH 200),
RANGE_N(p_weight               BETWEEN 1
                                AND 2000
                                EACH 1000),
CASE_N(p_catalog <= 1,NO CASE),
RANGE_N(p_code                 BETWEEN 1
                                AND 100
                                EACH 50),
RANGE_N(p_subcode              BETWEEN 1
                                AND 10000
                                EACH 5000),
RANGE_N(rating                 BETWEEN 1
                                AND 10
                                EACH 5),
CASE_N(discontinued = 0,
        discontinued = 1),
CASE_N(in_stock < 50, NO CASE),
CASE_N(back_order = 0, NO CASE),
CASE_N(total_sold < 1000,NO CASE),
RANGE_N(product_date BETWEEN DATE '2004-01-01'
                                AND DATE '2006-12-31'
                                EACH INTERVAL '1' YEAR));

```

Example: Volatile Table With Single-Level Partitioning

The following example demonstrates a valid CREATE VOLATILE TABLE request with single-level partitioning.

```

CREATE VOLATILE TABLE V_lineitem (
  l_orderkey INTEGER NOT NULL,
  l_partkey  INTEGER NOT NULL,
  l_suppkey  INTEGER,
  l_linenum  INTEGER,
  l_quantity INTEGER NOT NULL,

```

```

l_extendedprice DECIMAL(13,2) NOT NULL,
l_discount DECIMAL(13,2),
l_tax DECIMAL(13,2),
l_returnflag CHARACTER(1),
l_linestatus CHARACTER(1),
l_shipdate DATE FORMAT 'yyyy-mm-dd',
l_commitdate DATE FORMAT 'yyyy-mm-dd',
l_receiptdate DATE FORMAT 'yyyy-mm-dd',
l_shipinstruct VARCHAR(25),
l_shipmode VARCHAR(10),
l_comment VARCHAR(44))
PRIMARY INDEX (l_orderkey)
PARTITION BY RANGE_N(l_shipdate BETWEEN DATE '1992-01-01'
                      AND      DATE '1998-12-31'
                      EACH INTERVAL '1' MONTH);

```

Example: Global Temporary Table With Single-Level Partitioning

This example demonstrates a valid CREATE GLOBAL TEMPORARY TABLE request with single-level partitioning.

```

CREATE GLOBAL TEMPORARY TABLE g_orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk         CHARACTER(16),
  o_shippriority  INTEGER,
  o_comment       VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_orderdate BETWEEN DATE '1992-01-01'
                      AND      DATE '1998-12-31'
                      EACH INTERVAL '1' MONTH)

UNIQUE INDEX (o_orderkey);

```

Example: Specifying a Column-Level Primary Key

The request in this example specifies a primary key only, so column_2 is mapped to a UPI. In this case the PRIMARY index cannot be null.

```
CREATE TABLE good_6 (
  column_1 INTEGER,
  column_2 INTEGER NOT NULL PRIMARY KEY);
```

Example: Default Primary Index When Neither Primary Index Nor Primary Key Is Specified

The request in this example specifies neither a primary index nor a primary key. Assuming that the DBS Control parameter PrimaryIndexDefault is set to D or P, Vantage maps the first unique column, column_2, to a UPI, which cannot be null.

```
CREATE TABLE good_7 (
  column_1 INTEGER,
  column_2 INTEGER NOT NULL UNIQUE,
  column_3 INTEGER NOT NULL UNIQUE);
```

This request defaults to the following definition.

```
CREATE TABLE good_7 (
  column_1 INTEGER,
  column_2 INTEGER NOT NULL UNIQUE,
  column_3 INTEGER NOT NULL UNIQUE)
UNIQUE PRIMARY INDEX (column_2);
```

Example: Local Journaling

As an example, given the following CREATE TABLE requests:

```
CREATE TABLE cyk.test_1, NO FALLBACK, AFTER JOURNAL (
  x INTEGER,
  y INTEGER)
PRIMARY INDEX(x);

CREATE TABLE cyk.test_2, NO FALLBACK, NOT LOCAL AFTER JOURNAL (
  x INTEGER,
  y INTEGER)
PRIMARY INDEX(x);

CREATE TABLE cyk.test_3, NO FALLBACK, LOCAL AFTER JOURNAL (
  x INTEGER,
```

```

    y INTEGER)
PRIMARY INDEX(x);

```

SHOW TABLE produces the following output:

```

CREATE TABLE cyk.test_1, NO FALLBACK, NO BEFORE JOURNAL,
  NOT LOCAL AFTER JOURNAL (
    x INTEGER,
    y INTEGER)
PRIMARY INDEX(x);

CREATE TABLE cyk.test_2, NO FALLBACK, NO BEFORE JOURNAL,
  NOT LOCAL AFTER JOURNAL (
    x INTEGER,
    y INTEGER)
PRIMARY INDEX(x);

CREATE TABLE cyk.test_3, NO FALLBACK, NO BEFORE JOURNAL,
  LOCAL AFTER JOURNAL (
    x INTEGER,
    y INTEGER)
PRIMARY INDEX(x);

```

Example: CREATE TABLE AS ... WITH NO DATA with Named Expressions

This CREATE TABLE AS ... WITH NO DATA request creates target_table from a subquery where all column expressions are named explicitly. No column names are defined for target_table because it is unnecessary.

Because the request specifies a subquery and no explicit table kind is specified, the table kind of target_table defaults to the session mode default, not to the table kind of subquery_table.

```

CREATE TABLE target_table AS (SELECT column_x + 1 AS column_x,
                                column_y + 1 AS column_y
                                FROM subquery_table )

WITH NO DATA;

```

Example: CREATE TABLE AS ... WITH NO DATA with Nonunique Secondary Index

This CREATE TABLE AS ... WITH NO DATA request creates a NUSI on column_y of target_table. The primary index defaults to column_x and no indexes are copied from subquery_table.

Because the request specifies a subquery and no explicit table kind is specified, the table kind of `target_table` defaults to the session mode default, not to the table kind of `subquery_table`. `target_table` also takes the default definitions for all table options clause attributes.

```
CREATE TABLE target_table AS (SELECT column_x, column_y
                                FROM subquery_table)

WITH NO DATA
INDEX (column_y);
```

Example: CREATE TABLE AS ... WITH DATA with Nonunique Secondary Index

This `CREATE TABLE AS ... WITH DATA` request creates a new NUSI index on `column_2` of `target_table`. The request also copies the data from the selected columns into `target_table`.

Because the request specifies a subquery and no explicit table kind is specified, the table kind of `target_table` defaults to the session mode default, not to the table kind of `subquery_table`.

```
CREATE TABLE target_table (
    column_1,
    column_2)
AS (SELECT *
    FROM subquery_table )
WITH DATA
INDEX (column_2);
```

Example: Statistics Not Copied

Vantage does not copy statistics under the following scenarios.

Consider this table definition.

```
CREATE SET TABLE t1, NO FALLBACK,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT (
        x INTEGER,
        y CHAR(10) CHARACTER SET LATIN CASESPECIFIC,
        z INTEGER)
    UNIQUE PRIMARY INDEX (x);
```

Assume that single-column statistics have been collected on columns `x`, `y`, and `z`.

Example1

In this example, Vantage does not copy statistics because only partial data is copied. This occurs because neither y nor z is uniquely constrained, so the system eliminates all duplicate rows, which might cause a difference in cardinality between the 2 tables.

```
CREATE SET TABLE t9(x, y) AS (SELECT y, z
                                FROM t1)

WITH DATA AND STATISTICS
PRIMARY INDEX(x, y);
```

Example2

In this example, Vantage copies partial statistics. The single-column statistics from source table columns x and z are copied as single-column statistics for target table columns a and c, respectively. Single-column statistics on y in the source table are not eligible to be copied to target table column b in the target table because the NOT CASESPECIFIC column attribute of b is not the same as the column default attribute of source table column y.

```
CREATE TABLE t9(a, b NOT CASESPECIFIC, c) AS (SELECT x, y, z
                                                  FROM t1)

WITH DATA AND STATISTICS;
```

Vantage copies these statistics ...	From this source table column set ...	To this target table column set ...	Because ...
single-table	x	a	the columns x and z in source table t1 and columns a and c in target table t9 are identical, so the system copies their single-column statistics.
	z	c	
	y	none	the definition of column b in the target table is changed from the definition of its analog column y in the source table because it adds a NOT CASESPECIFIC attribute.

Example3

In this example, single-column statistics on source table column y are not eligible to be copied for target table column x because data for this column is modified in the target table with the UPPERCASE attribute.

```
CREATE TABLE t9(x UPPERCASE, y) AS (SELECT y, z
                                         FROM t1)

WITH DATA AND STATISTICS;
```

Vantage copies these statistics ...	From this source table column set ...	To this target table column set ...	Because ...
single-table	z	y	column z in source table t1 and column y in target table t9 are identical, so the system copies their single-column statistics.
	y	none	the definition of column x in the target table is changed from the definition of its analog column y in the source table because it adds an UPPERCASE attribute.

Example: Statistics Are Not Copied for Conditions in The Subquery

Vantage does not copy statistics if any single table conditions or join conditions are specified in the WHERE clause of the subquery as these examples demonstrate. A warning message is returned to the requestor that says statistics cannot be copied.

Example1

In this example, Vantage does not copy statistics because only partial data is copied based on the WHERE condition that limits copied rows to those having a value for column x greater than 10.

```
CREATE TABLE t2 AS (SELECT *
                     FROM test
                     WHERE x > 10)
WITH DATA AND STATISTICS;
```

Example2

In this example, Vantage does not copy statistics because only partial data is copied based on the WHERE clause in the subquery that limits copied rows to those having a value for column x1 greater than 10 AND a value for column y1 equal to 20.

```
CREATE TABLE t2 AS (SELECT *
                     FROM test1
                     WHERE x1 > 10
                     AND   y1 = 20)
WITH DATA AND STATISTICS;
```

Example3

In this example, Vantage does not copy statistics because tables are left outer joined in the subquery.

```
CREATE TABLE t2 AS (SELECT *
                    FROM test LEFT OUTER JOIN test1 ON test.x=test1.x
WITH DATA AND STATISTICS;
```

Example4

In this example, Vantage does not copy statistics because tables are right outer joined in the subquery.

```
CREATE TABLE t2 (a, b) AS (SELECT x1, y1
                          FROM test RIGHT JOIN
                          test1 ON test. a=test1.a1)
WITH DATA AND STATISTICS;
```

Example: Statistics Are Not Copied When Set Or Aggregation Operators Are Specified In The Query

Vantage does not copy statistics if set operators or aggregation operations are specified in the query as these examples demonstrate.

Example1

In this example, Vantage does not copy statistics because of the aggregating GROUP BY clause in the subquery.

```
CREATE TABLE t3 (c1, c2) AS (SELECT MAX(x), y
                              FROM test
                              GROUP BY 2)
WITH DATA AND STATISTICS;
```

Example2

In this example, Vantage does not copy statistics because of the aggregating GROUP BY clause in the subquery:

```
CREATE TABLE t3 (c1, c2) AS (SELECT COUNT(x), y
                              FROM test
                              GROUP BY 2)
WITH DATA AND STATISTICS;
```

Example3

In this example, Vantage does not copy statistics because of the aggregating GROUP BY and HAVING clauses in the subquery.

```
CREATE TABLE t3(a, b, c) AS (SELECT x1, MIN(y1), AVG(a1)
                              FROM test1
                              GROUP BY x1
                              HAVING AVG(y1) > 40)

WITH DATA AND STATISTICS;
```

Example: Statistics Are Not Copied If An OLAP Operation Is Specified in The Subquery

Vantage does not copy statistics if an OLAP function is specified in the subquery as the examples in this set demonstrate.

Example1

In this example, Vantage does not copy statistics because of the OLAP RANK function in the subquery.

```
CREATE TABLE t3 (c1, c2, c3) AS (SELECT x, RANK(y), z
                                   FROM test
                                   QUALIFY RANK(X)> 1)

WITH DATA AND STATISTICS;
```

Example2

In this example, Vantage does not copy statistics because of the partial data returned by the SAMPLE operator.

```
CREATE TABLE t3 (c1, c2, c3) AS (SELECT x, y, z
                                   FROM test SAMPLE 4)

WITH DATA AND STATISTICS;
```

Example: Statistics Are Not Copied If a Join Operation Is Specified in the Subquery

Vantage does not copy statistics if a join operation is specified in the subquery. as the examples in this set demonstrate.

Consider the following view definitions for the examples in this set.

```
CREATE VIEW v2 (i, j) AS (
  SELECT SUM(x), y
  FROM test
  GROUP BY 2);
CREATE VIEW v3 (i, j) AS (
```

```

SELECT CASE
      WHEN z1='abc'
      THEN 1
      ELSE 0, a1+1
FROM test1);

```

Example1

In this example, Vantage does not copy statistics because of the join specification in the subquery.

```

CREATE TABLE t4 AS (SELECT *
                     FROM test, test1)
WITH DATA AND STATISTICS
PRIMARY INDEX (z, z1)
INDEX (b, b1);

```

Example2

In this example, Vantage does not copy statistics because of the join specification in the subquery.

```

CREATE TABLE t6 (c1, c2, c3, c4) AS (SELECT x1, y1, x-1, y*2
                                         FROM test1, test)
WITH DATA AND STATISTICS;

```

Example3

In this example, Vantage does not copy statistics because of the join specification in the subquery.

```

CREATE TABLE t5 AS (SELECT *
                     FROM v2, v3)
WITH DATA AND STATISTICS;

```

Example4

In this example, Vantage does not copy statistics because of the join specification in the subquery.

```

CREATE TABLE t5 AS (SELECT *
                     FROM v3, test)
WITH DATA AND STATISTICS;

```

Example: Statistics Are Not Copied If Functions Or Expressions Are Specified in the Subquery

Consider the following view definitions for the examples in this set.

```
CREATE VIEW v1 (i, j) AS (
  SELECT x+1, b+1
  FROM test);
CREATE VIEW v3 (i, j) AS (SELECT CASE
                           WHEN z1='abc'
                           THEN 1
                           ELSE 0, a1+1
                           FROM test1);
```

Vantage does not copy statistics for specific column sets or indexes if the subquery specifies functions or expressions of any kind. This is because such data manipulations cause the resulting column set or index value set to be transformed into something different from that of the source relation as these examples demonstrate.

Example1

In this example, Vantage does not copy statistics because the view v1 selects column expressions from its underlying table test, so the resulting columns in the target table t1 are different from the columns in test.

```
CREATE TABLE t1 AS (SELECT *
                     FROM v1)
WITH NO DATA AND STATISTICS;
```

Example2

In this example, Vantage does not copy statistics because the view v1 selects column expressions from its underlying table test, so the resulting column data in the target table would be different from the columns in test. Also note that you cannot copy zeroed statistics.

```
CREATE TABLE t5 AS (SELECT *
                     FROM v1)
WITH DATA AND STATISTICS;
```

Example3

In this example, Vantage does not copy statistics because the view v3 is defined using a CASE expression, so the resulting column data in the target table would be different from the columns in test1.

```
CREATE TABLE t5 AS (SELECT *
                     FROM v3)
WITH DATA AND STATISTICS;
```

Example4

In this example, Vantage does not copy statistics because the subquery specifies simple arithmetic expressions that would cause the data in the target table to be different from the columns in test.

```
CREATE TABLE t5 (c1, c2) AS (SELECT x+1, y+1
                              FROM test)
WITH DATA AND STATISTICS;
```

Example5

In this example, Vantage does not copy statistics because the subquery specifies simple arithmetic expressions that would cause the data in the target table to be different from the columns in test1.

```
CREATE TABLE t5 (c1, c2) AS (SELECT (k1/(y1+2)), b1+7
                              FROM test1)
WITH DATA AND STATISTICS;
```

Example6

In this example, Vantage does not copy statistics because the subquery specifies simple arithmetic expressions that would cause the data in the target table to be different from the columns in test.

```
CREATE TABLE t5 (a, b) AS (SELECT x+1 AS x, y+1 AS y
                              FROM test)
WITH DATA AND STATISTICS;
```

Example: CREATE TABLE AS Requests That Create Table with Primary Index

Assume that you define the following primary-indexed table.

```
CREATE TABLE source (
  column_1 INTEGER
  column_2 INTEGER)
UNIQUE PRIMARY INDEX (column_1);
```

The following CREATE TABLE AS requests both produce primary-indexed tables; however, the second creates the table with a nonunique primary index.

```
CREATE TABLE target
AS source
WITH NO DATA;
```



```
CREATE TABLE target AS (SELECT column_1, column_2
                        FROM source)

WITH DATA

PRIMARY INDEX (column_1);
```

Example: CREATE Request with Presence or Absence of PRIMARY KEY or UNIQUE Constraint and Setting of PrimaryIndexDefault

This example specifies neither an explicit PRIMARY INDEX clause nor an explicit NO PRIMARY INDEX clause, but does specify a UNIQUE constraint on column_2.

```
CREATE TABLE test_1 (
    column_1 INTEGER NOT NULL,
    column_2 INTEGER NOT NULL
    CONSTRAINT UNIQUE (column_2));
```

Regardless of the setting for the PrimaryIndexDefault parameter, Vantage creates this table with a UPI on column_2 because you neither explicitly specified a primary index nor NO PRIMARY INDEX. Because of this, the system follows the rules for creating a default primary index and creates a UPI on the first column with a UNIQUE constraint, which is column_2.

As a result, a SHOW TABLE request on test_1 returns the following SQL text.

```
CREATE SET TABLE test_1 ,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT (
    (
    column_1 INTEGER NOT NULL,
    column_2 INTEGER NOT NULL, CONSTRAINT UNIQUE (column_2);
```

Because no INDEX clause was specified, Vantage creates the table with a unique primary index on column_2 (marked as being for a PRIMARY KEY constraint in the data dictionary), regardless of the setting of the PrimaryIndexDefault parameter in DBS Control.

The following example also does not specify either an explicit PRIMARY INDEX clause or an explicit NO PRIMARY INDEX clause, but does specify a PRIMARY KEY constraint on column_2.

```
CREATE TABLE test_2 AS (
    column_1 INTEGER NOT NULL
```

```
column_2 INTEGER NOT NULL
CONSTRAINT PRIMARY KEY (column_2));
```

Regardless of the setting for the PrimaryIndexDefault parameter, Vantage creates this table with a UPI on *column_2* (marked as being for a PRIMARY KEY constraint in the data dictionary), because *column_2* is defined as the primary key for the table, so a SHOW TABLE request on *test_2* returns the following SQL text.

```
CREATE SET TABLE test_2 ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT
(
  column_1 INTEGER NOT NULL,
  column_2 INTEGER NOT NULL, CONSTRAINT PRIMARY KEY (column_2);
```

Because no INDEX clause was specified, Vantage creates the table with a unique primary index on *column_2* regardless of the setting of the PrimaryIndexDefault parameter in DBS Control.

This example does not specify either an explicit PRIMARY INDEX clause or an explicit NO PRIMARY INDEX clause. The request also specifies a UNIQUE constraint on *column_1* and a PRIMARY KEY constraint on *column_2*.

```
CREATE TABLE test_3 AS (
  column_1 INTEGER NOT NULL
  column_2 INTEGER NOT NULL
  CONSTRAINT UNIQUE (column_1)
  CONSTRAINT PRIMARY KEY (column_2));
```

In this case, Vantage follows the default hierarchy for converting non-primary index columns into a primary index and converts the PRIMARY KEY constraint to the unique primary index for the table because it has precedence over UNIQUE constraints in the default primary index rules hierarchy.

Regardless of the setting for the PrimaryIndexDefault parameter, Vantage creates this table with a UPI on *column_2* and converts the UNIQUE constraint on *column_1* to a unique secondary index, so a SHOW TABLE request on *test_3* returns the following create text.

```
CREATE SET TABLE test_3 ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT
(column_1 INTEGER NOT NULL,
 column_2 INTEGER NOT NULL,

                                CONSTRAINT UNIQUE
```

```
(column_1)
    CONSTRAINT PRIMARY KEY (column_2));
```

Example: NoPI Table CREATE Request That Converts PRIMARY KEY or UNIQUE Constraints to USIs

If you create a NoPI table that also specifies a PRIMARY KEY, one or more UNIQUE constraints, or both, Vantage creates unique secondary indexes on the same column sets as the constraints.

Consider this CREATE TABLE request.

```
CREATE TABLE nopi_constraints (
    column_1 INTEGER NOT NULL
    column_2 INTEGER NOT NULL
    CONSTRAINT UNIQUE (column_1)
    CONSTRAINT PRIMARY KEY (column_2))
NO PRIMARY INDEX;
```

For example, Vantage creates USIs for the two constraints because you have specified NO PRIMARY INDEX explicitly.

```
CREATE TABLE nopi_constraints (
    column_1 INTEGER NOT NULL
    column_2 INTEGER NOT NULL)
UNIQUE INDEX (column_1)
UNIQUE INDEX (column_2)
NO PRIMARY INDEX;
```

Example: CREATE Request That Produces a NoPI Table

This example creates a NoPI table named sales_temp1 regardless of the setting for PrimaryIndexDefault because you have specified NO PRIMARY INDEX explicitly:

```
CREATE MULTiset TABLE sales_temp1, FALLBACK, NO BEFORE JOURNAL,
NO AFTER JOURNAL, CHECKSUM = DEFAULT (
    item_nbr INTEGER NOT NULL,
    sale_date DATE FORMAT 'MM/DD/YYYY' NOT NULL,
    item_count INTEGER)
NO PRIMARY INDEX;
```

Example: CREATE TABLE ... AS Requests That Create NoPI Table

Assume the following primary-indexed source table definition for the following example set.

```
CREATE TABLE source_pi (
  column_1 INTEGER
  column_2 INTEGER)
UNIQUE PRIMARY INDEX (column_1);
```

Also assume you have created the following NoPI table definition for the example set:

```
CREATE TABLE source_nopi (
  column_1 INTEGER
  column_2 INTEGER)
NO PRIMARY INDEX;
```

The following CREATE TABLE ... AS requests all produce a NoPI table because in each case you have explicitly specified NO PRIMARY INDEX:

```
CREATE TABLE target_nopi AS
  source_pi
WITH DATA
NO PRIMARY INDEX;
CREATE TABLE target_nopi AS
  source_nopi
WITH DATA
NO PRIMARY INDEX;
CREATE TABLE target_nopi AS (
  SELECT column_1, column_2
  FROM source_pi)
WITH DATA
NO PRIMARY INDEX;
CREATE TABLE target_nopi AS (
  SELECT column_1, column_2
  FROM source_nopi)
WITH DATA
NO PRIMARY INDEX;
```

Unlike the preceding examples, the outcome of the following CREATE TABLE request depends on the setting of the DBS Control parameter PrimaryIndexDefault.

```
CREATE TABLE target AS (
  SELECT column_1, column_2
  FROM source_nopi)
WITH DATA;
```

IF the value for PrimaryIndexDefault is ...	THEN the resulting table has ...
N	no primary index. Even though you have not explicitly specified NO PRIMARY INDEX, you also have not specified any PRIMARY KEY or UNIQUE constraints on the columns of the table, so Vantage defines target as a NoPI table by default.
D or P	a nonunique primary index. The NUPI is defined by default, following the rules for selecting the primary index for a table when none is specified. In this case, the primary index is defined on the first column defined for target, which is column_1. The primary index is defined as a NUPI because no PRIMARY KEY or UNIQUE constraints are defined on any of the columns of target.

Example: Current and Historical Partitioning Using CURRENT_DATE in a CASE_N Expression

This example partitions an insurance company customer table into history and current partitions, where the history partition contains expired policies and the current partition contains current policies.

```
CREATE TABLE customer (
  cust_name          CHARACTER(8),
  policy_number      INTEGER,
  policy_expiration_date DATE FORMAT 'YYYY/MM/DD')
PRIMARY INDEX (cust_name, policy_number)
PARTITION BY CASE_N(policy_expiration_date>=CURRENT_DATE, NO CASE);
```

Suppose that customer was created on April 17, 2010.

The output of a SHOW TABLE request displays the user-specified partitioning expression as a DATE in ANSI format.

```
SHOW TABLE customer;
CREATE SET TABLE MOVEDATE.customer, NO FALLBACK, NO BEFORE JOURNAL,
                                NO AFTER JOURNAL, CHECKSUM=DEFAULT (
  cust_name CHARACTER(8) CHARACTER SET LATIN NOT CASESPECIFIC,
  policy_number INTEGER,
  policy_expiration_date DATE FORMAT 'YYYY/MM/DD')
```

```
PRIMARY INDEX ( cust_name ,policy_number )
PARTITION BY CASE_N(
  policy_expiration_date >= DATE, NO CASE);
```

The output of a SHOW DML request on customer also displays the user-specified partitioning expression as a DATE in ANSI format.

```
SHOW SELECT * FROM customer;
CREATE SET TABLE MOVEDATE.customer, NO FALLBACK, NO BEFORE JOURNAL,
  NO AFTER JOURNAL, CHECKSUM = DEFAULT (
  cust_name CHARACTER(8) CHARACTER SET LATIN NOT CASESPECIFIC,
  policy_number INTEGER,
  policy_expiration_date DATE FORMAT 'YYYY/MM/DD')
PRIMARY INDEX ( cust_name ,policy_number )
PARTITION BY CASE_N(policy_expiration_date >= DATE, NO CASE);
```

The output of a SHOW QUALIFIED DML request on customer displays the partitioning expression with CURRENT_DATE replaced by the resolved date, which is 2010-04-17 in ANSI format.

```
SHOW QUALIFIED SELECT * FROM customer;
CREATE SET TABLE movedate.customer ,NO FALLBACK,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT
  (
    cust_name CHAR(8) CHARACTER SET LATIN NOT CASESPECIFIC,
    policy_number INTEGER,
    policy_expiration_date DATE FORMAT 'YYYY/MM/DD')
PRIMARY INDEX ( cust_name ,policy_number )
PARTITION BY CASE_N(
  policy_expiration_date >= DATE '2010-04-17', NO CASE);
```

Example: CURRENT_DATE Built-In Function in a CASE_N Expression

This example partitions the sales table with sales from the latest 4 quarters in 4 known partitions and all other historical data in another partition. Notice that all new sales data is recorded in the first partition. This form of partitioning allows one current partition with all other partitions as history sales data.

```
CREATE SET TABLE sales, NO FALLBACK (
  store_id  INTEGER,
  amount    DECIMAL(10, 2),
  region    CHARACTER(30),
```

```

    sale_date DATE FORMAT 'YYYY/MM/DD' NOT NULL)
PRIMARY INDEX (store_id)
PARTITION BY CASE_N(sale_date >= CURRENT_DATE /*latest data*/,
    sale_date < CURRENT_DATE
    AND sale_date >= CURRENT_DATE - INTERVAL '3'
        MONTH,/*previous quarter*/
    sale_date < CURRENT_DATE - INTERVAL '3' MONTH
    AND sale_date >= CURRENT_DATE - INTERVAL '6' MONTH,
    sale_date < CURRENT_DATE - INTERVAL '6' MONTH
    AND sale_date >= CURRENT_DATE - INTERVAL '9' MONTH,
    NO CASE /* older than 9 months data */);

```

Example: Quarterly Partitioning Using CURRENT_DATE in a RANGE_N Expression

This example partitions sales , with each quarter as a separate partition with CURRENT_DATE using RANGE_N. With this kind of partitioning, you can resolve CURRENT_DATE to a new date each quarter.

This CREATE TABLE request creates a PPI table with growing partitions at the end and shrinking partitions from the beginning. In this case, Vantage removes data from the partitions that are deleted because of the changed value for CURRENT_DATE when you reconcile the partitioning.

```

CREATE SET TABLE sales, NO FALLBACK (
    store_ID  INTEGER,
    amount    DECIMAL(10, 2),
    region     CHARACTER(30),
    sale_date DATE FORMAT 'YYYY/MM/DD' NOT NULL)
PRIMARY INDEX (store_ID)
PARTITION BY RANGE_N(sale_date BETWEEN CURRENT_DATE - INTERVAL '9'
    MONTH
    AND CURRENT_DATE + INTERVAL '3'
    MONTH - INTERVAL '1' DAY
    EACH INTERVAL '3' MONTH);

```

Example: Historical and Current Partitioning Using CURRENT_DATE in a CASE_N Expression

This example partitions a stock table into historical and current partitions.

```

CREATE TABLE movedate.stock (
    stock_name    CHARACTER(8),
    stock_code     INTEGER,

```

```

    stock_price          DECIMAL(10,6)
    stock_pricing_time   TIMESTAMP(6) WITH TIME ZONE)
PRIMARY INDEX (stock_code, stock_pricing_time)
PARTITION BY CASE_N(stock_pricing_time >= CURRENT_TIMESTAMP,
                    NO CASE);

```

The output of a SHOW TABLE request on stock displays the user-specified partitioning expression as a **TIMESTAMP**.

```

SHOW TABLE movedate.stock;
CREATE SET TABLE movedate.stock, NO FALLBACK,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT (
        stock_name          CHARACTER(8) CHARACTER SET LATIN
                           NOT CASESPECIFIC,
        stock_code          INTEGER,
        stock_price          DECIMAL(10,6)
        stock_pricing_time   TIMESTAMP(6) WITH TIME ZONE)
PRIMARY INDEX (stock_code ,stock_pricing_time)
PARTITION BY CASE_N(stock_pricing_time >= CURRENT_TIMESTAMP(6),
                    NO CASE);

```

Example: Partitioning Expression With **TIMESTAMP** Data Type

Assume that column *tsz* has a **TIMESTAMP(0) WITH TIME ZONE** data type and the current session time zone is **INTERVAL '-07:00' HOUR TO MINUTE** when the table that contains column *tsz* is created with the following **PARTITION BY** clause. The **RANGE_N** function has a **BIGINT** data type.

```

PARTITION BY RANGE_N(tsz BETWEEN TIMESTAMP '2003-01-01 00:00:00.000000'
                      AND      TIMESTAMP '2009-12-31 23:59:59.999999'
                      EACH INTERVAL '1' DAY)

```

Vantage implicitly rewrites this partitioning expression to include the time zone in the timestamp literals as follows.

```

RANGE_N(tsz BETWEEN TIMESTAMP '2003-01-01 00:00:00.000000-07:00'
      AND      TIMESTAMP '2009-12-31 23:59:59.999999-07:00'
      EACH INTERVAL '1' DAY)

```


Example: Partitioning Expression That Specifies AT LOCAL Date

Assume that column *ts* has a `TIMESTAMP(0)` without time zone data type and the current session time zone is `INTERVAL '04:00' HOUR TO MINUTE` when the table that contains column *ts* is created with the following `PARTITION BY` clause. The `RANGE_N` function has an `INTEGER` data type.

```
PARTITION BY RANGE_N(CAST(ts AS DATE AT LOCAL)
                     BETWEEN DATE '2003-01-01'
                     AND      DATE '2009-12-31'
                     EACH INTERVAL '1' MONTH)
```

Vantage implicitly rewrites this partitioning expression, replacing `LOCAL` with a specific time zone as follows.

```
PARTITION BY RANGE_N(CAST(ts AS DATE AT '04:00')
                     BETWEEN DATE '2003-01-01'
                     AND DATE '2009-12-31'
                     EACH INTERVAL '1' MONTH)
```

Example: Grouping Columns in the Column List of a Column-Partitioned Table Rather Than in the Partitioning Expression

This example defines a column-partitioned table with two column partitions. The first column partition contains 2 columns (*c1* and *c2*) and has system-determined `COLUMN` format with autocompression. The second column partition has 1 column (*c3*) and has system-determined `COLUMN` format with no autocompression.

```
CREATE MULTISET TABLE cr (
  (c1 INTEGER,
   c2 INTEGER),
  (c3 VARCHAR(5000)) NO AUTO COMPRESS)
NO PRIMARY INDEX
PARTITION BY COLUMN;
```

Example: Column-Partitioned Table With Default Autocompression

This example defines a column-partitioned table where each column is in its own partition with system-determined `COLUMN` format except for column partition *o_comment*, which has user-specified `ROW` format and is explicitly defined not to have autocompression.

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_shippriority  INTEGER,
  ROW(o_comment   VARCHAR(79)) NO AUTO COMPRESS)
PARTITION BY COLUMN,
UNIQUE INDEX (o_orderkey);
```

Example: Columns Defined Using Different Types of NUMBER Types

This example creates a table whose columns each use a different form of the NUMBER data type.

```
CREATE TABLE num_tab (
  n1 NUMBER(*,3),
  n2 NUMBER,
  n3 NUMBER(*),
  n4 NUMBER(5,1),
  n5 NUMBER(9));
```

Example: UDT Primary and Secondary Index Examples

This example set demonstrates how you can create a table that specifies a UDT column in a primary or secondary index or both.

First define the UDTs that are used for the example columns.

UDT *tbl_integer* is a distinct UDT based on the INTEGER data type.

```
CREATE TYPE tbl_integer AS INTEGER FINAL;
```

UDT *tbl_char50* is also a distinct UDT and is based on the CHARACTER data type.

```
CREATE TYPE tbl_char50 AS CHARACTER(50) FINAL;
```

EXSP_STRUCTURE_LOB is a structured UDT based on the INTEGER and CLOB data types.

```
CREATE TYPE EXSP_STRUCTURE_LOB AS (
  age INTEGER,
  c1  CLOB(512),
```

```

    c2 CLOB(512),
    cnt INTEGER,
    c3 CLOB(512))
NOT FINAL
CONSTRUCTOR METHOD exsp_structure_lob(
    file1 VARCHAR(20),
    file2 VARCHAR(20),
    file3 VARCHAR(20))
RETURNS exsp_structure_lob
SPECIFIC lobconstructor
SELF AS RESULT
NO SQL
PARAMETER STYLE TD_GENERAL
DETERMINISTIC
LANGUAGE C;

```

The first example table defines its UPI on the distinct UDT column *id*.

```

CREATE TABLE table_1 (
    id          tbl_integer,
    emp_name tbl_char50)
UNIQUE PRIMARY INDEX(id);

```

The second example table defines its NUPI on the distinct UDT column *id*.

```

CREATE TABLE table_2 (
    id          tbl_integer,
    emp_name tbl_char50)
PRIMARY INDEX(id);

```

The third example table defines a USI on the distinct UDT column *emp_name*.

```

CREATE TABLE table_3 (
    id          tbl_integer,
    emp_name tbl_char50)
UNIQUE INDEX(emp_name);

```

The fourth example table defines a NUSI on the distinct UDT column *emp_name*.

```

CREATE TABLE table_4 (
    id          tbl_integer,
    emp_name    tbl_char50,

```

```
start_date DATE)
INDEX(emp_name);
```

The fifth example table defines a composite NUSI on the distinct UDT column *emp_name* and the DATE column *start_date*.

```
CREATE TABLE table_5 (
  id          tbl_integer,
  emp_name    tbl_char50,
  start_date  DATE)
INDEX(emp_name, start_date);
```

Example: Specifying the Same Column in Different Ways

This example shows several ways to specify the UDT circle for columns *cir1* and *cir2*.

```
CREATE TABLE tab_1 (
  c1    INTEGER,
  cir_1 circle,
  cir_2 circle NOT NULL);
```

This example specifies the UDT ellipse for column *y* and the UDT circle for column *z*.

```
CREATE TABLE tab_2 (
  x CHARACTER(10),
  y SYSUDTLIB.ellipse,
  z SYSUDTLIB.circle FORMAT 'X(15)' );
```

Example: Create Tables with ST_GEOMETRY Data Type Columns

This example creates a table with an ST_GEOMETRY column that is non-LOB.

```
CREATE TABLE geoTable1(id INTEGER,
                        geo1 ST_GEOMETRY(64000));
```

This example creates a table with the following types of ST_GEOMETRY columns: non-LOB, LOB, and LOB with a specified inline length.

```
CREATE TABLE geoTable2(id INTEGER,
                        geo1 ST_GEOMETRY(100),
```

```
geo2 ST_GEOMETRY,
geo3 ST_GEOMETRY INLINE LENGTH 30000);
```

This example uses a different syntax to create the same structure as geoTable2.

```
CREATE TABLE geoTable3(id INTEGER,
                        geo1 ST_GEOMETRY(100) INLINE LENGTH 100,
                        geo2 ST_GEOMETRY,
                        geo3 ST_GEOMETRY INLINE LENGTH 30000);
```

This example creates a table with the following types of ST_GEOMETRY columns: non-LOB and LOB with specified inline lengths.

```
CREATE TABLE geoTable4(id INTEGER,
                        geo1 ST_GEOMETRY(30000) INLINE LENGTH 30000,
                        geo2 ST_GEOMETRY INLINE LENGTH 100);
```

This example creates a table with an ST_GEOMETRY column that is an LOB with a specified inline length.

```
CREATE TABLE geoTable5(id INTEGER,
                        geo1 ST_GEOMETRY(64000) INLINE LENGTH 100);
```

Example: Create Tables with XML Data Type Columns

This example creates a table with an XML column that is non-LOB.

```
CREATE TABLE xmlTable1(id INTEGER,
                        xml1 XML(64000));
```

This example creates a table with the following types of XML columns: non-LOB, LOB, and LOB with a specified inline length.

```
CREATE TABLE xmlTable2(id INTEGER,
                        xml1 XML(100),
                        xml2 XML,
                        xml3 XML INLINE LENGTH 30000);
```

This example uses a different syntax to create the same structure as xmlTable2, that is, a table with the following types of XML columns: non-LOB, LOB, and LOB with a specified inline length.

```
CREATE TABLE xmlTable3(id INTEGER,
                        xml1 XML(100) INLINE LENGTH 100,
```

```

        xml2 XML,
        xml3 XML INLINE LENGTH 30000);

```

This example creates a table with the following types of XML columns: non-LOB and LOB with specified inline lengths. When the specified maximum length is equal to the inline length, the XML data type is a non-LOB type.

```

CREATE TABLE xmlTable4(id INTEGER,
        xml1 XML(30000) INLINE LENGTH 30000,
        xml2 XML INLINE LENGTH 100);

```

This example creates a table with an XML column that is an LOB with a specified inline length.

```

CREATE TABLE xmlTable5(id INTEGER,
        xml1 XML(64000) INLINE LENGTH 100);

```

Example: Create Tables with JSON Data Type Columns

This example creates a table with a JASON column that is non-LOB.

```

CREATE TABLE jsonTable1(id INTEGER,
        jsn1 JSON(64000) CHARACTER SET LATIN);

```

This example creates a table with non-LOB and LOB JASON columns.

```

CREATE TABLE jsonTable2(id INTEGER,
        jsn1 JSON(1000) CHARACTER SET LATIN,
        jsn2 JSON INLINE LENGTH 30000 CHARACTER SET LATIN);

```

This example uses a different syntax to create the same structure as jsonTable2, a table with non-LOB and LOB JASON columns.

```

CREATE TABLE jsonTable3(id INTEGER,
        jsn1 JSON(1000) INLINE LENGTH 1000 CHARACTER SET LATIN,
        jsn2 JSON INLINE LENGTH 30000 CHARACTER SET LATIN);

```

This example creates a table with the following types of JSON columns: non-LOB and LOB with specified inline lengths. When the specified maximum length is equal to the inline length, the CDT is a non-LOB type.

```
CREATE TABLE jsonTable4(id INTEGER,
                        jsn1 JSON(30000) INLINE LENGTH 30000,
                        jsn2 JSON INLINE LENGTH 100);
```

This example creates a table with an JSON column that is an LOB with a specified inline length.

```
CREATE TABLE jsonTable5(id INTEGER,
                        jsn1 JSON(64000) INLINE LENGTH 100);
```

This example creates a table with an JSON column that is non-LOB with a specified inline length and a Binary JSON storage format.

```
CREATE TABLE jsonTable6(id INTEGER,
                        jsn1 JSON(64000) INLINE LENGTH 64000 STORAGE FORMAT BJSON);
```

This example creates a table with an JSON column that is non-LOB with a specified inline length and a Universal Binary JSON storage format.

```
CREATE TABLE jsonTable7(id INTEGER,
                        jsn1 JSON(64000) INLINE LENGTH 64000 STORAGE FORMAT UBJSON);
```

Example: Creating a Table With Row-Level Security Constraints

In this example, *classification_level* and *classification_category* are the names of previously defined constraints. The create text for the *classification_level* and *classification_category* constraint objects looks like the following.

```
CREATE CONSTRAINT classification_level SMALLINT, NOT NULL,
VALUES (top_secret:4, secret:3, confidential:2, unclassified:1),
INSERT SYSLIB.insert_level,
UPDATE SYSLIB.update_level,
DELETE SYSLIB.delete_level,
SELECT SYSLIB.read_level ;
CREATE CONSTRAINT classification_category BYTE(8),
VALUES (nato:1, united_states:2, canada:3, united_kingdom:4,
        france:5, norway:6, russia:7),
INSERT SYSLIB.insert_category,
UPDATE SYSLIB.update_category,
DELETE SYSLIB.delete_category,
SELECT SYSLIB.read_category ;
```

Vantage implicitly adds column definitions for the row-level constraint columns *classification_level* and *classification_category* to the definition of *table_1_rls_constraints* when it creates the table. The function definitions specified in the constraint objects are executed as row-level security controls on SQL requests that access the rows of *table_1_rls_constraints*.

```
CREATE TABLE table_1_rls_constraints (
  column_1 INTEGER,
  column_2 CHARACTER(30),
  classification_level CONSTRAINT,
  classification_category CONSTRAINT)
UNIQUE PRIMARY INDEX(col1);
```

The create text for the *group_membership* constraint object looks like this. This is a hierarchical constraint that defines four value name:value code pairs.

```
CREATE CONSTRAINT group_membership SMALLINT, NOT NULL,
VALUES (exec:100, manager:90, clerk:50, peon:25),
INSERT SYSLIB.ins_grp_member,
SELECT SYSLIB.rd_grp_member ;
```

In this table definition, Vantage implicitly adds a constraint column named *group_membership* to the *emp_record* table when it creates that table. The *group_membership* column contains the data for the row-level constraint.

```
CREATE TABLE emp_record (
  emp_name VARCHAR(30),
  emp_number INTEGER,
  salary INTEGER,
  group_membership CONSTRAINT)
UNIQUE PRIMARY INDEX(emp_name);
```

After you create *emp_table*, you can submit a SELECT request that retrieves the data in the *group_membership* column by specifying it in the select list for the query. A SELECT request on table *emp_record* that includes *group_membership* in its select list might look something like this.

```
SELECT emp_name, group_membership
FROM emp_record
WHERE group_membership=90;
```

For a user with the appropriate row-level security credentials, this request returns the *emp_name* and *group_membership* code for all managers. Note that you cannot specify a value name as the search condition. Instead, you must specify a value code. In this example, the value code 90 represents the value name *manager*.

For a user who does not have the row-level security credentials to read group_membership 90, this request returns 0 rows.

Note:

You can also retrieve this information from *DBC.AsgdSecConstraints*.

Related Information

- *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146
- *Teradata Vantage™ - Database Design*, B035-1094
- *Teradata® Unity™ User Guide*, B035-2520

CREATE TABLE (Queue Table Form)

Creates a queue table.

You cannot create a NoPI or temporal queue table.

Queue tables do not support row-level security constraint columns.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Other SQL dialects support similar non-ANSI standard statements with names such as the following:

- CREATE QUEUE

Required Privileges

You must have the CREATE TABLE privilege on the database or user in which the queue table is created.

Privileges Granted Automatically

The creator receives all the following privileges on the newly created queue table.

- CREATE TRIGGER (WITH GRANT OPTION)
- DELETE
- DROP TABLE
- DUMP
- INDEX
- INSERT
- REFERENCES
- RESTORE
- SELECT
- UPDATE

CREATE TABLE Syntax (Queue Table Form)

```
{ CREATE table_kind | CT } table_specification,
  QUEUE [ table_option [,...] ]
  ( QITS_specification )
  [ index_specification [[,]...] ] [;]
```

table_kind

```
[ SET | MULTISET ]
```

table_specification

```
[ database_name. | user_name. ] table_name
```

table_option

```
{ MAP = map_name |
  [NO] FALLBACK [ PROTECTION ] |
  NO JOURNAL |
  FREESPACE = integer [ PERCENT ] |
  data_block_size |
  blockcompression |
  CHECKSUM = { ON | OFF | DEFAULT }
}
```

QITS_specification

```
QITS_column_name TIMESTAMP [ (6) ] NOT NULL DEFAULT
  CURRENT_TIMESTAMP [ (6) ]
  [ data_type_attribute [...] |
    [ CONSTRAINT name ] CHECK ( boolean_condition )
  ]
QITS_attribute [,...]
```

index_specification

```
{ [ UNIQUE ] INDEX [ index_name ] ( index_column_name [,...] ) |

  [ UNIQUE ] PRIMARY INDEX [ index_name ] ( primary_index_column
```

```
[,...] ) |  
  
INDEX [ index_name ] ( index_column_name [...] )  
ORDER BY [ VALUES | HASH ] ( order_column_name )  
}
```

data_block_size

```
{ DATABLOCKSIZE = integer [ BYTE[S] | KBYTE[S] | KILOBYTE[S] ] |  
  [ MINIMUM | MAXIMUM ] DATABLOCKSIZE }
```

blockcompression

```
BLOCKCOMPRESSION = { AUTOTEMP | MANUAL | ALWAYS | NEVER | DEFAULT }  
[ , BLOCKCOMPRESSIONALGORITHM = { ZLIB | ELZS_H | DEFAULT } ]  
[ , BLOCKCOMPRESSIONLEVEL = { value | DEFAULT } ]
```

data_type_attribute

```
{ { UPPERCASE | UC } |  
  [NOT] { CASESPECIFIC | CS } |  
  FORMAT quotestring |  
  TITLE quotestring |  
  NAMED name |  
  DEFAULT { number | USER | DATE | TIME | NULL } |  
  WITH DEFAULT |  
  CHARACTER SET server_character_set  
}
```

QITS_attribute

```
{ column_name data_type column_attribute [...] |  
  column_constraint_attributes  
}
```

column_attribute

```
{ data_type_attribute [...] |  
  [NOT] NULL |  
  compression_attributes |
```

```

column_constraint_attributes |
column_identity_attributes |
}

```

column_constraint_attributes

```

[ CONSTRAINT name ]
  { { UNIQUE | PRIMARY KEY } ( column_name [,...] ) | CHECK
    (boolean_condition ) }

```

compression_attributes

```

COMPRESS [ { constant | NULL } | ( { constant | NULL } [,...] ) ]

```

column_identity_attributes

```

GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ (
identity_attribute [...] ) ]

```

identity_attribute

```

{ START WITH |
  INCREMENT BY |
  [NO] MINVALUE |
  [NO] MAXVALUE |
  [NO] CYCLE
}

```

CREATE TABLE Syntax Elements (Queue Table Form)***table_kind***

Duplicate row control.

If you do not specify SET or MULTiset, the default table kind depends on the session mode.

- In ANSI session mode, the default table kind is MULTiset. Duplicate rows are allowed.
- In Teradata session mode, the default table kind is SET. Duplicate rows are not allowed.

MULTISET

Duplicate rows are allowed.

If there are uniqueness constraints on any column or set of columns in the table definition, the table cannot have duplicate rows even if you specify MULTISET.

Some client utilities have restrictions with respect to MULTISET tables. See *CREATE TABLE* in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

SET

Duplicate rows are not allowed.

If the only column in the table is its QITS column, then you cannot specify SET.

In this case, the table must be defined, either implicitly or explicitly, as MULTISET.

See [QITS_column_name](#).

table_specification

Specify the table name with these options.

database_name

Name of the containing database if different from the current database.

user_name

Name of the containing user if different from the current user.

table_name

Name of the new queue table. For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141. If the name is not fully qualified, then the system assigns the name of the default database for the current session.

*table_option***MAP**

You can specify an existing contiguous map for the queue table.

map_name

Name of an existing contiguous map.

For a queue table, you cannot specify a sparse map.

You cannot specify a nonexistent map, TD_DataDictionaryMap, or TD_GlobalMap.

If the name begins with a digit or includes spaces, you must enclose the name in quotation marks.

Usage Notes

User, Database, or Profile DEFAULT MAP OVERRIDE ON ERROR Option

If the map you specify is not valid for any reason, (such as it does not exist, has not been granted to you, or is not in the same secure zone), Vantage either substitutes a default map or returns an error, subject to the values of the DEFAULT MAP settings for your PROFILE, USER, or DATABASE.

Default Map for a CREATE TABLE (Queue Table Form) Statement

If you do not specify the MAP option when creating a queue table, the system determines a default map for the table in following order of precedence:

- If the immediate owner is not the creator:
 - Default map, if defined, for the profile of the immediate owner.
 - Default map, if defined, for the immediate owner.
 - System-default map.
- Default map, if defined, for the profile of the creator.
- Default map, if defined, for the creator.
- System-default map.

Example

For a queue table, you must specify a contiguous map. You cannot specify a sparse map.

```
CREATE TABLE qtbl_1, QUEUE,
  MAP=TD_Map1 (
col_1 TIMESTAMP(6) NOT NULL DEFAULT CURRENT_TIMESTAMP(6),
col_2 INTEGER,
col_3 INTEGER)
PRIMARY INDEX (col_2, col_3);
```

FALLBACK

Duplicate copy protection for the table.

When you specify FALLBACK, Vantage creates and stores duplicate copies of rows in the table.

The default for this option is established by a CREATE DATABASE, CREATE USER, or MODIFY USER statement for the database or user in which the table is to be created.

When an unrecoverable bit error occurs, the system reads the fallback copy of the data. Support for unrecoverable bit error recovery using fallback is limited to the following cases:

- Requests that do not attempt to modify data in the bad data block
- Primary subtable data blocks
- Reading the fallback data in place of the primary data. The system does not attempt to fix the bad data block.

NO

No duplicate copy protection for the table.

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

PROTECTION

Optional word that can be specified after the FALLBACK keyword.

FREESPACE

Percent of free space that will remain on a cylinder during loading operations.

integer

Specifies the value of the percent freespace attribute. If the specified value does not fall within the allowable range (0 to 75 percent), an error message is generated.

PERCENT

An optional keyword to indicate percentage.

data_block_size

Maximum data block size for blocks that contain multiple rows as the value *numeric*.

If DATABLOCKSIZE is not specified, then data blocks default to the sizes set in the PermDBSize and JournalDBSize fields of the DBS Control Record.

- The default value for a PERM data block is specified using the PermDBSize parameter of the DBS Control record.
- The default value for transient and permanent journal data blocks is specified using the JournalDBSize parameter of the DBS Control Record.

numeric

Value for data block size.

BYTES

BYTES can be abbreviated as BYTE.

KBYTES**KILOBYTES**

KILOBYTES can be abbreviated as KBYTE.

MINIMUM

The smallest data block size for this table.

MINIMUM DATABLOCKSIZE sets the maximum data block size for blocks that contain multiple rows to the minimum value of 21,504 bytes (42 sectors) for systems with large cylinders or 9,216 bytes (18 sectors) for systems without large cylinders.

You can abbreviate MINIMUM as MIN.

For details about minimum data block sizes, see DATABLOCKSIZE of CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

MAXIMUM

The largest data block size for this table.

Systems that do not support block sizes 128KB and larger use a value of 127.5KB.

Systems that allow blocks larger than 127.5KB use a maximum of 1,048,064 bytes (2047 sectors) for large cylinders or 262,144 bytes (512 sectors) for small cylinders.

The value can be expressed either as a decimal number or integer number or using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

You can abbreviate MAXIMUM as MAX.

DEFAULT

The default data block size for this table.

If you specify DEFAULT or do not specify a value, Vantage uses the system-wide default data block size specified by the DBS Control setting PermDBSize. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

blockcompression

Table data is compressed at the block level.

You cannot specify this option to modify the definitions of global temporary tables or volatile tables.

For details, see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

AUTOTEMP

The file system determines the block-level compression setting for the table based on its Teradata Virtual Storage temperature. The definitions of the various thresholds are determined by the DBS Control setting TempBLCThresh. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

DEFAULT

The table uses the block-level compression setting in the DBS Control setting DefaultTableMode. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

MANUAL

Block level compression is applied based on the default for the table at the time the table is created. Tables can be compressed or uncompressed at any time after loading by using the Ferret COMPRESS and UNCOMPRESS commands. Data inserted into the existing table inherits the current compression status of the table at the time the data is inserted.

ALWAYS

The table and its subtables are always block-level compressed, even if a query band or the applicable DBS Control block-level compression settings indicate otherwise. The DBS Control field BlockLevelCompression must be enabled.

Tables can be compressed at any time after loading by using the Ferret COMPRESS command. This can be useful if this BLOCKCOMPRESSION option was set after the table was already populated.

NEVER

The table and its subtables are not block-level compressed, even if a query band or the applicable DBS Control block compression settings indicate otherwise.

BLOCKCOMPRESSIONALGORITHM

Specifies the algorithm to use for block-level compression (BLC).

This option only applies when the effective BLOCKCOMPRESSION is MANUAL, AUTOTEMP, or ALWAYS.

ZLIB

Block-level compression using the zlib software algorithm.

ELZS_H

Block-level compression using a hardware compression engine board in every system node.

DEFAULT

Uses the setting of the DBS Control field CompressionAlgorithm. See *Teradata Vantage™ - Database Utilities*, B035-1102.

BLOCKCOMPRESSIONLEVEL

Specifies a value to indicate a preference for compression speed or compression effectiveness. This option only applies when the BLOCKCOMPRESSIONALGORITHM option is set to ZLIB. Although this option is accepted in combination with BLOCKCOMPRESSIONALGORITHM = ELZS_H, this option is ignored. If BLOCKCOMPRESSIONALGORITHM is altered to ZLIB, this option takes effect.

value

Integer from 1 through 9, where 1 specifies the least processor-intensive compression speed with the lowest compression ratio and 9 specifies the most processor-intensive compression speed with highest compression ratio.

DEFAULT

The table uses the compression level setting of the DBS Control field CompressionLevel. See *Teradata Vantage™ - Database Utilities*, B035-1102.

CHECKSUM

A table-specific disk I/O integrity checksum. Specifies the integrity checking level.

The checksum setting applies to primary data rows, fallback data rows, and all secondary index rows for the table.

If you do not specify a value, then the system assumes the system-wide default value for this table type. This is equivalent to specifying DEFAULT.

If you are changing the checksum for this table to the system-wide default value, then specify DEFAULT.

ON

Calculate checksums using the entire disk block. Sample 100% of the disk blocks to generate a checksum.

OFF

Disables checksum disk I/O integrity checks.

DEFAULT

The default setting is the current DBS Control checksum setting specified for this table type.

QITS_specification

You can specify these options for a column.

*QITS_column_name***The Query Insertion Timestamp (QITS) column**

The first column defined for any queue table must be a QITS column. Vantage uses the QITS column to maintain the FIFO ordering of rows in the queue table. Each queue table has only one QITS column, and it must be defined with the following attributes:

```
column_name TIMESTAMP(6) [WITH TIME ZONE] NOT NULL
        DEFAULT CURRENT_TIMESTAMP(6)
```

The precision specification is optional for the TIMESTAMP data type specification and its DEFAULT attribute, but you cannot define either with a precision value other than 6.

The QITS column cannot be defined as any of the following:

- UNIQUE PRIMARY INDEX
- UNIQUE
- PRIMARY KEY
- Unique secondary index (see [CREATE INDEX](#))
- Identity column

The QITS column can be the NUPI for a table, but you should avoid following that practice. Be aware that if you do not define an explicit primary index, primary key, or uniquely constrained column in the table, then the QITS column becomes its primary index by default because it is the first column defined for the table.

You might find it useful to find additional queue management columns for functions such as message identification or queue sequencing.

column_name

Specifies the name of one or more non-QITS columns, in the order they are to be defined for the table.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

You can define up to 2,048 columns, including the mandatory QITS column, for a queue table.

data type

You must specify a single data type for each *column_name*.

Queue tables cannot contain columns with BLOB or CLOB data types.

For information on data types, data type attributes, and converting between data types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

If you do not specify explicit formatting, a column assumes the default format for the data type, which can be specified by a custom data formatting specification (SDF) defined by the *tdlocaledef* utility. See *Teradata Vantage™ - Database Utilities*, B035-1102. Explicit formatting applies to the parsing and retrieval of character strings.

column attributes

One or more data definition phrases that define data for the column.

You cannot specify a character server data set of KANJI1. Otherwise, Vantage returns an error to the requestor.

Column data attribute specifications are optional. If you specify attributes for a column, you should define its data type prior to defining the attributes.

Example: Using SELECT AND CONSUME on Queue Tables with JSON, ST_GEOMETRY, or XML Columns

You can create queue tables with inline JSON, ST_GEOMETRY, or XML Columns, for example, JSON(1000). However, to retrieve data from this table using SELECT AND CONSUME, you must use non-LOB transforms.

Below is the table definition for this example.

```
CREATE SET TABLE qt1, QUEUE
(
    QITS TIMESTAMP(6) NOT NULL DEFAULT CURRENT_TIMESTAMP(6),
    col1 INTEGER,
    jsn JSON(64000) CHARACTER SET LATIN
)
PRIMARY INDEX (col1);
```

Then, we insert a row of data.

```
INSERT INTO qt1 VALUES (current_timestamp, 1, '{"a":123}');
```

Then, we create a user with non-LOB transforms for JSON, ST_GEOMETRY, and XML data.

```
CREATE USER User1 AS PERM=1e8 * (HASHAMP () + 1), PASSWORD=secret,
  TRANSFORM ( JSON CHARACTER SET LATIN=TD_JSON_VARCHAR,
    ST_GEOMETRY=TD_GEO_VARCHAR,
    XML=TD_XML_VARCHAR);
```

Now, log in as User1 to retrieve the data.

```
SELECT AND CONSUME TOP 1 col1, jsn FROM qt1;
```

```
*** Query completed. One row found. 2 columns returned.
*** Total elapsed time was 1 second.
```

```
col1 jsn
```

```
-----
1 {"a":123}
```

compression_attributes

These attributes are Teradata extensions to the ANSI SQL:2011 specification.

COMPRESS

A specified set of distinct values in a column that is to be compressed to zero space.

See *Teradata Vantage™ - Database Design*, B035-1094 for a detailed description of multivalue compression.

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for more information about the COMPRESS attribute.

constant

Specifies that nulls and the value are compressed.

Compressed constants are typically string literal. For numeric columns, constants have a numeric data type.

The most space a numeric value can occupy is eight bytes.

Using COMPRESS on fixed-length character data can save space depending on the percentage of rows for which the compressed value is assigned.

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for information about limits for this value.

NULL

Specifies that nulls are compressed.

You can only specify NULL once per column.

column_attribute

column_constraint_attributes

A column listed as UNIQUE or PRIMARY KEY must be declared NOT NULL.

CONSTRAINT *name*

Name for a named constraint.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Constraints are not permitted for volatile tables.

UNIQUE

No two rows in the table can have the same value in the field. The system uses a unique primary or secondary index to enforce this constraint.

This constraint can only be specified on a single column. To specify UNIQUE on multiple columns, use the UNIQUE definition clause.

You cannot specify a UNIQUE constraint for a QITS column (see [QITS_column_name](#)).

Vantage implicitly creates a unique index on any column specified as UNIQUE.

If all of the following conditions are true, then the implicitly defined index is a unique primary index.

- No explicit primary index is specified
- No explicit primary key is specified
- This is the first unique constraint defined for the table

If none of the conditions listed previously is true, then the implicitly defined index is a unique secondary index.

PRIMARY KEY

No two rows in the table can have the same value for the defined column set. The system uses a unique primary or secondary index to enforce this constraint.

Only one primary key can be specified per table. To specify candidate primary keys for referential integrity relationships with other tables, use the UNIQUE definition clause.

You cannot specify a PRIMARY KEY constraint for a QITS column (see [QITS_column_name](#)).

The system implicitly creates a unique index on any column set specified as PRIMARY KEY if it is not also defined to be the primary index for the table.

column_name

A column in the column set to be used as the primary key or as unique.

You cannot specify the QITS column to be either UNIQUE or a PRIMARY KEY.

See [QITS_column_name](#).

If more than one *column_name* is specified, the PRIMARY KEY or UNIQUE column set is based on the combined values of the column named.

A maximum of 64 columns can be specified for a primary key.

Columns listed in a UNIQUE or PRIMARY KEY table constraint must all be defined as NOT NULL.

CHECK (*boolean condition*)

A Boolean conditional expression, including scalar comparison conditions using, for example, any of the following:

- =
- <>
- >
- >=

Also see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

When a CHECK constraint is part of the column definition, then a search condition cannot reference any columns other than the one being defined, nor are aggregate functions allowed.

Also see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

column_identity_attributes

The identity column is ANSI SQL:2011 compliant.

The QITS column cannot also be defined to be an identity column.

ALWAYS

Specifies that identity column values are always system-generated.

You cannot insert values into, nor can you update, an identity column defined as GENERATED ALWAYS AS IDENTITY.

BY DEFAULT

Identity column values can be system-generated or user-inserted, depending on the circumstance.

- If you try to insert a null into an identity column, Vantage generates an identity column value instead.
- If you try to insert a value into an identity column, Vantage inserts the value you specify into the identity column.

Example: Creating a Queue Table Defined with an Identity Column

This example creates a queue table defined with an identity column for the queue sequence number, which is named QSN. Because the primary index for the table is simple, defined only on the QITS column, it must be defined as a NUPI:

```
CREATE TABLE qtbl_4, QUEUE (
  qits  TIMESTAMP(6) NOT NULL DEFAULT CURRENT_TIMESTAMP(6),
  qsn   INTEGER GENERATED ALWAYS AS IDENTITY (CYCLE),
  col_3 INTEGER)
PRIMARY INDEX (qits);
```

The value of a QSN is expected to be unique under normal circumstances because queue table rows should be consumed at a high enough rate that the queue (which is modeled as one table row per enqueued event) does not grow large enough for the QSN value to be reused by completing a cycle through the identity column values.

Example: Creating a Queue Table with Constraints and a UPI

This example creates a queue table with constraints and a UPI to uniquely identify rows externally.

Note that while the definitively non-unique QITS column, col_1_qits, is a component of the unique primary index for the table, that index is composite, and its other component has the attributes NOT NULL and UNIQUE, ensuring that the composite primary index is unique.

```
CREATE SET TABLE qtbl_2, QUEUE, NO FALLBACK (
  col_1_qits TIMESTAMP(6) NOT NULL DEFAULT CURRENT_TIMESTAMP(6),
  col_2      INTEGER NOT NULL UNIQUE,
  col_3      INTEGER, CONSTRAINT check_1 CHECK (col_3 > 0))
UNIQUE PRIMARY INDEX primary_1 (col_1_qits, col_2);
```

Example: Creating a Queue with a UPI and a NUSI

This example creates a queue table with a UPI defined on the queue sequence number identity column, QSN, and a NUSI defined on the queue insertion timestamp, QITS:

```
CREATE TABLE qtbl_5, QUEUE (
  qits  TIMESTAMP(6) NOT NULL DEFAULT CURRENT_TIMESTAMP(6),
```



```

    qsn    INTEGER GENERATED ALWAYS AS IDENTITY (NO CYCLE),
    col_3  INTEGER)
UNIQUE PRIMARY INDEX (qsn)
INDEX qits;

```

With a high rate of consumption of queue table rows, you would typically replace the NO CYCLE option with CYCLE. CYCLE has the advantage of enabling the reuse of QSN numbers while not failing when MAXVALUE is exceeded. The UPI on QSN would then guarantee that, at any point, the value of QSN is unique.

identity_attribute

The identity column parameters are optional and can be specified in any order.

START WITH

Lowest number in the system-generated numeric series for an identity column. The default is 1.

The value can be any exact negative or positive whole number within the range of the data type for the column as long as it is less than MAXVALUE for an incremental series or greater than MINVALUE for a decremental series.

INCREMENT BY

Interval on which to increment system-generated numbers. You can specify a negative interval. The default is 1.

The value can be any whole number less than or equal to the value of DBS Control parameter IdCol Batch Size except 0.

MINVALUE

NO MINVALUE

Minimum value to which a system-generated numeric series can decrement.

The value can be any whole number with an absolute value less than the value specified for START WITH.

The default is the minimum number for the data type defined for the column.

When cycling is not enabled, the sum of the specified values for START WITH and INCREMENT BY must be greater than MINVALUE. If they are not, then the system generates only one number before the minimum limit is exceeded.

- If you specify a positive INCREMENT BY value with CYCLE, renumbering begins from MINVALUE when MAXVALUE is reached.
- If you specify a negative INCREMENT BY value, then MINVALUE, if specified, must be a whole number such that $\text{MINVALUE} \leq \text{START WITH}$.

MINVALUE also cannot be smaller than the minimum value for the data type assigned to the identity column.

- If you specify NO CYCLE, then MINVALUE is not applicable for positive increments.

No warning or error is returned if you specify a MINVALUE with NO CYCLE.

MAXVALUE

NO MAXVALUE

Maximum value to which a system-generated numeric series can increment.

Can be any whole number with a value greater than the value specified for START WITH.

The default is the maximum number for the data type defined for the column.

When cycling is not enabled, the sum of the specified values for START WITH and INCREMENT BY must be less than MAXVALUE. If they are not, then the system generates only one number before the maximum limit is exceeded.

- If you specify a positive INCREMENT BY value, then MAXVALUE, if specified, must be a whole number such that $\text{MAXVALUE} \geq \text{START WITH}$.

MAXVALUE also cannot be larger than the maximum value for the data type assigned to the identity column.

- If you specify a negative INCREMENT BY value with CYCLE, then renumbering begins with MAXVALUE when MINVALUE is reached.
- If you specify NO CYCLE, then MAXVALUE is not applicable for negative increments.

No warning or error is returned if you specify a MAXVALUE with NO CYCLE.

CYCLE

System-generated values can be recycled when their minimum or maximum is reached.

NO CYCLE

System-generated values are not recycled when their minimum or maximum is reached.

The default is NO CYCLE.

index_specification

You can specify these options to define an index.

PRIMARY INDEX

The primary index.

The primary index is used by the hashing algorithm to partition table rows across the AMPs.

Queue tables cannot be NoPI tables nor can they have a partitioned primary index.

If no primary index is specified, then Vantage assigns one implicitly, using the following guidelines to determine which column set is to be defined as the primary index.

- If you do not specify a primary index, but do specify a PRIMARY KEY constraint, then the implicitly defined primary index is the primary key for the table.
- If you do not specify a primary key, but do specify a UNIQUE constraint, then the implicitly defined primary index is the first UNIQUE constraint defined for the table.
- If you do not specify either a PRIMARY KEY constraint or a UNIQUE constraint, then the implicitly defined primary index is the QITS column.

In this case, the primary index is nonunique by default and cannot be defined with a unique constraint or as a USI.

The reason for this restriction is that timestamp values can repeat and therefore cannot be assumed to be unique.

UNIQUE

The named column must be unique.

The primary index and any secondary indexes can be defined to be unique. The only exception is if a queue table has only a QITS column. In this case, the QITS column must also be the primary index, so it cannot be a UPI. This is because timestamp values can repeat and therefore cannot be assumed to be unique (see *Teradata Vantage™ - Data Types and Literals*, B035-1143).

primary_index_column

A column in the column set whose values are to be used as the basis for a primary index.

Columns in the list cannot have a BLOB, CLOB, Period, XML, Geospatial, JSON, or DATASET data type.

If you specify more than one column name, the index is created on the combined values of each column named. A maximum of 64 columns can be specified for an index, and a combined maximum of 32 secondary, hash, and join indexes can be created for one table (a multicolumn NUSI defined with an ORDER BY clause counts as two indexes in this calculation). This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints for nontemporal tables and the single-table join indexes used to implement PRIMARY KEY and UNIQUE constraints for temporal tables.

INDEX, secondary index definition

A keyword used to define secondary indexes for the table. The INDEX list is an extension to ANSI SQL.

UNIQUE

The named column must be unique.

The primary index and any secondary indexes can be defined to be unique. The only exception is if a queue table has only a QITS column. In this case, the QITS column must also be the primary index, so it cannot be a UPI. This is because timestamp values can repeat and therefore cannot be assumed to be unique (see *Teradata Vantage™ - Data Types and Literals*, B035-1143).

index_name

Optional name for the index.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

index_column_name

A column in the column set whose values are to be used as the basis for a secondary index.

Columns in the list cannot have a BLOB, CLOB, Period, XML, Geospatial, JSON, or DATASET data type.

If you specify more than one column name, the index is created on the combined values of each column named. A maximum of 64 columns can be specified for an index, and a combined maximum of 32 secondary, hash, and join indexes can be created for one table (a multicolumn NUSI defined with an ORDER BY clause counts as two indexes in this calculation). This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints for nontemporal tables and the single-table join indexes used to implement PRIMARY KEY and UNIQUE constraints for temporal tables.

INDEX, secondary index definition using an ORDER BY clause

A keyword to introduce a secondary index definition using an ORDER BY clause to order the index on a single column. The INDEX list is an extension to ANSI SQL.

Unlike the indexes created by the UNIQUE and PRIMARY KEY constraint definitions, indexes defined by the index list can include nullable fields.

index_column_name

A column in the column set whose values are to be used as the basis for a primary or secondary index.

Columns in the list cannot have a BLOB, CLOB, Period, XML, Geospatial, JSON, or DATASET data type.

If you specify more than one column name, the index is created on the combined values of each column named. A maximum of 64 columns can be specified for an index, and a combined maximum of 32 secondary, hash, and join indexes can be created for one

table (a multicolumn NUSI defined with an ORDER BY clause counts as two indexes in this calculation). This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints for non-temporal tables and the single-table join indexes used to implement PRIMARY KEY and UNIQUE constraints for temporal tables.

ORDER BY, Index Definition

Row ordering on each AMP by a single NUSI column: either value-ordered or hash-ordered.

You cannot order rows on a BLOB, CLOB, Period, UDT, XML, Geospatial, JSON, or DATASET column.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

Rules for using an ORDER BY clause are shown in the following table:

- If you specify ORDER BY with VALUES, then the ORDER BY *column_name* must be numeric with four bytes or less.
- If you specify ORDER BY without also specifying either HASH or VALUES, then Vantage assumes VALUES by default.
- If you specify ORDER BY with a column name, then the ORDER BY *column_name* must be one of the columns in the INDEX list of columns.

VALUES

Value-ordering for the ORDER BY column.

Select VALUES to optimize queries that return a contiguous range of values, especially for a covering index or a nested join.

HASH

Hash-ordering for the ORDER BY column.

Select HASH to limit hash-ordering to one column, rather than all columns, which is the default.

Hash-ordering a multicolumn NUSI on one of its columns allows the NUSI to participate in a nested join where join conditions involve only that ordering column.

order_column_name

A column in the INDEX list of columns that specifies the sort order to be used.

You can specify the following data types for a value-ordered, four-bytes-or-less *order_column_name*:

- BYTEINT
- DATE
- DECIMAL
- INTEGER

- SMALLINT

Examples

Example: Creating a Simple Queue Table

This example creates a simple queue table.

The first column defined for the table is the Query Insertion Timestamp (QITS) column. This is mandatory for all queue tables.

The QITS column must be defined with a timestamp data type with exactly the same precision and default value as specified in the example. You do not need to specify a precision for the `TIMESTAMP` data type specification because it defaults to 6 when no precision is specified. You cannot explicitly specify a precision other than 6 for the timestamp. This is mandatory for queue tables.

The QITS column is not the (nonunique) primary index for the table.

```
CREATE SET TABLE qtbl_1, QUEUE, FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL (
  col_1 TIMESTAMP(6) NOT NULL DEFAULT CURRENT_TIMESTAMP(6),
  col_2 INTEGER,
  col_3 INTEGER)
PRIMARY INDEX (col_2, col_3);
```

Example: Creating a Multiset Queue Table

This example creates a multiset queue table with constraints defined at the column level:

```
CREATE MULTiset TABLE qtbl_3, QUEUE,
NO BEFORE JOURNAL,
NO AFTER JOURNAL (
  qits TIMESTAMP(6) NOT NULL DEFAULT CURRENT_TIMESTAMP(6),
  col_2 INTEGER,
  col_3 INTEGER CHECK (col_3 > 10))
PRIMARY INDEX (qits);
```

This queue table permits duplicate rows. The system uniquely identifies its rows, including duplicates, by their internally-generated ROWID value.

Related Information

- CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - Database Design*, B035-1094

CREATE GLOBAL TEMPORARY TRACE TABLE

Creates a global temporary trace table to support the UDF- and external SQL procedure-related trace option of the SET SESSION statement. See [SET SESSION FUNCTION TRACE](#).

You cannot partition or define indexes for a global temporary trace table.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the CREATE TABLE privilege on the database or user in which the table is created.

The creator receives all the following privileges on the newly created table.

- DROP TEMPORARY TABLE
- DUMP
- REFERENCES
- RESTORE
- SELECT

Privileges Granted Automatically

None.

CREATE GLOBAL TEMPORARY TRACE TABLE Syntax

```
CREATE GLOBAL TEMPORARY TRACE TABLE [ database_name. | user_name. ] table_name
( proc_ID BYTE(2) , sequence INTEGER [ column_specification [,...] ] )
[ ON COMMIT { DELETE | PRESERVE } ROWS ] [;]
```

column_specification

```
column_name data_type data_type_attribute [...]
```

data_type

```

{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [( fractional_seconds_precision)] [WITH TIME
ZONE] |

  INTERVAL YEAR [( precision)] [TO MONTH] |

  INTERVAL MONTH [( precision)] |

  INTERVAL DAY [( precision)]
    [ TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL HOUR [(precision)]
    [ TO { MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL MINUTE [(precision)] [ TO SECOND
[(fractional_seconds_precision)] ] |

  INTERVAL SECOND [ (precision [, fractional_seconds_precision ] ) ] |

  REAL |

  DOUBLE PRECISION |

  FLOAT [(integer)] |

  NUMBER [( { integer | *} [, integer ]...)] |

  { DECIMAL | NUMERIC } [(integer [, integer ]...)] |

  { CHAR | BYTE | GRAPHIC } [(integer)] |

  { VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [(integer)] |

  LONG VARCHAR |

  LONG VARGRAPHIC
}

```


data_type_attribute

```

{ NOT NULL |
  { UPPERCASE | UC } |
  [NOT] { CASESPECIFIC | CS } |
  FORMAT quotestring |
  TITLE quotestring |
  NAMED name |
  CHARACTER SET server_character_set
}

```

CREATE GLOBAL TEMPORARY TRACE TABLE Syntax Elements***database_name******user_name***

Name of the database or user to contain *table_name*, if other than the current database or user.

table_name

Name of the global temporary trace table.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

If the name is not fully qualified, then the system assigns it to the default database or user for the current session.

***proc_ID* BYTE(2)**

Mandatory first column and data type for all global temporary trace tables.

This column records the number of the AMP on which the UDF that wrote the row is running.

The column can have any name, but it must be declared as BYTE(2).

***sequence* INTEGER**

Mandatory second column and data type for all global temporary trace tables.

This column records a sequence number to indicate the order in which the function trace rows in the table were written for the logged AMP.

The column can have any name, but it must be declared as INTEGER.

column_name

Specifies the name of one or more optional columns, in the order in which they and their attributes are to be defined for the table. Up to 2,048 columns can be defined for a table.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

data_type**data_type_attributes**

One or more data definition phrases that define data for the column.

You must specify a single data type for each *column_name*.

You cannot specify a BLOB, CLOB, UDT, or Period type for any column in a global temporary trace table. You can dump all the predefined attributes of a UDT into a trace table using FNC calls, however (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about how to use FNC calls).

You cannot write Java strings into a global temporary trace table column defined with any of the following data types because the Java language does not support the types:

- GRAPHIC
- VARGRAPHIC
- LONG GRAPHIC

Column data type attribute specifications are optional. If you specify attributes for a column, you should define its data type prior to defining the attributes.

The only valid constraints for global temporary trace table columns are NULL and NOT NULL.

If you specify NOT NULL, but the UDF does not return data, then that column is padded with appropriate values, depending on the data type.

You cannot specify DEFAULT or WITH DEFAULT column attributes for global temporary trace table columns.

If you do not specify explicit formatting, a column assumes the default format for the data type, which can be specified by a custom data formatting specification (SDF) defined by the `tdlocaledef` utility. See *Teradata Vantage™ - Database Utilities*, B035-1102. Explicit formatting applies both to the parsing and to the retrieval of character strings.

You cannot define a temporal column for a global temporary trace table.

Data types and data type attributes are described in detail in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

ON COMMIT

Action to perform when a transaction completes.

PRESERVE ROWS

Preserve the contents of a global temporary or volatile table when a transaction completes.

DELETE ROWS

Delete the contents of a global temporary or volatile table when a transaction completes.

DELETE is the default.

CREATE GLOBAL TEMPORARY TRACE TABLE Examples**Example: Creating a Global Temporary Trace Table**

This example defines the mandatory columns for a global temporary trace table as well as several optional, application-specific, columns. When the transaction that materializes *udf_test* commits, its contents are to be deleted.

```
CREATE GLOBAL TEMPORARY TRACE TABLE udf_test (
  proc_ID      BYTE(2),
  sequence     INTEGER,
  udf_name     CHARACTER(15),
  in_quantity  INTEGER,
  trace_val1   FLOAT,
  trace_text   CHARACTER(30))
ON COMMIT DELETE ROWS;
```

A simple query to retrieve the results recorded in *udf_test* might look like this:

```
SELECT *
FROM udf_test
ORDER BY proc_id ASC, sequence ASC;
```

Example: Creating a Trace Table

The following example defines a simple trace table that has a single-column, variable-length string to capture function output.

This single VARCHAR column approach lends itself to a flexible trace output that can be used by many different functions without having to resort to specific single-purpose trace table column definitions.

```
CREATE GLOBAL TEMPORARY TRACE TABLE udf_test, NO LOG (
  proc_ID      BYTE(2),
  sequence     INTEGER,
  trace_string VARCHAR(256))
ON COMMIT DELETE ROWS;
```

Example: Complete Function Traceback Scenario

The following simple procedure trace example is a complete scenario that shows the various aspects of using global temporary trace tables to debug a procedure.

The following CREATE FUNCTION request, returned by a SHOW FUNCTION request submitting using BTEQ, defines the traceback UDF that was used for this example. You can make the procedure as simple, or as complex, as your application requires. This particular trace UDF is fairly simple.

```
SHOW FUNCTION sptrace;
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
-----

CREATE FUNCTION sptrace (
  p1 VARCHAR(100) CHARACTER SET LATIN)
RETURNS INTEGER
SPECIFIC sptrace
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
NOT DETERMINISTIC
RETURNS NULL ON NULL INPUT
EXTERNAL NAME sptrace
*** Text of DDL statement returned.
-----

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
/*
  Install in SYSLIB for general use and change to NOT PROTECTED mode
  ALTER FUNCTION sptrace EXECUTE NOT PROTECTED;
*/
/* Assumes that the following trace table has been created */
/*
CREATE MULTISET GLOBAL TEMPORARY TRACE TABLE tracetst,
NO FALLBACK, NO LOG (
  proc_id  BYTE(2),
  sequence INTEGER,
  trace_str VARCHAR(100))
ON COMMIT PRESERVE ROWS;
Turn on tracing with following SQL (decide what your options mean):
SET SESSION FUNCTION TRACE USING 'T' FOR TABLE tracetst;
*/
void sptrace(VARCHAR_LATIN    *trace_text,
             INTEGER          *result,
```

```

        char                sqlstate[6])
{
    SQL_TEXT    trace_string[257];
    void        *argv[20]; /* only need 1 arg -- its an array */
    int         length[20]; /* one length for each argument */
    int         tracelen;
    /* read trace string */
    FNC_Trace_String(trace_string);
    /* Get length of string */
    tracelen = strlen(trace_string);
    /* Make sure tracing is turned on */
    if (tracelen == 0)
        return;
    if (trace_string[0] == 'T')
    {
        argv[0] = trace_text;
        length[0] = strlen(trace_text) +1;
        /* Have something to trace */
        FNC_Trace_Write_DL(1, argv, length);
    }
}

```

The following SQL text defines the procedure that calls the trace function *sptracedemo*.

```

SHOW PROCEDURE sptracedemo;
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
-----
CREATE PROCEDURE sptracedemo (
    num_rows INTEGER )
BEGIN
-- Teradata mode SQL procedure
    DECLARE dummy INTEGER;
    DECLARE start_val INTEGER;
    DECLARE fnum FLOAT;
    SET dummy = sptrace('Start of sptrace demo');
    BEGIN
        DECLARE EXIT HANDLER FOR SQLSTATE '52010'
        BEGIN
            -- Table already exists, delete the contents
            DELETE sp_demo1 ALL;
            SET dummy=sptrace('Deleted contents of sp_demo1 in handler');
            -- Get error here and procedure will exit
        END;
    END;

```

```

CREATE TABLE sp_demo1 (
  a INTEGER,
  b FLOAT);
SET dummy = sptrace('Table sp_demo1 created');
END;
SET start_val = 1;
SET fnum = 25.3;
WHILE start_val <= num_rows DO
  INSERT INTO sp_demo1 (start_val, fnum);
  SET dummy = sptrace('did: insert (' || start_val || ', ' || fnum
    || ');');
  SET start_val = start_val +1;
  SET fnum = sqrt(fnum);
END WHILE;
SET dummy = sptrace('Got to end of sptrace demo'); END;

```

The remainder of the scenario constitutes the actual example.

Note that global temporary trace tables are user-defined except for first two columns, which are always fixed.

First show the definition of the global temporary trace table, *tracetst*, used for this example.

```

SHOW TABLE tracetst;
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
-----
CREATE MULTISET GLOBAL TEMPORARY TRACE TABLE tracetst,NO FALLBACK,
CHECKSUM = DEFAULT, NO LOG (
  proc_id  BYTE(2),
  sequence INTEGER,
  trace_str VARCHAR(100) CHARACTER SET LATIN NOT CASESPECIFIC)
ON COMMIT PRESERVE ROWS;

```

The next request enables function traceback for the session (see [SET SESSION FUNCTION TRACE](#) for additional information). The variable *t* is a user-defined string, interpreted by your trace UDF, that can be up to 255 characters long.

```

SET SESSION FUNCTION TRACE USING 't' FOR TABLE tracetst;
*** Set SESSION accepted.
*** Total elapsed time was 1 second.

```

Function traceback has been enabled for the session.

Now select the contents of the global temporary trace table *tracetst* to ensure that it has no rows.

```

SELECT *
FROM tracetest
ORDER BY 1, 2;
*** Query completed. No rows found.
*** Total elapsed time was 1 second.

```

The trace table is empty.

Run the trace procedure *sptracedemo*, whose definition is indicated by the SHOW PROCEDURE request at the end of this example:

```

CALL sptracedemo(3);
*** Procedure has been executed.
*** Total elapsed time was 1 second.

```

Select the contents of your global temporary trace table *tracetst* after the procedure executes. Refer to the procedure definition to see exactly what it traces.

```

SELECT *
FROM tracetest
ORDER BY 1, 2;
*** Query completed. 6 rows found. 3 columns returned.
*** Total elapsed time was 1 second.
proc_id Sequence trace_str
-----
FF3F      1      Start of sptrace demo
FF3F      2      Table sp_demo1 created
FF3F      3      did: insert (          1, 2.530000000000000E 001);
FF3F      4      did: insert (          2, 5.02991053598372E 000);
FF3F      5      did: insert (          3, 2.24274620409526E 000);
FF3F      6      Got to end of sptrace demo

```

Now call *sptracedemo* again.

```

CALL sptracedemo(3);
*** Procedure has been executed.
*** Total elapsed time was 1 second.

```

Select the contents of the global temporary trace table again. Note that it takes a different path because the global temporary trace table has already been populated with rows this time.

```

SELECT *
FROM tracetest

```

```

ORDER BY 1, 2;
*** Query completed. 12 rows found. 3 columns returned.
*** Total elapsed time was 1 second.
proc_id Sequence trace_str
-----
FF3F      1      Start of sptrace demo
FF3F      2      Table sp_demo1 created
FF3F      3      did: insert (          1, 2.530000000000000E 001);
FF3F      4      did: insert (          2, 5.02991053598372E 000);
FF3F      5      did: insert (          3, 2.24274620409526E 000);
FF3F      6      Got to end of sptrace demo
FF3F      7      Start of sptrace demo
FF3F      8      deleted contents of sp_demo1 in handler
FF3F      9      did: insert (          1, 2.530000000000000E 001);
FF3F     10      did: insert (          2, 5.02991053598372E 000);
FF3F     11      did: insert (          3, 2.24274620409526E 000);
FF3F     12      Got to end of sptrace demo

```

Disable function traceback to show that nothing additional is written to the trace table when the procedure is called, but function traceback is not enabled.

```

SET SESSION FUNCTION TRACE OFF;
*** Set SESSION accepted.
*** Total elapsed time was 1 second.
CALL sptracedemo(3);
*** Procedure has been executed.
*** Total elapsed time was 1 second.
SELECT *
FROM tracetest
ORDER BY 1, 2;
*** Query completed. 12 rows found. 3 columns returned.
*** Total elapsed time was 1 second.
proc_id Sequence trace_str
-----
FF3F      1      Start of sptrace demo
FF3F      2      Table sp_demo1 created
FF3F      3      did: insert (          1, 2.530000000000000E 001);
FF3F      4      did: insert (          2, 5.02991053598372E 000);
FF3F      5      did: insert (          3, 2.24274620409526E 000);
FF3F      6      Got to end of sptrace demo
FF3F      7      Start of sptrace demo
FF3F      8      deleted contents of sp_demo1 in handler
FF3F      9      did: insert (          1, 2.530000000000000E 001);
FF3F     10      did: insert (          2, 5.02991053598372E 000);

```


FF3F	11	did: insert (3, 2.24274620409526E 000);
FF3F	12	Got to end of sptrace demo

The identical 12 rows are selected from *tracetst*, so function traceback was successfully disabled.

Related Information

See [SET SESSION FUNCTION TRACE](#) and *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for information about enabling function traceback.

See CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for general information about global temporary tables and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about coding external functions.

CREATE FOREIGN TABLE

Foreign tables enable Vantage to access data in external object storage, such as semi-structured and unstructured data in Amazon S3, Azure Blob storage, and Google Cloud Storage. In-database integration of this data allows data scientists and analysts to read and process this data with Vantage, using standard SQL. You can join external data to relational data in Vantage, and process it using built-in Vantage analytics and functions.

The external data can be in various standard data formats, such as JSON, CSV, and Parquet.

Creating a foreign table requires specifying the location of the data. You must also specify the credentials, except if you are using AWS IAM.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the CREATE TABLE privilege on the database or user in which the table is created.

Privileges Granted Automatically

The creator receives the following privileges WITH GRANT OPTION on the newly created table:

- DROP TABLE
- SELECT

For more information about database privileges, see *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

CREATE FOREIGN TABLE Syntax

```
CREATE [MULTISET] FOREIGN TABLE table_specification
  [ , table_option [,...] ]
  [ , external_security_clause ]
  [ ( location_column, { payload_column | data_column_definition }) ]
  USING (
    LOCATION ( 'external_file_path' )
    [ PATHPATTERN ( 'value' ) ]
    [ MANIFEST ( { 'TRUE' | 'FALSE' } ) ]
    [ ROWFORMAT ( 'encoding_format' ) ]
    [ STOREDAS ( { 'TEXTFILE' | 'PARQUET' } ) ]
    [ HEADER ( { 'TRUE' | 'FALSE' } ) ]
    [ STRIP_EXTERIOR_SPACES ( { 'TRUE' | 'FALSE' } ) ]
    [ STRIP_ENCLOSING_CHAR ( 'NONE' ) ]
  )
  [ [,] NO PRIMARY INDEX ]
  [ [,] PARTITION BY COLUMN ] [;]
```

table_specification

```
[database_name. | user_name.] table_name
```

external_security_clause

```
EXTERNAL SECURITY [ { INVOKER | DEFINER } TRUSTED ]
  [ database_name. | user_name. ] authorization_name
```

The total number of specified columns cannot exceed 2048.

CREATE FOREIGN TABLE Syntax Elements

MULTISET

Optional keyword you can specify for readability. Foreign tables are always multiset tables.

*table_specification***database_name**

Name of the database to contain the foreign table, if other than the current database.

user_name

Name of the user to contain the foreign table, if other than the current user.

table_name

The name of the new foreign table.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

table_option

Specifies table options for the foreign table. Can be any `table_option` allowed in CREATE TABLE (see [table_option](#)), except as noted:

The following table options are not supported for foreign tables:

- BLOCKCOMPRESSIONLEVEL
- BLOCKCOMPRESSIONALGORITHM
- BLOCKCOMPRESSION
- MERGEBLOCKRATIO
- CHECKSUM
- DATABLOCKSIZE
- FREESPACE
- Journal table

The following table options are not supported for external data formatted as Parquet:

- Row partitioning
- Subrow partitioning
- Multicolumn partitions
- Autocompression

MAP

You can specify an existing contiguous or sparse map for the table.

Specifying a map is optional. If you do not specify a map, the default map is determined according to the following order of precedence:

- If the immediate owner is not the creator:

- Default map, if defined, for the profile of the immediate owner.
- Default map, if defined, for the immediate owner.
- System-default map.
- Default map, if defined, for the profile of the creator.
- Default map, if defined, for the creator.
- System-default map.

See GRANT MAP in *Teradata Vantage™ - SQL Data Control Language*, B035-1149. Also, see the CREATE PROFILE [DEFAULT MAP](#) option and the CREATE USER [DEFAULT MAP](#) option.

map_name

Name of an existing contiguous or sparse map.

You cannot specify TD_DataDictionaryMap or TD_GlobalMap.

COLOCATE USING *colocation_name*

Optionally, you can specify a colocation name so that the tables reside on the same AMPs to avoid a redistribution of the rows when the tables are joined on a primary index or primary AMP index. For example, you can colocate two tables, then join the tables on the primary index or primary AMP index columns.

You can only specify this option for a sparse map. For a contiguous map, the *colocation_name* is not needed for colocation and is set to NULL.

If you do not specify a colocation name, the name defaults to *database_table*, where *database* is the name of the database followed by an underscore (_) and *table* is the name of the table. If *database* exceeds 63 characters, *database* is truncated to 63 characters. If *table* exceeds 64 characters, *table* is truncated to 64 characters.

For a CREATE TABLE AS *source_table* statement, where the map for the created table defaults to the sparse map of the source table, *colocation_name* defaults to the colocation name of the source table.

external_security_clause

Specifies authorization for accessing remote storage. See [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#).

TRUSTED

Required keyword.

database_name

You can create this authorization in any database for which you have the CREATE AUTHORIZATION privilege.

user_name

You can create this authorization in the schema of any user for which you have the CREATE AUTHORIZATION privilege.

authorization_name

Name of authorization object.

You can use a Google Cloud authorization with a foreign table. See [Example: Creating a Google Cloud Authorization](#).

location_column

[Optional] Specify a column that has this definition:

```
Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC
```

The *location_column* and the *payload_column* or *data_column_definition* fields are optional, because NOS generates them. For CSV, if NOS cannot generate the columns, it generates the payload column of DATASET CSV instead.

The Payload column type of the generated location column depends on extension of files in location path:

File Extension	Payload Column Type
.json or .json.gz	JSON
.csv or .csv.gz	DATASET STORAGE FORMAT CSV
No file extension or unrecognized file extension	JSON, with unsupported file type warning
mixed .json and .csv files at the location path	Based on the type of the first valid file Vantage encounters. Subsequent files from the location that have different extensions are ignored. For this reason, it is best to specify a payload column for a location that has mixed format data, if you cannot arrange to have only one data format at the specified LOCATION.

If you specify *location_column*, you must specify it first.

For information about NOS, see *Teradata Vantage™ - Native Object Store Getting Started Guide*, B035-1214.

payload_column

[Optional for JSON and CSV tables.] Specify one column with one of the following definitions. The data type in the definition must match the type of data at the specified LOCATION.

The *location_column* and the *payload_column* or *data_column_definition* fields are optional, because NOS generates them. For CSV, if NOS is cannot generate the columns, it generates the payload column of DATASET CSV instead.

- For JSON data:
 - Payload JSON(8388096) INLINE LENGTH 32000 CHARACTER SET UNICODE
 - Payload JSON(16776192) INLINE LENGTH 64000 CHARACTER SET LATIN
- For CSV data:
 - Payload DATASET (2097088000) INLINE LENGTH 64000 STORAGE FORMAT CSV CHARACTER SET UNICODE
 - Payload DATASET (2097088000) INLINE LENGTH 64000 STORAGE FORMAT CSV CHARACTER SET LATIN

You cannot define the following attributes on a Payload column:

- NOT NULL
- COMPRESS
- CONSTRAINT
- REFERENCES
- CHECK

For more information about schema objects, see [CSV File Headers and Schemas](#).

NOS automatically discovers the schema for you.

Default CSV file format has these characteristics:

- The first record is the header.
- The field delimiter is comma (",").
- The record delimiter is line feed ("\n") or carriage return line feed ("\r\n").

Note:

ROWFORMAT must specify that the record delimiter is line feed.

Table Type	<i>payload_column</i>
JSON	Optional. If omitted, NOS generates a payload column whose type depends on extension of files in location path (see following table).
CSV	Optional. If omitted, NOS discovers the schema automatically.

File Extension	Payload Column Type
.json or .json.gz	JSON
.csv or .csv.gz	DATASET STORAGE FORMAT CSV
No file extension or unrecognized file extension	JSON, with unsupported file type warning
mixed .json and .csv files at the location path	Based on the type of the first valid file Vantage encounters. Subsequent files from the location that have different extensions are ignored. For this reason, it is best to specify a payload column for a location that has mixed format data, if you cannot arrange to have only one data format at the specified LOCATION.

For information about NOS, see *Teradata Vantage™ - Native Object Store Getting Started Guide*, B035-1214.

data_column_definition

[Optional for CSV- or Parquet-formatted data, disallowed otherwise.] Specify the definition of one or more data columns in the external file, in the order in which the records appear in the external file.

The *location_column* and the *payload_column* or *data_column_definition* fields are optional, because NOS generates them. For CSV, if NOS cannot generate the columns, it generates the payload column of DATASET CSV instead.

Each *data_column_definition* has the format *column_name data_type*.

CSV Files

Data columns are optional because NOS creates them from the information in the data itself. If you declare the columns and their data types, NOS checks their correctness.

If you specify data columns, these rules apply:

- The *column_name* need not match the name of the corresponding column in the header record.
- The *data_type* is the Vantage column data type to associate with the corresponding field in the CSV record.
- For a CSV table, *data_type* must be one of the following:
 - SMALLINT
 - INTEGER
 - BIGINT
 - DECIMAL (*precision*, *scale*) where *precision* is in the range [1, 38] and *scale* is in the range [0, 38].
 - BYTEINT
 - NUMBER

- REAL, FLOAT, or DOUBLE PRECISION
- BYTE
- VARBYTE
- CHAR CHARACTER SET { LATIN | UNICODE }
- VARCHAR CHARACTER SET { LATIN | UNICODE }
- CLOB of at most 16 MB
- BLOB of at most 16 MB
- DATE
- INTERVAL
- TIME
- TIMESTAMP

Note:

User-defined types (UDTs) and period data types (PDTs) are not supported.

- *data_type* can specify these attributes:
 - CASESPECIFIC
 - UPPERCASE
 - FORMAT
 - TITLE
- *data_type* cannot specify these attributes:
 - NOT NULL
 - COMPRESS
 - CONSTRAINT
 - REFERENCES
 - CHECK
- If *data_type* is a numeric type:
 - You can specify the FORMAT in which to display the data.
 - A field value enclosed in single quotation marks is treated as a string, causing an error and causing Vantage to skip the record.
- If *data_type* is a date type:
 - Vantage uses its FORMAT to determine the positions of the day, month, and year.
 - The default FORMAT is 'YY/MM/DD'.
 - You can specify a different FORMAT.
 - If the date field in a CSV record does not have the default FORMAT, you must specify its FORMAT.
 - Any FORMAT you specify must represent the month as MM and the day as DD.

- If *data_type* is a character type:
 - A field value can be enclosed in single or double quotation marks.
To strip the enclosing double quotation marks, use [STRIP_ENCLOSING_CHAR](#). (You cannot strip single quotation marks.)
 - A field value can have leading spaces, trailing spaces, or both.
 - By default, Vantage does not strip exterior space characters (leading or trailing spaces).
To strip exterior space characters, use either [STRIP_EXTERIOR_SPACES](#) or the SQL functions TRIM, LTRIM, and RTRIM.
- If you specify more columns than the number of fields that some records have, Vantage gives each extra column the value NULL.
- If you specify fewer columns than the number of fields that some records have, Vantage ignores the extra fields.

Parquet Files

The *data_type* must be the Vantage data type that corresponds to the Parquet data type in the Parquet schema:

Parquet Data Type	Corresponding Vantage Data Type	Description
UINT_8	SMALLINT	Unsigned integer with range 0 to 255.
UINT_16	INTEGER	Unsigned integer with range 0 to 65535.
UINT_32	BIGINT	Unsigned integer with range 0 to 4,294,967,295.
UINT_64	DECIMAL(20,0)	Unsigned integer with range 0 to 18,446,744,073,709,551,615.
INT_8	BYTEINT	Signed integer with range -128 to 127.
INT_16	SMALLINT	Signed integer with range -32768 to 32767.
INT_32	INTEGER	Signed integer with range -2,147,483,648 to 2,147,483,647.
INT_64	BIGINT	Signed integer with range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
INT_96	TIMESTAMP(6)	Used to represent timestamp. First 8 bytes has number of nanoseconds and next 4 bytes has number of days since Julian day.
DECIMAL(p,s)	DECIMAL(p,s)	Decimal where precision is between 1 and 38 (inclusive) and scale is between 0 and 38 (inclusive).
	VARBYTE(p+1), where p is the precision of the DECIMAL.	Precision and scale is greater than 38.

Parquet Data Type	Corresponding Vantage Data Type	Description
FLOAT	REAL	IEEE 32-bit floating point.
DOUBLE	REAL	IEEE 64-bit floating point.
DATE	DATE	Number of days from the Unix epoch, 1 January 1970
TIME_MILLIS	TIME(3) or TIME(6)	Stores time in milliseconds.
TIME_MICROS	TIME(6)	Stores time in microseconds.
TIMESTAMP_MILLIS	TIMESTAMP(3) or TIMESTAMP(6)	Stores timestamp in milliseconds.
TIMESTAMP_MICROS	TIMESTAMP(6)	Stores timestamp in microseconds.
INTERVAL	VARCHAR CHARACTER SET UNICODE	Stores interval in months, days and milliseconds.
BOOL	BYTEINT	Boolean (True or False).
STRING	VARCHAR or CLOB CHARACTER SET UNICODE	Encoded as a UTF8 byte array.
BSON	JSON STORAGE FORMAT BSON	BSON data.
JSON	JSON CHARACTER SET UNICODE	JSON data.
STRUCT	VARCHAR or CLOB CHARACTER SET UNICODE	Group of fixed members.
MAP	VARCHAR or CLOB CHARACTER SET UNICODE	Maps keys to values.
LIST	VARCHAR or CLOB CHARACTER SET UNICODE	Contains data that is stored in array.
ENUM	VARCHAR or CLOB CHARACTER SET UNICODE	Store enumerated values encoded as a UTF8 string.
ARRAY	VARCHAR or CLOB CHARACTER SET UNICODE	Stored as repeated fields. Can be an array of a single value or multiple values.

USING Clause

Specifies attributes of the foreign table. This clause is required.

LOCATION

You must specify a LOCATION value, which is a Uniform Resource Identifier (URI) pointing to the data in the external object storage system. The LOCATION value includes the following components:

- **Amazon S3:** `/connector/bucket.endpoint/[key_prefix]`
- **Azure Blob storage and Azure Data Lake Storage Gen2:** `/connector/storage-account.endpoint/container/[key_prefix]`
- **Google Cloud Storage (GCS):** `/connector/endpoint/bucket/[key_prefix]`

connector

Identifies the type of external storage system where the data is located.

Teradata requires the storage location to start with the following for all external storage locations:

- Amazon S3 storage location must begin with `/S3` or `/s3`
- Azure Blob storage location (including Azure Data Lake Storage Gen2 in Blob Interop Mode) must begin with `/AZ` or `/az`.
- Google Cloud Storage location must begin with `/GS` or `/gs`

storage-account

Used by Azure. The Azure storage account contains your Azure storage data objects.

endpoint

A URL that identifies the system-specific entry point for the external object storage system.

bucket (Amazon S3, GCS) or container (Azure Blob storage and Azure Data Lake Storage Gen2)

A container that logically groups stored objects in the external storage system.

key_prefix

Identifies one or more objects in the logical organization of the bucket data. Because it is a key prefix, not an actual directory path, the key prefix may match one or more objects in the external storage. For example, the key prefix `'/fabrics/cotton/colors/b'` would match objects: `/fabrics/cotton/colors/blue`, `/fabrics/cotton/colors/brown`, and `/fabrics/cotton/colors/black`. If there were organization levels below those, such as `/fabrics/cotton/colors/blue/shirts`, the same key prefix would gather those objects too.

Note:

Vantage validates only the first file it encounters from the location key prefix.

See [External File Object Rules](#).

For example, this LOCATION value might specify all objects on an Amazon cloud storage system for the month of December, 2001:

```
LOCATION(' /S3/YOUR-BUCKET.s3.amazonaws.com/csv/US-Crimes/csv-files/2001/Dec/')
```

connector	bucket	endpoint	key_prefix
S3	YOUR-BUCKET	s3.amazonaws.com	csv/US-Crimes/csv-files/2001/Dec/

This LOCATION could specify an individual storage object (or file), Day1.csv:

```
LOCATION(' /S3/YOUR-BUCKET.s3.amazonaws.com/csv/US-Crimes/csv-files/2001/Dec/Day1.csv')
```

connector	bucket	endpoint	key_prefix
S3	YOUR-BUCKET	s3.amazonaws.com	csv/US-Crimes/csv-files/2001/Dec/Day1.csv

This LOCATION specifies an entire container in an Azure external object store (Azure Blob storage or Azure Data Lake Storage Gen2). The *container* may contain multiple file objects:

```
LOCATION(' /AZ/YOUR-STORAGE-ACCOUNT.blob.core.windows.net/YOUR-CONTAINER/nos-csv-data')
```

```
LOCATION(' /az/nos1.blob.core.windows.net/demo/year2016')
```

connector	storage-account	endpoint	container	key_prefix
AZ	YOUR-STORAGE-ACCOUNT	blob.core.windows.net	YOUR-CONTAINER	nos-csv-data
az	nos1	blob.core.windows.net	demo	year2016

This is an example of a Google Cloud Storage location:

```
LOCATION(' /gs/storage.googleapis.com/YOUR-BUCKET/CSVDATA/RIVERS/rivers.csv')
```

connector	endpoint	bucket	key_prefix
GS	storage.googleapis.com	YOUR-BUCKET	CSVDATA/RIVERS/rivers.csv

PATHPATTERN

[Optional] NOS populates *value*.

value has this syntax:

```
/$variable_name[...]
```

Each slash (/) represents a key level.

The final *variable_name* (that is, *value*) cannot end with a slash.

Each *variable_name* can appear only once.

The variable names align with the entries in the key prefix location string. The first *variable_name* corresponds to first entry after the bucket name. For example, this *value*:

```
/$category/$filetype/$year/$month
```

Corresponds to this key prefix:

```
US-Crimes/csv-files/2001/Dec/
```

Default:

```
'$Var1/$Var2/$Var3/$Var4/$Var5/$Var6/$Var7/$Var8/$Var9/$Var10  
/$Var11/$Var12/$Var13/$Var14/$Var15/$Var16/$Var17/$Var18/$Var19/$Var20'
```

MANIFEST

Specifies whether the LOCATION value points to a manifest file or key prefix. The key prefix must include the full path and file name.

TRUE

LOCATION value points to a single manifest file.

FALSE

Default value. LOCATION value points to a key prefix.

ROWFORMAT

[Optional] NOS generates the ROWFORMAT clause.

For JSON data, *encoding_format* is:

```
{"record_delimiter":"\n", "character_set":"cs_value"}
```

For CSV data, *encoding_format* is:

```
{"field_delimiter":"fd_value",  
"record_delimiter":"\n", "character_set":"cs_value"}
```

For Parquet data, *encoding_format* is:

```
{"character_set":"cs_value"}
```

Note:

If the Payload column is defined for a CSV file, and its data type is DATASET CSV WITH SCHEMA, the schema specification takes precedence over ROWFORMAT. For more information about CSV schemas, see [CSV File Headers and Schemas](#).

record_delimiter

Specify that the record delimiter is the line feed character, "\n". The key name and "\n" and case-sensitive.

character_set

Specify the field character set. The key name is case-sensitive, but *cs_value* is not.

If you specify a Payload column, *cs_value* is "UTF8" if the Payload column character set is UNICODE and "LATIN" if the Payload column character set is LATIN.

If you do not specify a Payload column, NOS gets the Payload column character set from the file encoding and sets *cs_value* to "UTF8" if the Payload column character set is UNICODE and "LATIN" if the Payload column character set is LATIN.

field_delimiter

Specify the field delimiter. The key name and *fd_value* are case-sensitive.

Default: "," (comma)

STOREDAS

[Optional] Specifies the file format.

TEXTFILE

File format is text, for example, JSON or CSV. See [JSON External Files](#) and [CSV External Files](#). This is the default value.

PARQUET

File format is Parquet. If the system detects the format, it adds the code `STORED AS ('PARQUET')`. See [Parquet External Files](#).

HEADER

[Optional] Specify whether the first record in the CSV file is a header—that is, whether to skip the first record.

HEADER is disallowed with any of the following:

- Payload defined with data type DATASET CSV or JSON
- STOREDAS ('PARQUET')
- PARTITION BY COLUMN

A header contains the names of the record fields. If you specify HEADER ('FALSE'), Vantage generates field names of the form `col_n` (`col1`, `col2`, ...). For more information about headers, see [CSV External Files](#).

HEADER	CSV File Has Header	CSV File Has No Header
'TRUE' (default)	CREATE FOREIGN TABLE skips first record (header) and issues no warning. You can access record fields only by column names you specify (<i>location_column</i> , <i>data_column_definition</i> [...]).	CREATE FOREIGN TABLE skips first record.
'FALSE'	CREATE FOREIGN TABLE does not skip first record but treats header fields like data fields. Treating header fields like data fields causes either skipped records (because they cannot be converted to column data types) or invalid rows. Invalid rows can cause unexpected behavior and unexpected results. CREATE FOREIGN TABLE issues warning for skipped records.	CREATE FOREIGN TABLE does not skip first record and issues no warning.

STRIP_EXTERIOR_SPACES

[Optional] Specify whether to strip exterior (leading and trailing) spaces.

STRIP_EXTERIOR_SPACES is disallowed with any of the following:

- Payload defined with data type DATASET CSV or JSON
- STOREDAS ('PARQUET')
- PARTITION BY COLUMN

Default: 'FALSE'

STRIP_ENCLOSING_CHAR

[Optional] Specify either 'NONE' or the enclosing character to skip when there are no exterior spaces. The *enclosing_character* must be double quotation mark (").

STRIP_ENCLOSING_CHAR is disallowed with any of the following:

- Payload defined with data type DATASET CSV or JSON
- STOREDAS ('PARQUET')
- PARTITION BY COLUMN

Default: 'NONE'

NO PRIMARY INDEX

[Optional] The foreign table is created without a primary index. This is the default.

Foreign tables are always NOPI tables. A primary index, secondary index, join index, or hash index cannot be defined on a foreign table.

PARTITION BY COLUMN

Note:

This option is required only for foreign tables that will contain Parquet-formatted data.

The foreign table data is grouped in storage such that all data from each column is stored together in a separate partition.

Usage Notes

External File Types

Vantage can read data from external object storage, such as Amazon S3, Azure Blob storage (including Azure Data Lake Storage Gen2 in Blob Interop Mode), and GCS.

External File Object Rules

External files must be in one of the following formats:

- JSON
- CSV
- Parquet

Field names in CSV and JSON are case sensitive.

Unsupported files and compression types are skipped.

Records that contain errors are skipped. Errors can include mismatched quotation marks (" "), embedded line feeds, and so forth.

External files with UTF-8 BOMs (Byte Order Markers) are supported, but UTF-16LE, UTF-16BE, UTF-32LE, UTF-32BE BOMs are not supported.

Queries return results only for file formats where the format matches the format of the first file specified by the LOCATION option, including:

- File type (JSON, CSV, or Parquet)
- File compression (GZIP for JSON or CSV, SNAPPY for Parquet, or uncompressed)
- Character encoding (LATIN or UTF-8)
- Field delimiter (comma, tab, and so on)
- Record delimiter (line feed)

JSON External Files

Each record in the JSON external data must be formatted on a single line, terminated by a new line symbol (\n).

External JSON files can contain individual records or a single JSON array that uses a comma (,) as the record delimiter. For single-array JSON files, the record can span across multiple lines. Optionally, the JSON array can be named.

Vantage can read the following JSON document in external storage:

```
{"field1": true,"field2": "somestring","field3": {"field4": 1}}
```

Vantage cannot read the external JSON data if it is stored in this format, due to the line breaks:

```
{
  "field1" : true,
  "field2":
    "somestring",
  "field3":
    {
      "field4":1
    }
}
```

Because the following external JSON data is in a single array (delimited by the opening and closing square bracket characters), using commas to delimit individual records in the array, Vantage can read the data, even with the embedded line breaks:

```
[{
    "field1": "string1", "field2": "string2"
}, {
    "field1": "string3", "field2": "string4"
}]
```

The field names are case sensitive. So, a reference to Field1 will not match the "field1" in the examples above.

Spaces inside quotes for the field names are significant. So, if the record contained "field1 ", then a reference to payload.field1 does not match.

Here is an example of a named JSON array:

```
{ "Fruits": [
    \{ "fruit": "Apple", "size": "Large", "color": "Red" }
    ,
    { "fruit": "Banana", "size": "Medium", "color": "Yellow" },
    { "fruit": "Orange", "size": "Medium", "color": "Orange" },
    { "fruit": "Guava", "size": "Small", "color": "Green" },
    { "fruit": "Grapes", "size": "VerySmall", "color": "Black" }
  ]
}
```

External JSON files can be compressed in GZIP format. Other forms of file compression are not supported.

CSV External Files

Do not enclose a CSV value in quotation marks if it is to be used or cast as a numeric value. If you do, Vantage returns a null value or skipped record error. To strip quotation marks, use STRIP_ENCLOSING_CHAR.

You can compress an external CSV file only in GZIP format.

CSV File Headers and Schemas

If the file has no header record, specify HEADER ('FALSE').

Whether or not the table has a header record:

- You can specify a nondefault field delimiter or character set only with ROWFORMAT.
- If you do not specify ROWFORMAT, NOS generates the ROWFORMAT clause.

The generated clause specifies the values of `field_delimiter` and `character_set` (*fd_value* and *cs_value*), based on the file encoding.

Parquet External Files

To access external files containing Parquet-formatted data, you must specify PARQUET for the USING STOREDAS option in the CREATE FOREIGN TABLE statement.

Column names for Parquet data can contain a maximum of 128 characters and are case-sensitive.

Foreign tables for Parquet data are created without a primary index (NOPI) and are column-partitioned.

You cannot specify the following table-level options for foreign tables that contain Parquet formatted data:

- Row partitioning
- Subrow partitioning (partial row partitioning for a column-partitioned table)
- Multicolumn partitions
- Autocompression
- ROWFORMAT option of the USING clause
- A primary index

External Parquet files can be compressed in Snappy format.

You must explicitly define data columns for foreign tables that contain Parquet data. Define the columns in the same order as the columns in the Parquet files, and use Vantage data types that correspond to the Parquet data types:

Parquet Data Type	Corresponding Vantage Data Type	Description
UINT_8	SMALLINT	Unsigned integer with range 0 to 255.
UINT_16	INTEGER	Unsigned integer with range 0 to 65535.
UINT_32	BIGINT	Unsigned integer with range 0 to 4,294,967,295.
UINT_64	DECIMAL(20,0)	Unsigned integer with range 0 to 18,446,744,073,709,551,615.
INT_8	BYTEINT	Signed integer with range -128 to 127.
INT_16	SMALLINT	Signed integer with range -32768 to 32767.
INT_32	INTEGER	Signed integer with range -2,147,483,648 to 2,147,483,647.
INT_64	BIGINT	Signed integer with range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

Parquet Data Type	Corresponding Vantage Data Type	Description
INT_96	TIMESTAMP(6)	Used to represent timestamp. First 8 bytes has number of nanoseconds and next 4 bytes has number of days since Julian day.
DECIMAL(p,s)	DECIMAL(p,s)	Decimal where precision is between 1 and 38 (inclusive) and scale is between 0 and 38 (inclusive).
	VARBYTE(p+1), where p is the precision of the DECIMAL.	Precision and scale is greater than 38.
FLOAT	REAL	IEEE 32-bit floating point.
DOUBLE	REAL	IEEE 64-bit floating point.
DATE	DATE	Number of days from the Unix epoch, 1 January 1970
TIME_MILLIS	TIME(3) or TIME(6)	Stores time in milliseconds.
TIME_MICROS	TIME(6)	Stores time in microseconds.
TIMESTAMP_MILLIS	TIMESTAMP(3) or TIMESTAMP(6)	Stores timestamp in milliseconds.
TIMESTAMP_MICROS	TIMESTAMP(6)	Stores timestamp in microseconds.
INTERVAL	VARCHAR CHARACTER SET UNICODE	Stores interval in months, days and milliseconds.
BOOL	BYTEINT	Boolean (True or False).
STRING	VARCHAR or CLOB CHARACTER SET UNICODE	Encoded as a UTF8 byte array.
BSON	JSON STORAGE FORMAT BSON	BSON data.
JSON	JSON CHARACTER SET UNICODE	JSON data.
STRUCT	VARCHAR or CLOB CHARACTER SET UNICODE	Group of fixed members.
MAP	VARCHAR or CLOB CHARACTER SET UNICODE	Maps keys to values.
LIST	VARCHAR or CLOB CHARACTER SET UNICODE	Contains data that is stored in array.

Parquet Data Type	Corresponding Vantage Data Type	Description
ENUM	VARCHAR or CLOB CHARACTER SET UNICODE	Store enumerated values encoded as a UTF8 string.
ARRAY	VARCHAR or CLOB CHARACTER SET UNICODE	Stored as repeated fields. Can be an array of a single value or multiple values.

The Parquet files do not need the same column layout and do not need the same data type (if the data types are compatible) as the foreign table definition. For example:

File	Column 1	Column 2	Column 3	Column 4	Allowed?
Foreign table is defined with 3 columns with the data types specified to the right	INT	VARCHAR	DATE		N/A
File 1 - has 3 columns of the same data types as the foreign table	INT	VARCHAR	DATE		YES
File 2 - added columns at the end	INT	VARCHAR	DATE	TIMESTAMP	YES
File 3 - removed columns from the end	INT	VARCHAR			YES, if column 3 is nullable
File 4 - different data types	VARCHAR	INTERVAL	BIGINT		NO, the file is skipped
File 5 - different, but compatible data types	BIGINT	CLOB	DATE		YES

Foreign Table Locking

Vantage cannot issue standard database transaction locking against data in external object storage, therefore these queries behave as if the foreign table has only an access lock (as if you used the LOCKING TABLE ... FOR ACCESS request modifier).

If you perform a join between a foreign table and a Vantage native table, normal transaction locking rules still apply to the Vantage table.

Queries on External Data are Nondeterministic

Because data may be added or deleted to the external storage system at any time, identical queries against external storage may yield different results.

LOCATION Key Prefix Best Practices

- All files that match to a key prefix must be of the same data format type (csv, json, or Parquet).
- Related data of different formats must be put into different key prefix location, (for example:
 - **Amazon S3:** `/S3/YOUR-BUCKET.s3.amazonaws.com/csv-table1` and `/S3/YOUR-BUCKET.s3.amazonaws.com/json-table-1`
 - **Azure Blob storage and Azure Data Lake Storage Gen2:** `/AZ/YOUR-STORAGE-ACCOUNT.blob.core.windows.net/YOUR-CONTAINER/csv-table1` and `/AZ/YOUR-STORAGE-ACCOUNT.blob.core.windows.net/YOUR-CONTAINER/json-table-1`
 - **GCS:** `/gs/storage.googleapis.com/YOUR-BUCKET/csv-table1` and `/gs/storage.googleapis.com/YOUR-BUCKET/json-table-1`
- Files that are part of different logical tables must be located at different key prefix locations, for example:
 - **Amazon S3:** `/S3/YOUR-BUCKET.s3.amazonaws.com/emp-table` and `/S3/YOUR-BUCKET.s3.amazonaws.com/dept-table`
 - **Azure Blob storage and Azure Data Lake Storage Gen2:** `/AZ/YOUR-STORAGE-ACCOUNT.blob.core.windows.net/YOUR-CONTAINER/emp-table` and `/az/YOUR-STORAGE-ACCOUNT.blob.core.windows.net/YOUR-CONTAINER/dept-table`
 - **GCS:** `/gs/storage.googleapis.com/YOUR-BUCKET/emp-table` and `/gs/storage.googleapis.com/YOUR-BUCKET/dept-table`
- If the CSV files have different fields of data, but do not have individual file headers, you should group the files with the same fields into different key prefix locations.
- If different kinds of data are included at a single key prefix location, querying that data will be inefficient. For example if you mix department and employee data in a single key prefix location, a query looking for a particular employee would require Vantage to read all the files at that location, including files that contained only department data.

Best practice is to group files containing the same kind of data into a single external storage key prefix location, and use that key prefix location to create a single foreign table, or in a single `READ_NOS` query. This applies to all kinds of external data, JSON, CSV, and Parquet.

The same logic applies to mixing CSV files that have different headers. A key prefix location may include several of these CSV files, but querying the data would be inefficient due to the disparate field patterns.

For the same reason, do not mix CSV or Parquet files that have different schemas at the same key prefix locations. In these cases, querying the data is not only inefficient, but it also results in many warnings indicating skipped records and files.

Indexes and Foreign Tables

You cannot create primary indexes, secondary indexes, join indexes, or hash indexes on a foreign table.

Table Types and Foreign Tables

You cannot create a volatile, error, queue, temporary, or global temporary trace table as a foreign table.

Copying Foreign Tables

To create a copy of a foreign table with same options as an existing foreign table, use the statement `CREATE TABLE AS`. The form of `CREATE TABLE AS` determines whether you must use the `WITH NO DATA` clause:

- `CREATE TABLE new_foreign_table AS original_foreign_table`
You must use the `WITH NO DATA` clause.
- `CREATE TABLE new_foreign_table AS (SELECT select_list FROM original_foreign_table)`

You can use either the `WITH NO DATA` or `WITH DATA` clause.

`CREATE TABLE AS ... WITH NO DATA` copies the definition of *original_foreign_table* to *new_foreign_table*, and *new_foreign_table* accesses the same external data as *original_foreign_table*.

To copy the payload data from *original_foreign_table* into columns of *new_foreign_table*:

1. Create a view of the foreign table and use `CAST` to name the columns.
2. Use the `INSERT ... SELECT` statement to copy the payload data into the columns.

Related Information:

[CREATE TABLE and CREATE TABLE AS](#)

[Example: Copying Data from Foreign Table into Permanent Table](#)

[Example: Importing External Data from Foreign Table into Permanent Table](#)

Load Isolation and Foreign Tables

You cannot create a foreign table as a load isolated table. That is, you cannot specify the `WITH CONCURRENT ISOLATED LOADING` option when creating a foreign table.

Column Level Compression and Foreign Tables

You cannot specify algorithmic or multivalued compression on the columns of a foreign table.

Disable HASH BY RANDOM

You can use CREATE TABLE AS with a SELECT subquery to copy data from a foreign table into a permanent table without a primary index. See [Example: Copying Data from Foreign Table into Permanent Table](#).

Internally, the SELECT subquery performs an INSERT ... SELECT operation with the HASH BY RANDOM clause to provide even distribution of data. The HASH BY RANDOM clause is the default for an INSERT ... SELECT operation from a foreign table into a permanent table without a primary index. You can disable this behavior.

This statement disables the random distribution of rows for the session.

```
SET QUERY_BAND = 'DisableHashByRandom = ON' FOR SESSION;
```

This statement disables the random distribution of rows for the transaction.

```
SET QUERY_BAND = 'DisableHashByRandom = ON' FOR TRANSACTION;
```

Native Object Store Limitations

These Native Object Store limitations are relevant to the CREATE FOREIGN TABLE statement.

CSV and JSON Payload Size

- The maximum payload size is 16,776,192 bytes for READ_NOS for CSV and JSON data formats.
 - For UNICODE, that is equivalent to 8,388,096 characters.
 - For LATIN, that is equivalent to 16,776,192 characters.

The actual payload size is equivalent to the dataset individual record size.

Parquet Page Size

The maximum Parquet page size supported is for both READ_NOS and WRITE_NOS is 16,776,192 bytes (including LOB and LOB-based UDT columns).

If some of the data in the Parquet record is binary, the maximum number of characters is proportionately reduced.

Foreign Table Limitations

- The following are not supported on foreign tables:
 - Primary indexes
 - Secondary indexes
 - Join indexes

- Hash indexes
- Row or column partitioning (except for Parquet foreign tables, which require column partitioning)
- DML operations that change table contents (INSERT, UPDATE, DELETE)

Parquet Format Limitations

- Certain complex data types are not supported, including STRUCT, MAP, LIST, and ENUM.
- Because support for the STRUCT data type is not available, nested Parquet object stores cannot be processed by Native Object Store.

Bad Values in a Numeric Field

As with any CSV or JSON datasets, a NOS table can become unusable if a newly uploaded Amazon S3 file contains a single bad value in a numeric field. If this happens, you can identify the bad record/field by using TRYCAST. You can first upload new Amazon S3 files to a temporary location in the object storage and use TRYCAST to make sure the numeric data is not corrupted. Then, you can move the data to its permanent location in the object store. See *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

For example:

Create an authorization object, if not already done:

```
CREATE AUTHORIZATION MyAuthObj
USER 'YOUR-ACCESS-KEY-ID'
PASSWORD 'YOUR-SECRET-ACCESS-KEY';
```

Create an example table.

```
CREATE MULTISET FOREIGN TABLE bad_numeric_fields
, EXTERNAL SECURITY MyAuthObj
USING (
    LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/DATA-BAD/bad_numeric_data')
);
```

Select:

```
SELECT payload FROM bad_numeric_fields;
```

Result:

```
Payload
-----
{ "site_no": "09396100", "datetime": "2018-07-16 00:00", "Flow": "44.7",
  "GageHeight": "1.50", "Precipitation": "0.00", "GageHeight2": "1.50" }
{ "site_no": "09396100", "datetime": "2018-07-14 00:00", "Flow": "232",
  "GageHeight": "2.16", "Precipitation": "0.00", "GageHeight2": "2.16" }
{ "site_no": "09396100", "datetime": "2018-07-14 00:14", "Flow": "186",
```

```
"GageHeight":"2.05", "Precipitation":"","GageHeight2":"2.05"}
{"site_no":"09400812", "datetime":"2018-07-12 00:09", "Flow":"","GageHeight":"jjjj", "Precipitation":"bcde", "BatteryVoltage":"12.5"}
{"site_no":"09400815", "datetime":"2018-07-12 00:00", "Flow":"0.00", "GageHeight":"-0.01", "Precipitation":"0.00", "BatteryVoltage":""}
{"site_no":"09400815", "datetime":"2018-07-12 00:07", "Flow":"","GageHeight":"","Precipitation":"","BatteryVoltage":"12.5"}
```

Select from the table:

```
SELECT payload.site_no, payload.GageHeight(INT) FROM bad_numeric_fields;
```

Result: The query returns an error.

Use TRYCAST to figure out which data is bad:

```
SELECT payload.site_no (CHAR(20)), TRYCAST(payload.GageHeight AS INT)
FROM bad_numeric_fields;
```

Sample result:

Payload.site_no	TRYCAST(Payload.GageHeight)
-----	-----
09396100	1
09396100	2
09396100	2
09400812	?
09400815	0
09400815	0

The results show there is bad data in the record with site_no 09400812.

WRITE_NOS Limitations

- The maximum Parquet page size supported for both READ_NOS and WRITE_NOS is 16,776,192 bytes (including LOB and LOB-based UDT columns). WRITE_NOS can create an unlimited number of these files.
- It is the responsibility of the user to clean up object store files on interrupted write operations. Write operations can be interrupted on transaction aborts or system resets, among other reasons.
- Concurrent WRITE_NOS requests to the same location are likely to cause an error to be returned. If a request gets an error, any objects written by the request are not deleted from external storage and must be manually cleaned up using tools provided by the external object store vendor, such as s3cmd. To avoid this, Teradata recommends specifying a unique location for concurrent requests.
- An error is reported if you attempt to write an object with same path name in the same location.
- Manifest files are not cumulative. If you want to add entries to a manifest, you must create a new manifest that includes the original entries plus the ones you want to add.

- An error is reported if you attempt to write another manifest file in the same location. Use `OVERWRITE('TRUE')` with `MANIFESTONLY('TRUE')` keywords to replace a manifest in the same location.

Examples

Examples: Creating Foreign Tables

These examples create foreign tables with the default columns and options by specifying only the *table_name* and `LOCATION`. The system detects the file format and creates the appropriate columns.

JSON File

CREATE FOREIGN TABLE Statement

```
CREATE FOREIGN TABLE riverflow_json
USING ( LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/JSONDATA/') );
```

SHOW TABLE Statement

Run the following:

```
SHOW TABLE riverflow_json;
```

Result:

```
CREATE MULTISET FOREIGN TABLE NOS_USR.riverflow_json ,FALLBACK ,
  MAP = TD_MAP1
  (
    Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
    Payload JSON(16776192) INLINE LENGTH 64000 CHARACTER SET LATIN)
USING
  (
    LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/JSONDATA/')
    MANIFEST ('FALSE')
    PATHPATTERN ('$var1/$var2/$var3/$var4/$var5')
    ROWFORMAT ('{"record_delimiter":"\n","character_set":"LATIN"}')
    STOREDAS ('TEXTFILE')
  )
NO PRIMARY INDEX ;
```

CSV File

CREATE FOREIGN TABLE Statement

```
CREATE FOREIGN TABLE riverflow_csv
USING ( LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/CSVDATA/') );
```

SHOW TABLE Statement

Run the following:

```
SHOW TABLE riverflow_csv;
```

Result:

```
CREATE MULTISET FOREIGN TABLE NOS_USR.riverflow_csv ,FALLBACK ,
  MAP = TD_MAP1
  (
    Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
    GageHeight2 DECIMAL(3,2),
    Flow DECIMAL(3,2),
    site_no INTEGER,
    datetime TIMESTAMP(0) FORMAT 'Y4-MM-DDBHH:MI',
    Precipitation DECIMAL(3,2),
    GageHeight DECIMAL(3,2))
USING
  (
    LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/CSVDATA/')
    MANIFEST ('FALSE')
    PATHPATTERN ('$var1/$site_no/$var3/$var4/$var5')
    ROWFORMAT ('{"field_delimiter":",","record_delimiter":"\n","character_set":"LATIN"}')
    STOREDAS ('TEXTFILE')
    HEADER ('TRUE')
    STRIP_EXTERIOR_SPACES ('FALSE')
    STRIP_ENCLOSING_CHAR ('NONE')
  )
NO PRIMARY INDEX ;
```

Parquet File

CREATE FOREIGN TABLE Statement

```
CREATE FOREIGN TABLE riverflow_parquet
USING ( LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/PARQUETDATA/') );
```

SHOW TABLE Statement

Run the following:

```
SHOW TABLE riverflow_parquet;
```

Result:

```
CREATE MULTISET FOREIGN TABLE NOS_USR.riverflow_parquet ,FALLBACK ,
  MAP = TD_MAP1
  (
    Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
    GageHeight2 FLOAT,
    Flow FLOAT,
    site_no BIGINT,
    datetime VARCHAR(16) CHARACTER SET UNICODE NOT CASESPECIFIC,
    Precipitation FLOAT,
    GageHeight FLOAT)
USING
  (
    LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/PARQUETDATA/')
    MANIFEST ('FALSE')
    PATHPATTERN ('$var1/$var2/$var3/$var4/$var5')
    STOREDAS ('PARQUET')
  )
NO PRIMARY INDEX
PARTITION BY COLUMN ADD 65525;
```

Example: Creating Foreign Table with PATHPATTERN

This example creates a foreign table by specifying the *table_name* (sensordata), LOCATION, and PATHPATTERN. In this example, LOCATION specifies levels below the bucket (data and logs), PATHPATTERN specifies several more, and the external file has JSON-formatted data.

CREATE FOREIGN TABLE Statement

```
CREATE MULTISET FOREIGN TABLE sensordata (
  Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
  Payload JSON(8388096) INLINE LENGTH 32000 CHARACTER SET UNICODE
)
USING (
  LOCATION ('/S3/YOUR-BUCKET.s3.amazonaws.com/data/logs')
  PATHPATTERN ('$Var1/$Var2/$var3/$Var4/$date')
)
NO PRIMARY INDEX;
```

Example: Creating Foreign Table with Payload Column with Latin Characters

This example is like [Example: Creating Foreign Table with PATHPATTERN](#) except for the character set of the Payload column—LATIN instead of UNICODE.

CREATE FOREIGN TABLE Statement

```
CREATE MULTISET FOREIGN TABLE sensordata (
  Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
  Payload JSON(16776192) INLINE LENGTH 64000 CHARACTER SET LATIN
)
USING (
  LOCATION ('/S3/YOUR-BUCKET.s3.amazonaws.com/data/logs')
  PATHPATTERN ('$Var1/$Var2/$var3/$Var4/$date')
)
NO PRIMARY INDEX;
```

Example: Creating and Using an Authorization for a Foreign Table**Create Authorization**

```
CREATE AUTHORIZATION MyAuthObj
USER 'YOUR-ACCESS-KEY-ID'
PASSWORD 'YOUR-SECRET-ACCESS-KEY';
```

Create Foreign Table

```
CREATE FOREIGN TABLE sensordata2,
EXTERNAL SECURITY MyAuthObj
```

```

USING (
  LOCATION ('/s3/YOUR-BUCKET.s3.amazonaws.com/')
);

```

Setting Up an Object Store for River Flow Data

Many of the examples use a sample river flow data set. USGS Surface-Water Data Sets are provided courtesy of the U.S. Geological Survey.

To run the examples, you can use the river flow data from Teradata-supplied public buckets. Or you can set up a small object store for the data set.

The following instructions explain how to set up the river flow data on your own external object store.

Your external object store must be configured to allow Advanced SQL Engine access.

When you configure external storage, you set the credentials to your external object store. Those credentials are used in SQL commands. The supported credentials for USER and PASSWORD (used in the CREATE AUTHORIZATION command) and for ACCESS_ID and ACCESS_KEY (used by READ_NOS and WRITE_NOS) correspond to the values shown in the following table:

System/Scheme	USER	PASSWORD
AWS	Access Key ID	Access Key Secret
Azure / Shared Key	Storage Account Name	Storage Account Key
Azure Shared Access Signature (SAS)	Storage Account Name	Account SAS Token
Google Cloud (S3 interop mode)	Access Key ID	Access Key Secret
Google Cloud (native)	Client Email	Private Key
On-premises object stores	Access Key ID	Access Key Secret
Public access object stores	<empty string> Enclose the empty string in single straight quotes: USER ''	<empty string> Enclose the empty string in single straight quotes: PASSWORD ''

The following provides further details for setting up credentials on your external object store:

Platform	Notes
Amazon S3 IAM	<p>IAM is an alternative to using an access key and password to secure S3 buckets. To allow Advanced SQL Engine access to S3 buckets that use IAM, your S3 bucket policy must be configured with the following Actions for the role that allows access to the bucket:</p> <ul style="list-style-type: none"> • S3:GetObject • S3:ListBucket

Platform	Notes
	<ul style="list-style-type: none"> • S3:GetBucketLocation <p>For WRITE_NOS:</p> <ul style="list-style-type: none"> • S3:PutObject <p>Note: Other Actions are also allowed, such as S3:HeadBucket, S3:HeadObject, S3:ListBucket, and so on.</p>
Azure Blob storage and Azure Data Lake Storage Gen2	<p>A user with access key information has full control over the entire account. Alternatively, SAS can be defined on Containers, or on objects within containers, so it provides a more fine-grained authentication approach. NOS uses either type of authentication and does not need to know what type of secret is being supplied.</p> <p>Note: Only Account SAS tokens are supported. Service SAS tokens generate errors and are rejected.</p>
Google Cloud Storage	<p>To allow Advanced SQL Engine access, the following permissions are needed:</p> <ul style="list-style-type: none"> • storage.objects.get • storage.objects.list

See your cloud vendor documentation for instructions on creating an external object store account.

The following steps may require the assistance of your public cloud administrator.

1. Create an external object store on a Teradata-supported external object storage platform. Give your external object store a unique name. In the Teradata-supplied examples, the bucket/container is called td-usgs. Because the bucket/container name must be unique, choose a name other than td-usgs.
2. On Amazon, generate an access ID and matching secret key for your bucket or generate an Identity and Access Management (IAM) user credential. On Azure, generate Account SAS tokens (not Service SAS tokens) for your td-usgs container. On Google Cloud Storage, generate an access ID and matching secret key for your bucket.
3. Download the sample data from <https://downloads.teradata.com/> (look for NOS Download Data) to your client/laptop. The ZIP file contains sample river flow data in CSV, JSON, and Parquet data formats.
4. Copy the sample data to your bucket or container, being careful to preserve the data directory structure. For example, use a location similar to the following:
 - Amazon S3: /S3/YOUR-BUCKET.s3.amazonaws.com/JSONDATA
 - Azure Blob storage and Azure Data Lake Storage Gen2: /az/YOUR-STORAGE-ACCOUNT.blob.core.windows.net/td-usgs/CSVDATA/
 - Google Cloud Storage: /gs/storage.googleapis.com/YOUR-BUCKET/CSVDATA/

Note, you can use the Amazon S3 or Azure management consoles or a utility like AWS CLI to copy the data to your external object store. For Google Cloud Storage, you can use the gsutil tool to copy the data to your external object store.

5. In the example code replace `td-usgs`, *YOUR-BUCKET*, and *YOUR-STORAGE-ACCOUNT* with the location of your object store.
6. Replace *YOUR-ACCESS-KEY-ID* and *YOUR-SECRET-ACCESS-KEY* with the access values for your external object store.

Example: Copying Foreign Table without Data

This example uses `CREATE TABLE AS ... WITH NO DATA` to create a copy of a foreign table without its data. The copy has the same options as the original and accesses the same external data.

CREATE FOREIGN TABLE Statement for Original Foreign Table

The following statement creates the foreign table `riverflow_0627`. Vantage detects the JSON data type and creates the JSON payload column.

```
CREATE FOREIGN TABLE riverflow_0627,
  EXTERNAL SECURITY MyAuthObj
  USING (
    LOCATION ('/S3/YOUR-BUCKET.s3.amazonaws.com/
JSONDATA/09380000/2018/06/27.json')
  );
```

CREATE TABLE AS ... WITH NO DATA Statement for Copy

```
CREATE TABLE rivercopy_0627_t
AS riverflow_0627 WITH NO DATA;
```

Definition of Copy

Show the definition of the table:

```
SHOW TABLE myDB.rivercopy_0627_t;
```

Result:

```
CREATE MULTISET FOREIGN TABLE myDB.rivercopy_0627_t ,FALLBACK ,
  EXTERNAL SECURITY MyAuthObj ,
  MAP = TD_MAP1
  (
    Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
    Payload JSON(8388096) INLINE LENGTH 32000 CHARACTER SET UNICODE)
```

```

USING
(
    LOCATION ('/S3/YOUR-BUCKET.s3.amazonaws.com/JSONDATA/
09380000/2018/06/27.json')
    PATHPATTERN
('$Var1/$Var2/$Var3/$Var4/$Var5/$Var6/$Var7/$Var8/$Var9/$Var10/$Var11/$Var12/$Var13/$Var14/$Var15/$Var16/$Var17/$Var18/$Var19/$Var20')
    ROWFORMAT ('{"record_delimiter": "\n", "character_set": "UTF8"}')
    STOREDAS ('TEXTFILE')
)
NO PRIMARY INDEX ;

```

Example: Copying Data from Foreign Table into Permanent Table

This example uses `CREATE TABLE AS ... WITH DATA` to copy data from a foreign table into a permanent table without a primary index.

The foreign table is `riverflow_0627`, created in [Example: Copying Foreign Table without Data](#).

CREATE TABLE AS ... WITH DATA Statement for Permanent Table

```

CREATE TABLE riverdata_0627_perm
AS (SELECT location, payload FROM riverflow_0627)
WITH DATA
NO PRIMARY INDEX;

```

The `SELECT` subquery selects the `location` and `payload` columns from `riverflow_0627`, including the data. Internally, the subquery performs an `INSERT ... SELECT` operation with the `HASH BY RANDOM` clause to provide even distribution of data. `HASH BY RANDOM` is the default for an `INSERT ... SELECT` operation into a permanent table without a primary index from a foreign table. To disable this behavior, see [Disable HASH BY RANDOM](#).

Example: Creating Foreign Table to Access Parquet Data

This example shows how to create a foreign table for accessing Parquet data. The table definition includes an authorization object for accessing the remote repository containing the river flow Parquet files.

CREATE AUTHORIZATION Statement

```

CREATE AUTHORIZATION MyAuthObj
USER 'YOUR-ACCESS-KEY-ID'
PASSWORD 'YOUR-SECRET-ACCESS-KEY';

```

Parquet Data Schema

```
message schema {
  optional double GageHeight2;
  optional double Flow;
  optional int64 site_no;
  optional binary datetime (UTF8);
  optional double Precipitation;
  optional double GageHeight;
}
```

CREATE FOREIGN TABLE Statement

This statement creates the foreign table and specifies the authorization MyAuthObj. This table definition defines a column with the corresponding Teradata data type for each Parquet logical data type in the river flow data. The column definitions must match the data formats and positions in the Parquet file. Parquet foreign tables are partitioned by column.

```
CREATE FOREIGN TABLE riverflow_parquet
, EXTERNAL SECURITY MyAuthObj
(
  Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC
, GageHeight2 DOUBLE PRECISION
, Flow DOUBLE PRECISION
, site_no BIGINT
, datetime VARCHAR(16) CHARACTER SET UNICODE CASESPECIFIC
, Precipitation DOUBLE PRECISION
, GageHeight DOUBLE PRECISION
)
USING (
  LOCATION ('/S3/YOUR-BUCKET.s3.amazonaws.com/PARQUETDATA')
  STOREDAS ('PARQUET')
) NO PRIMARY INDEX
PARTITION BY COLUMN ;
```

Select Data from Foreign Table

This statement retrieves a row count, maximum, and minimum values from riverflow_parquet .

```
SELECT COUNT(Flow) AS count_col
, MAX(Flow) ( FORMAT '-ZZZZ9.99') AS max_col
, MIN(Flow) ( FORMAT '-ZZZZ9.99') AS min_col
FROM riverflow_parquet;
```

count_col	max_col	min_col
-----	-----	-----
15406	1570.00	0.00

Example: Importing External Data from Foreign Table into Permanent Table

This example imports external data from a foreign table.

Create Foreign Table

The following statement defines a foreign table using the authorization object called MyAuthObj. The foreign table is for CSV formatted river flow data stored in an Amazon S3 bucket.

```
CREATE FOREIGN TABLE riverflow_csv,
  EXTERNAL SECURITY MyAuthObj
  USING (
    LOCATION ('/s3/YOUR-BUCKET.s3.amazonaws.com/CSVDATA/09513780/')
  );
```

The following statement shows the new foreign table:

```
SHOW TABLE riverflow_csv;
```

Result:

```
CREATE MULTISET FOREIGN TABLE NU.riverflow_csv ,FALLBACK ,
  EXTERNAL SECURITY MyAuthObj ,
  MAP = TD_MAP1
  (
    Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
    GageHeight2 DECIMAL(3,2),
    Flow DECIMAL(3,2),
    site_no INTEGER,
    datetime TIMESTAMP(0) FORMAT 'Y4-MM-DDBHH:MI',
    Precipitation DECIMAL(3,2),
    GageHeight DECIMAL(3,2))

  USING
  (
    LOCATION ('/s3/YOUR-BUCKET.s3.amazonaws.com/CSVDATA/')
    MANIFEST ('FALSE')
```

```

    PATHPATTERN ('$var1/$site_no/$var3/$var4/$var5')
    ROWFORMAT
('{"field_delimiter":",","record_delimiter":"\n","character_set":"LATIN"}')
    STOREDAS ('TEXTFILE')
    HEADER ('TRUE')
    STRIP_EXTERIOR_SPACES ('FALSE')
    STRIP_ENCLOSING_CHAR ('NONE')
)
NO PRIMARY INDEX ;

```

Create View of Foreign Table

This statement defines a view that includes selected columns cast to appropriate data types.

```

CREATE VIEW riverflow_csv_v AS (
  SELECT
    CAST(datetime AS VARCHAR(20)) datetime,
    CAST(site_no AS CHAR(8)) Site_no,
    CAST(Flow AS FLOAT) Flow,
    CAST(GageHeight AS FLOAT) GageHeight,
    CAST(GageHeight2 AS FLOAT) GageHeight2,
    CAST(Precipitation AS FLOAT) Precipitation
  FROM riverflow_csv
);

```

The following statement shows the new view:

```

SELECT * FROM riverflow_csv_v;

```

datetime	Site_no	Flow	GageHeight	GageHeight2	Precipitation
2018-06-29 00:00	9400815	0.00000000000000E 000	-1.00000000000000E-002	?	0.00000000000000E 000
2018-07-02 14:35	9429070	?	?	?	1.16000000000000E 000
2018-07-01 00:00	9400815	0.00000000000000E 000	-1.00000000000000E-002	?	0.00000000000000E 000
2018-07-02 00:00	9400815	0.00000000000000E 000	-1.00000000000000E-002	?	0.00000000000000E 000
2018-07-12 00:00	9400815	0.00000000000000E 000	-1.00000000000000E-002	?	0.00000000000000E 000
2018-07-10 00:00	9400815	0.00000000000000E 000	-1.00000000000000E-002	?	0.00000000000000E 000
2018-07-09 00:00	9400815	0.00000000000000E 000	-1.00000000000000E-002	?	0.00000000000000E 000
2018-07-08 00:00	9400815	0.00000000000000E 000	-1.00000000000000E-002	?	0.00000000000000E 000
2018-06-29 00:15	9400815	0.00000000000000E 000	-1.00000000000000E-002	?	0.00000000000000E 000
2018-07-02 14:36	9429070	?	?	?	1.09000000000000E 000

Create Permanent Table

This statement defines a table with a primary index on the datetime column.

```

CREATE MULTISET TABLE riverflow_csv_t (
  datetime VARCHAR(20),
  site_no CHAR(8),
  Flow Float,

```

```
GageHeight Float,  
GageHeight2 Float,  
Precipitation Float  
) PRIMARY INDEX (datetime);
```

Insert Foreign Table Rows into Permanent Table

This statement inserts rows for the specified columns into the table riverflow_csv_t from the view riverflow_csv_v.

```
INSERT INTO riverflow_csv_t (  
datetime, Site_no, Flow, GageHeight, Precipitation, GageHeight2  
)  
SELECT datetime, Site_no, Flow, GageHeight, Precipitation, GageHeight2  
FROM riverflow_csv_v;
```

Display Permanent Table

```
SELECT * FROM riverflow_csv_t;
```

Result:

datetime GageHeight	site_no GageHeight2	Flow Precipitation
-----	-----	-----
2018-07-05 08:15	9513780	0.000000000000000E 000
-9.800000000000000E-001	0.000000000000000E 000	-1.440000000000000E 000
2018-06-28 10:15	9513780	0.000000000000000E 000
-9.800000000000000E-001	0.000000000000000E 000	-1.440000000000000E 000
2018-07-03 16:30	9513780	0.000000000000000E 000
-9.800000000000000E-001	0.000000000000000E 000	-1.440000000000000E 000
2018-07-11 21:45	9513780	0.000000000000000E 000
-9.800000000000000E-001	1.000000000000000E-002	-1.440000000000000E 000
[...]		

Example: SHOW TABLE for Foreign Table

This example creates a foreign table and uses SHOW TABLE to display its output in default and XML formats.

CREATE FOREIGN TABLE Statement

```
CREATE FOREIGN TABLE riverflow_csv,
  EXTERNAL SECURITY MyAuthObj (
    Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
    GageHeight2 DOUBLE PRECISION FORMAT '-ZZZ9.99',
    Flow DOUBLE PRECISION FORMAT '-ZZZZ9.99',
    site_no BIGINT,
    datetime VARCHAR(16) CHARACTER SET UNICODE CASESPECIFIC,
    Precipitation DOUBLE PRECISION FORMAT '-ZZZ9.99',
    GageHeight DOUBLE PRECISION FORMAT '-ZZZ9.99'
  )
  USING (
    LOCATION ('/s3/YOUR-BUCKET.s3.amazonaws.com/CSVDATA/')
  );
```

Default Format

```
SHOW TABLE riverflow_csv;
```

```
CREATE MULTISET FOREIGN TABLE riverflow_csv ,FALLBACK ,
  EXTERNAL SECURITY MyAuthObj,
  MAP = TD_MAP1
  (
    Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
    GageHeight2 FLOAT FORMAT '-ZZZ9.99',
    Flow FLOAT FORMAT '-ZZZZ9.99',
    site_no BIGINT,
    datetime VARCHAR(16) CHARACTER SET UNICODE CASESPECIFIC,
    Precipitation FLOAT FORMAT '-ZZZ9.99',
    GageHeight FLOAT FORMAT '-ZZZ9.99')
  USING
  (
    LOCATION ('/s3/YOUR-BUCKET.s3.amazonaws.com/CSVDATA/')
    MANIFEST ('FALSE')
    PATHPATTERN ('$var1/$site_no/$var3/$var4/$var5')
    ROWFORMAT
    ('{"field_delimiter":",","record_delimiter":"\n","character_set":"LATIN"}')
    STOREDAS ('TEXTFILE')
    HEADER ('TRUE')
    STRIP_EXTERIOR_SPACES ('FALSE')
```

```

        STRIP_ENCLOSING_CHAR ('NONE')
    )
    NO PRIMARY INDEX ;

```

XML Format

```
SHOW IN XML TABLE riverflow_csv;
```

```

*** XML of DDL statement returned.
*** Total elapsed time was 1 second.

```

```

-----
-----
<?xml version="1.0" encoding="UTF-16" standalone="no" ?><TeradataDBObjectSet
version="1.0" xmlns="http://schemas.teradata.com/dbobject" xmlns:xsi="htt
p://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
schemas.teradata.com/dbobject http://schemas.teradata.com/dbobject/
DBObject.xsd"><
Table authName="MyAuthObj" authType="GENERAL" baseClass="Table" dbName="NU"
fallback="true" foreigntable="true" kind="Multiset" map="TD_MAP1" map_kind=
"contiguous" name="riverflow_csv" objId="2:52890"
objVer="1"><ColumnList><Column name="Location" nullable="true"
order="1"><DataType><Char casespecif
ic="true" charset="UNICODE" length="2048" uppercase="false" varying="true"/></
DataType></Column><Column name="GageHeight2" nullable="true" order="2">
<DataType><Decimal precision="3" scale="2"/></DataType></Column><Column
name="Flow" nullable="true" order="3"><DataType><Decimal precision="3" scale=
"2"/></DataType></Column><Column name="site_no" nullable="true"
order="4"><DataType><Integer/></DataType></Column><Column format="Y4-MM-
DDBHH:MI" nam
e="datetime" nullable="true" order="5"><DataType><TimeStamp
fractionalSecondsPrecision="0"/></DataType></Column><Column
name="Precipitation" nullable
="true" order="6"><DataType><Decimal precision="3" scale="2"/></DataType></
Column><Column name="GageHeight" nullable="true" order="7"><DataType><Deci
mal precision="3" scale="2"/></DataType></Column></
ColumnList><UsingClauseList><Clause name="LOCATION" value="/s3/YOUR-
BUCKET.s3.amazonaws.com/CSV
DATA"/><Clause name="MANIFEST" value="FALSE"/><Clause name="PATHPATTERN"
value="$var1/$site_no/$var3/$var4/$var5"/><Clause name="ROWFORMAT" value="{&
quot;field_delimiter&quot;;&quot;;&quot;;&quot;;record_delimiter&quot;;&quot;;\n&
quot;;&quot;;character_set&quot;;&quot;;LATIN&quot;;}"><Clause name="STO
REDAS" value="TEXTFILE"/><Clause name="HEADER" value="TRUE"/><Clause

```



```

name="STRIP_EXTERIOR_SPACES" value="FALSE"/><Clause name="STRIP_ENCLOSING_CHAR"
value="NONE"/></UsingClauseList><Indexes><NoPrimaryIndex/></Indexes><SQLText><![
CDATA[CREATE MULTISET FOREIGN TABLE riverflow_csv ,FALLBACK ,
    EXTERNAL SECURITY MyAuthObj ,
    MAP = TD_MAP1
    (
        Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
        GageHeight2 DECIMAL(3,2),
        Flow DECIMAL(3,2),
        site_no INTEGER,
        datetime TIMESTAMP(0) FORMAT 'Y4-MM-DDBHH:MI',
        Precipitation DECIMAL(3,2),
        GageHeight DECIMAL(3,2))
USING
(
    LOCATION ('/s3/YOUR-BUCKET.s3.amazonaws.com/CSVDATA')
    MANIFEST ('FALSE')
    PATHPATTERN ('$var1/$site_no/$var3/$var4/$var5')
    ROWFORMAT
('{"field_delimiter":",","record_delimiter":"\n","character_set":"LATIN"}')
    STOREDAS ('TEXTFILE')
    HEADER ('TRUE')
    STRIP_EXTERIOR_SPACES ('FALSE')
    STRIP_ENCLOSING_CHAR ('NONE')
)
NO PRIMARY INDEX ]]></SQLText></Table><Environment><Server
dbRelease="17.10.01.01" dbVersion="17.10c.00.198dr196079" hostName="TDLabs"/
><User userI
d="00000604" userName="JOE"/><Session charset="ASCII"
dateTime="2021-06-09T15:03:48"/></Environment></TeradataDBObjectSet>

```

Example: NOS Detects Columns

In this example, the CREATE FOREIGN TABLE statement specifies no *location_column*, *payload_column*, or *data_definition_column*.

CREATE FOREIGN TABLE Statement

```

CREATE FOREIGN TABLE riverflow_csv_2,
    EXTERNAL SECURITY MyAuthObj
    USING (
        LOCATION ('/S3/s3.amazonaws.com/YOUR-BUCKET/CSVDATA/')
    );

```

SHOW TABLE Statement

```
SHOW TABLE riverflow_csv_2;
```

Result:

```
CREATE MULTISET FOREIGN TABLE NU.riverflow_csv_2 ,FALLBACK ,
  EXTERNAL SECURITY MyAuthObj ,
  MAP = TD_MAP1
  (
    Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
    GageHeight2 DECIMAL(3,2),
    Flow DECIMAL(3,2),
    site_no INTEGER,
    datetime TIMESTAMP(0) FORMAT 'Y4-MM-DDBHH:MI',
    Precipitation DECIMAL(3,2),
    GageHeight DECIMAL(3,2))
USING
  (
    LOCATION ('/S3/YOUR-BUCKET.s3.amazonaws.com/CSVDATA/')
    MANIFEST ('FALSE')
    PATHPATTERN ('$CSVDATA/$site_no/$2018/$byteint4/$filename')
    ROWFORMAT
    ('{"field_delimiter":",","record_delimiter":"\n","character_set":"LATIN"}')
    STOREDAS ('TEXTFILE')
    HEADER ('TRUE')
    STRIP_EXTERIOR_SPACES ('FALSE')
    STRIP_ENCLOSING_CHAR ('NONE')
  )
NO PRIMARY INDEX ;
```

SELECT Statement

```
SELECT GageHeight2, flow, site_no, datetime, Precipitation, GageHeight
FROM riverflow_csv_2;
```

Result:

GageHeight2	Flow	site_no	datetime	Precipitation	GageHeight
?	.00	9400815	2018-06-29 00:00	.00	-.01
?	?	9429070	2018-07-02 14:35	1.16	?
?	.00	9400815	2018-07-01 00:00	.00	-.01

?	.00	9400815	2018-07-02 00:00	.00	-.01
?	.00	9400815	2018-07-12 00:00	.00	-.01
?	.00	9400815	2018-07-10 00:00	.00	-.01
?	.00	9400815	2018-07-09 00:00	.00	-.01
?	.00	9400815	2018-07-08 00:00	.00	-.01
?	.00	9400815	2018-06-29 00:15	.00	-.01

Example: CREATE FOREIGN TABLE Statement Specifies Columns

In this example, the CREATE FOREIGN TABLE statement specifies *location_column* and more than one *data_definition_column*.

CREATE FOREIGN TABLE Statement

```
CREATE FOREIGN TABLE riverflow_csv,
  EXTERNAL SECURITY MyAuthObj (
    Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
    GageHeight2 FLOAT,
    flow FLOAT,
    site_no CHAR(8),
    datetime VARCHAR(20),
    Precipitation FLOAT,
    GageHeight FLOAT
  )
  USING (
    LOCATION ('/S3/YOUR-BUCKET.s3.amazonaws.com/CSVDATA/')
  );
```

SELECT Statement

```
SELECT GageHeight2, flow, site_no, datetime, Precipitation, GageHeight
FROM riverflow_csv;
```

Result:

datetime	GageHeight2	Precipitation	flow	site_no	GageHeight
2.16000000000000E 000	2.32000000000000E 002	09396100	2018-07-14 00:00		
0.00000000000000E 000	2.16000000000000E 000				
7.91000000000000E 001	1.56000000000000E 002	09429070	2018-07-02 00:00		
8.60000000000000E-001	5.52000000000000E 000				
1.50000000000000E 000	4.47000000000000E 001	09396100	2018-07-16 00:00		
0.00000000000000E 000	1.50000000000000E 000				
?	0.00000000000000E 000	09400815	2018-07-08 00:00		
0.00000000000000E 000	-1.00000000000000E-002				
2.05000000000000E 000	1.86000000000000E 002	09396100	2018-07-14		

```
00:14          ? 2.05000000000000E 000
7.90000000000000E 001 1.54000000000000E 002 09429070 2018-07-02 00:15
8.30000000000000E-001 5.60000000000000E 000
```

Example: NOS and Nondefault Field Delimiter, Header

This example shows how to create a foreign table with a nondefault field delimiter with NOS on (the default) and off. The CSV file has a header and its field delimiter is tab (`\t`).

rivers.tsv

```
09379180 LAGUNA CREEK AT DENNEHOTSO, AZ
09379910 COLORADO RIVER BELOW GLEN CANYON DAM, AZ
09380000 COLORADO RIVER AT LEES FERRY, AZ
09382000 PARIA RIVER AT LEES FERRY, AZ
09383300 FILLER DITCH AT GREER, AZ
09383400 LITTLE COLORADO RIVER AT GREER, AZ
09383100 COLORADO R ABV LITTLE COLORADO R NR DESERT VIEW
09379200 CHINLE CREEK NEAR MEXICAN WATER, AZ
09379050 LUKACHUKAI CREEK NEAR LUKACHUKAI, AZ
09379025 CHINLE CREEK AT CHINLE, AZ
```

Using NOS (Default)

NOS detects the field delimiter. You do not have to specify it.

```
CREATE FOREIGN TABLE rivers_t,
  EXTERNAL SECURITY MyAuthObj
  USING (
    LOCATION ('/s3/YOUR-BUCKET.s3.amazonaws.com/RIVERS/rivers.tsv')
  );
```

Not Using NOS

You must specify the field delimiter with ROWFORMAT.

```
CREATE FOREIGN TABLE rivers_t1,
  EXTERNAL SECURITY MyAuthObj
  USING
  (
    LOCATION ('/s3/YOUR-BUCKET.s3.amazonaws.com/RIVERS/rivers.tsv')
    ROWFORMAT ('{"field_delimiter": "\t"}')
  );
```

Example: STRIP_EXTERIOR_SPACES and STRIP_ENCLOSING_CHAR

This example shows a CSV table, `csv_tab_2`, and the effect of `STRIP_EXTERIOR_SPACES` and `STRIP_ENCLOSING_CHAR` on it. The asterisk (*) represents a single space character.

Select from `csv_tab_2`

```
SELECT a,b,c,d FROM csv_tab_2;
```

```
a,b,c,d
" $1.11", " $2,000 ",*$511*,*888
" $2.22"*,*" $6,000",*" $911"*,*"444"
```

STRIP_EXTERIOR_SPACES ('FALSE'), STRIP_ENCLOSING_CHAR ('NONE') (Default)

```
a " $1.11"
b " $2,000 "
c *$511*
d *888
a " $2.22"*
b *" $6,000"
c *"$911"*
d *"444"
```

STRIP_EXTERIOR_SPACES ('TRUE'), STRIP_ENCLOSING_CHAR ('NONE')

```
a " $1.11"
b " $2,000 "
c $511
d 888
a " $2.22"
b " $6,000"
c "$911"
d "444"
```

STRIP_EXTERIOR_SPACES ('FALSE'), STRIP_ENCLOSING_CHAR ('")

```
a *$1.11
b *$2,000*
c *$511*
d *888
a " $2.22"*
```

```

b  "*" $6,000"
c  *"$911"*
d  *"444"

```

STRIP_EXTERIOR_SPACES ('TRUE'), STRIP_ENCLOSING_CHAR ('')

```

a  *$1.11
b  *$2,000*
c  *$511*
d  *888
a  *$2.22
b  *$6,000
c  $911
d  444

```

CREATE ERROR TABLE

Defines the name and containing database of a new error table and specifies the name of the associated data table.

The associated data table can be a foreign table. By default, the maximum number of errors logged for a foreign table is 10. To specify a different maximum, use the DBS Control setting `ForeignTableErrorsLimit`, described in *Teradata Vantage™ - Database Utilities*, B035-1102.

If the associated data table is column-partitioned, Vantage creates the error table as it would for a non-column-partitioned table except that the error table is a NoPI table without partitioning.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Other SQL dialects support similar non-ANSI standard statements with names such as `START VIOLATIONS TABLE`.

Required Privileges

You must have the `CREATE TABLE` privilege on the database or user in which the error table is created.

To access an error table that contains UDT columns, you must have at least one of the following privileges:

- `UDTUSAGE` on the specified UDT
- `UDTUSAGE` on the `SYSUDTLIB` database
- `UDTTYPE` on the `SYSUDTLIB` database
- `UDTMETHOD` on the `SYSUDTLIB` database

Privileges Granted Automatically

The creator receives all the following privileges on the newly created error table:

- DELETE
- DROP TABLE
- INSERT
- SELECT
- UPDATE

CREATE ERROR TABLE Syntax

```
CREATE ERROR TABLE [ error_table_name_specification ]
  FOR data_table_name_specification [NO RLS] [;]
```

error_table_name_specification

```
[ database_name. | user_name. ] error_table_name
```

data_table_name_specification

```
[ database_name. | user_name. ] data_table_name
```

CREATE ERROR TABLE Syntax Elements

error_table_name_specification

database_name

user_name

Name of the database or user in which the error table is to be contained if not in the current database or user.

error_table_name

Name of the new error table.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

If you do not explicitly specify an error table name, the system generates one in the form *ET_data_table_name*, where the characters *ET_* are a literal string.

If *data_table_name* is longer than 27 characters, the system truncates it at the end and does not return a warning message.

The system returns an error message if the *ET_data_table_name* default name duplicates an existing table name. You can then specify an explicit unique error table name in place of the duplicate error table name.

data_table_name_specification

database_name

user_name

Name of the database or user in which the error table is to be contained if not in the current database or user.

data_table_name

Name of the data table for which this error table is being created.

data_table_name cannot specify a nonpartitioned NoPI table.

The name must correspond to either a permanent base data table or a queue table. Otherwise, the system returns an error message to the requestor.

NO RLS

NO RLS

Suppresses the generation of row-level security (RLS) constraint columns from the base table to the error table.

NO RLS is optional. If you do not specify this option, Vantage adds row-level security constraint columns to the error table by default if they are defined for the base table.

CREATE ERROR TABLE Examples

Example: Creating a Named Error Table

The following example creates an error table named *e* for data table *t*.

```
CREATE ERROR TABLE e FOR t;
```

Example: Creating an Error Table for a Row-Level Security Table

The following example creates an error table named *err_table_1* for the row-level security constraint-protected data table named *table_1_RLS_constraints*, but does not generate any row-level security columns for the error table because the request specifies the NO RLS option.


```
CREATE ERROR TABLE err_table_1 FOR table_1_RLS_constraints
NO RLS;
```

database_name

user_name

Name of the database or user in which the data table is contained if not in the current database or user.

Example: Creating an Error Table with a Default Name

The following example creates an error table with a default name of `et_t` for data table `t`.

```
CREATE ERROR TABLE FOR t;
```

Related Information

- CREATE ERROR TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- [DELETE DATABASE](#)
- [DROP MACRO](#)
- [SHOW object](#)
- INSERT and INSERT ... SELECT in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146
- MERGE in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146
- *Teradata® FastLoad Reference*, B035-2411
- *Teradata® MultiLoad Reference*, B035-2409

ALTER TABLE

Add one or more columns to a table or global temporary table, add or change attributes and options, including partitioning, constraints, and compression. You can also drop columns, change a join index, or revalidate a table.

Adds any of the following items:

- New attributes for one or more columns of a table or global temporary table definition.
- A set of new columns to an existing table or global temporary table definition.
- A set of columns to an existing column partition of a column-partitioned table.
- A set of columns as a new column partition for an existing column-partitioned table.

Drops one or more columns from a table or global temporary table definition.

Drops a column partition when all of its component columns are dropped.

Adds or drops the FALLBACK option for a table.

Adds or modifies the JOURNAL option for a table.

Modifies the following:

- Column-level and table-level constraints.
- Referential constraints.
- DATABLOCKSIZE or percent FREESPACE for a table or global temporary table definition.
- MERGEBLOCKRATIO for a permanent table or permanent journal table definition.
- LOG and ON COMMIT options for a global temporary table.
- Name of a column.
- Properties of the primary index for a table.
- Partitioning properties of a table, including modifications to the partitioning expression of a row-partitioned table.
- Column partitioning properties for column-partitioned tables and join indexes.
- Disk I/O integrity option for a table.
- Compression attributes for the columns of a table.
- Load isolation attribute for a table.

Regenerates table headers and optionally validates and corrects the partitioning of row-partitioned table rows.

For information about temporal tables and temporal syntax, see *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

Other SQL dialects support similar non-ANSI standard statements with names such as the following:

Required Privileges

To alter a table, you must have DROP TABLE privilege on that table or on the database containing the table.

To add or drop a row-level security column using ALTER TABLE, you must also have the CONSTRAINT ASSIGNMENT privilege in addition to DROP TABLE privilege.

When you execute either of the following ALTER TABLE forms, you must also have the INDEX privilege on the table:

- ALTER TABLE ... ADD UNIQUE (*column_list*)
- ALTER TABLE ... ADD PRIMARY KEY (*column_list*)

When you execute any of the following ALTER TABLE forms, you must also have the INSERT privilege on *save_table*:

- ALTER TABLE ... MODIFY WITH INSERT
- ALTER TABLE ... REVALIDATE PRIMARY INDEX WITH INSERT

You must have either of the following privileges to add a UDT column to a table or to drop a UDT column from a table:

- UDTUSAGE privilege on the specified UDT.
- At least one of the following privileges on the SYSUDTLIB database:
 - UDTUSAGE
 - UDTTYPE
 - UDTMETHOD

You must have the CONSTRAINT ASSIGNMENT privilege to add a row-level security constraint column to a table or drop a row-level security constraint column from a table.

Privileges Granted Automatically

ALTER TABLE ADD CONSTRAINT, where the added constraint is a referential integrity constraint, grants the following privileges automatically to the requestor adding the RI constraint:

- DELETE on the error table for the constraint
- DROP TABLE on the error table for the constraint
- INSERT on the error table for the constraint
- SELECT on the error table for the constraint
- UPDATE on the error table for the constraint

No privileges are granted automatically for any other ALTER TABLE option.

ALTER TABLE Syntax (Basic)

```
ALTER TABLE [ database_name. | user_name. ] table_name
{ alter_option [,...] |
  table_option [,...] [ alter_option [,...] ] |
  normalize |
  DROP NORMALIZE |
  modify_primary |
  MODIFY NO PRIMARY [ INDEX ] [ alter_partitioning ] |
  MODIFY alter_partitioning |
  FROM TIME ZONE = [ sign ] 'quotestring'
    [, TIMEDATEWZCONTROL = n ] [, WITH TIME ZONE ] |
  { SET | RESET } DOWN
} [;]
```

alter_option

```
{ ADD add_option |
  MODIFY [ [ CONSTRAINT ] name ] CHECK ( boolean_condition ) |
  RENAME { column_name { AS | TO } column_name |
```

```

        constraint_name { AS | TO } constraint_name
    } |
DROP drop_option
}

```

table_option

```

{ [NO] FALLBACK [ PROTECTION ] |
  WITH JOURNAL TABLE = [ database_name. ] table_name |
  [ NO | DUAL ] [ BEFORE ] JOURNAL |
  ON COMMIT { DELETE | PRESERVE } ROWS |
  [NO] LOG |
  [ NO | DUAL | [NOT] LOCAL ] AFTER JOURNAL |
  CHECKSUM = { DEFAULT | ON | OFF } [ IMMEDIATE ] |
  DEFAULT FREESPACE |
  FREESPACE = integer [ PERCENT ] |
  { DEFAULT | NO } MERGEBLOCKRATIO |
  MERGEBLOCKRATIO = integer [ PERCENT ] |

  { DATABLOCKSIZE = data_block_size [ BYTES | KBYTES | KILOBYTES ] |
    { MINIMUM | MAXIMUM | DEFAULT } DATABLOCKSIZE
  } IMMEDIATE |

  blockcompression |
  WITH [NO] [ CONCURRENT ] ISOLATED LOADING [ FOR { ALL | INSERT
| NONE } ]

  [ USING FAST MODE { ON | OFF } ]
}

```

normalize

```

ADD NORMALIZE [ ALL BUT ( column_name [,...] ) ]
  ON column_name [ ON { MEETS OR OVERLAPS | OVERLAPS OR MEETS } ]

```

modify_primary

```

MODIFY [ [NOT] UNIQUE ] PRIMARY [ AMP ] [ INDEX ]
  [ index_name | NOT NAMED ] [ ( column_name [,...] ) ] [
alter_partitioning ]

```

alter_partitioning

```

{ { PARTITION BY { partitioning_level | ( partitioning_level [,...] ) } |
  { DROP range_expression [ ADD range_expression ] |
    ADD range_expression
  } [,...] [ WITH { INSERT [ INTO ] save_table | DELETE } ] |

  NOT PARTITIONED
}

```

add_option

```

{ { column_specification | ( column_specification [,...] ) } [ INTO
column_name ] |

  [ COLUMN | ROW | SYSTEM ] ( { column_name | column_specification
[,...] } )
  [ [NO] AUTO COMPRESS ] |

  ( column_name ) [NO] AUTO COMPRESS |

  PERIOD FOR period_name ( period_begin , period_end ) |

  [ CONSTRAINT name ]
  { FOREIGN KEY ( column_name [,...] ) references |
    { UNIQUE | PRIMARY KEY } ( column_name [,...] ) |
    CHECK ( boolean_condition ) |
  } |

  row_level_security_constraint_column_name [,...] CONSTRAINT
}

```

drop_option

```

{ PERIOD FOR period_name |
  name [ IDENTITY ] |
  CONSTRAINT name |
  [ CONSTRAINT name ] FOREIGN KEY ( column_name [,...] ) references |
  [ [ CONSTRAINT ] name ] CHECK |
  INCONSISTENT REFERENCES |
}

```

```

    row_level_security_constraint_column_name [,...] CONSTRAINT
}

```

blockcompression

```

BLOCKCOMPRESSION = { AUTOTEMP | MANUAL | ALWAYS | NEVER | DEFAULT }
    [, BLOCKCOMPRESSIONALGORITHM = { ZLIB | ELZS_H | DEFAULT } ]
    [, BLOCKCOMPRESSIONLEVEL = { value | DEFAULT } ]

```

partitioning_level

```

{ partitioning_expression |
  COLUMN [ [NO] AUTO COMPRESS ] [ [ ALL BUT ] (
column_partition [,...] ) ]
} [ ADD constant ]

```

range_expression

```

{ RANGE | RANGE#Ln }
  { BETWEEN range [,...] [, { NO RANGE [ { OR | , } UNKNOWN ] |
UNKNOWN } ] |
  NO RANGE [ { OR | , } UNKNOWN ] |
  UNKNOWN |
  WHERE conditional_expression
}

```

column_specification

```

column_name data_type [ column_attribute [...] ]

```

references

```

REFERENCES [ WITH [NO] CHECK OPTION ] referenced_table_name
    [ ( referenced_column_name [,...] ) ]

```

column_partition

```
[ COLUMN | ROW ] { column_name | ( column_name [,...] ) }
[ [NO] AUTO COMPRESS ]
```

range

```
start_expression [ AND end_expression ] [ EACH range_size ]
```

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [( fractional_seconds_precision)] [WITH TIME
ZONE] |

  INTERVAL YEAR [( precision)] [TO MONTH] |

  INTERVAL MONTH [( precision)] |

  INTERVAL DAY [( precision)]
    [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL HOUR [(precision)]
    [TO { MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL MINUTE [(precision)] [ TO SECOND
[(fractional_seconds_precision)] ] |

  INTERVAL SECOND [(precision) [, fractional_seconds_precision)] |

  PERIOD (DATE) |

  PERIOD ({ TIME | TIMESTAMP } [(precision)] [ WITH TIME ZONE ]) |

  REAL |

  DOUBLE PRECISION |

  FLOAT [(integer)] |

  NUMBER [( { integer | * } [, integer ]...)] |
```

```

{ DECIMAL | NUMERIC } [(integer [, integer ]...)] |

{ CHAR | BYTE | GRAPHIC } [(integer)] |

{ VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [(integer)] |

LONG VARCHAR |

LONG VARGRAPHIC |

{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
(integer [ G | K | M ]) |

[SYSUDTLIB.] { XML | XMLTYPE } [(integer [ G | K | M ])]
[ INLINE LENGTH integer ] |

[SYSUDTLIB.] JSON [(integer [ K | M ])] [ INLINE LENGTH integer ]
[ CHARACTER SET { UNICODE | LATIN } | STORAGE FORMAT { BSON |
UBJSON } ] |

[SYSUDTLIB.] ST_GEOMETRY [(integer [ K | M ])] [ INLINE LENGTH
integer ] |

[SYSUDTLIB.] DATASET [(integer [ K | M ])] [ INLINE LENGTH integer ]
storage_format |

[SYSUDTLIB.] { UDT_name | MBR | ARRAY_name | VARRAY_name }
}

```

column_attribute

```

{ { UPPERCASE | UC } |
[NOT] { CASESPECIFIC | CS } |
FORMAT quotestring |
TITLE quotestring |
NAMED name |
DEFAULT { number | USER | DATE | TIME | NULL } |
WITH DEFAULT |
CHARACTER SET server_character_set |
[NOT] NULL |
[NOT] AUTO COLUMN |
NO COMPRESS |

```



```

COMPRESS [ constant | ( { constant | NULL } [,...] ) ] |
COMPRESS USING compress_UDF_name DECOMPRESS USING
decompress_UDF_name |
[ CONSTRAINT constraint_name ]
  { UNIQUE | PRIMARY KEY | CHECK ( boolean_condition ) | references }
}

```

storage_format

```

STORAGE FORMAT { Avro | CSV [ CHARACTER SET { UNICODE | LATIN } ] }
[ WITH SCHEMA [ database. ] schema_name ]

```

ALTER TABLE Syntax (Join Index)

```

ALTER TABLE [ database_name. | user_name. ] join_index j_add_option [,...] [;]

```

j_add_option

```

ADD { { COLUMN | ROW | SYSTEM } ( column_name ) [ [NO] AUTO
COMPRESS ] |
      ( column_name ) [NO] AUTO COMPRESS
}

```

ALTER TABLE Syntax (Revalidation)

```

[ NONTEMPORAL ] ALTER TABLE [ database_name. | user_name. ] { table_name |
join_index }
  REVALIDATE [ WITH { INSERT [ INTO ] save_table | DELETE } ] [;]

```

ALTER TABLE Syntax (Release Rows)

```

ALTER TABLE [ database_name. | database_name. ] { table_name | join_index_name }
  RELEASE [ DELETED ] ROWS [ AND RESET LOAD IDENTITY ] [;]

```

ALTER TABLE Syntax Elements

ALTER TABLE Syntax Elements (Basic)

database_name

Containing database. The default is the database for the current session.

user_name

Containing user name. The default is the database for the current user.

table_name

Name of the table to be altered. This can be a base data table or a user-defined join index, but cannot be an error table, a volatile table, or a system-defined join index.

ALTER TABLE Basic Options

You cannot modify or revalidate a table or join index in the same ALTER TABLE request that modifies basic table parameters.

NORMALIZE

Increases individual date ranges in the column to provide an overall range.

You can only add one NORMALIZE option for a table.

You cannot normalize a volatile table.

To normalize a table defined with columns of the JSON or DATASET data type, you must explicitly specify those columns as members of the ALL BUT (*normalize_ignore_column_name*) column set.

ADD

Add the specified *normalize_column* to *table_name* as a constraint for normalization purposes.

ON *normalize_column*

Column to add as a constraint for normalization purposes.

ON OVERLAPS

When date ranges in the column overlap.

ON MEETS OR OVERLAPS**ON OVERLAPS OR MEETS**

When date ranges in the column overlap or meet.

ALL BUT (*normalize_ignore_column_name*)

Add table columns to the list of columns that are not to be considered for normalization.

DROP

The NORMALIZE constraint is to be removed from the *normalize_column*.

MODIFY PRIMARY

Add or modify a primary index or primary AMP index. You cannot modify basic table parameters or revalidate a table or join index in the same ALTER TABLE statement that includes MODIFY options.

index_name

Name for an unnamed primary index or a change to the name of a primary index.
For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

(*index_column_name*)

Change the properties of the primary index for table. For a composite primary index, *index_column_name* indicates a comma-separated list of all the index columns in parenthesis. You cannot specify the begin or end columns of a derived period column in a primary index.

The table must be empty.

You cannot alter a table to have a row-level security constraint column as a component of its primary index.

You cannot define a primary index on a column with a JSON or DATASET data type.

AMP

Rows are hash-distributed to AMPs for a column-partitioned table or join index. Column partition values are ordered on each AMP by an internal partition number and a row hash for a column-partitioned table or join index.

Optionally, the INDEX keyword can be specified with AMP for readability.

If a PRIMARY AMP clause is specified, you must specify a PARTITION BY clause that includes a column-partitioning level, either in the PRIMARY AMP clause or by itself in the index list.

INDEX

If a PRIMARY INDEX clause and a PARTITION BY clause are specified, a column-partitioning level may be included in the PARTITION BY clause. The PARTITION BY clause can be included in the PRIMARY INDEX specification or by itself in the index list. The PARTITION BY clause can include row-partitioning levels.

NOT NAMED

Drop the name of a named index.

UNIQUE

Change a nonunique primary index to a unique primary index. The table must be empty.
A temporal table cannot have a unique primary index.

NOT UNIQUE

Change a unique primary index to a nonunique primary index. The table must be empty.

MODIFY NO PRIMARY

Remove the primary index or primary AMP index from the table definition. The table can contain rows when dropping or adding ranges under certain conditions. Otherwise, the table must be empty. If you specify NO PRIMARY INDEX and a PARTITION BY clause, the partitioning expression must specify a COLUMN partitioning level.

INDEX

Optional keyword.

MODIFY partitioning

Change partitioning for a column-partitioned table or join index.

Modifies the partitioning for a primary-indexed table, but not the primary index.

You can only alter the partitioning of a table that contains rows by dropping or adding row partitions at the beginning or end of a RANGE_N partitioning expression.

FROM TIME ZONE

Update columns with a TIME, TIMESTAMP, PERIOD(TIME), or PERIOD(TIMESTAMP) data type from one time zone setting to another.

You can only use this option on permanent tables.

Join indexes are converted implicitly as the base table is updated.

Update or alter table triggers are ignored.
You cannot include the USING clause.

sign

Enter a plus (+) sign to indicate a positive offset.

Enter a minus (-) sign to indicate a negative offset.

'quotestring'

Name of time zone.

TIMEDATEWZCONTROL =*n*

A value for the DBS Control setting. The default is 0. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

WITH TIME ZONE

Convert TIME WITH TIME ZONE and TIMESTAMP WITH TIME ZONE columns when upgrading a system with TIMEDATEWZCONTROL = 3 from Vantage pre-13.10.03 versions.

DOWN TABLE Option

Vantage attempts to avoid database crashes whenever possible by converting them to a transaction abort and a snapshot dump for a selected set of file system errors.

The system marks a table as down if its maximum number of permissible down regions is exceeded on its data subtable on any AMP in the system.

DOWN

Down status flag on all AMPs in the system, making the specified table inaccessible for DML requests.

SET

Set the down status flag on all AMPs in the system, making the specified table inaccessible for DML requests.

RESET

Remove the down status flag from all AMPs in the system and makes the specified table accessible for DML requests.

ALTER TABLE Syntax Elements (Join Index)

database_name

The name of the containing database. The default is the database for the current session.

user_name

The name of the containing user. The default is the database for the current user.

join_index

The name of the join index to be modified.

ALTER TABLE Join Index Options

You can specify the following options for a join index.

ADD COLUMN (column_name)

The column partition containing the column uses COLUMN format, if it is not already defined with COLUMN format.

You can include columns in the join index with a data type of non-LOB XML, non-LOB ST_GEOMETRY, non-LOB JSON, and non-LOB DATASET. However, you cannot include these data types in the primary index of a join index.

A column in a join index cannot have a data type of BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, VARIANT_TYPE, ARRAY, VARRAY, LOB XML, LOB ST_GEOMETRY, LOB JSON, or DATASET LOB.

AUTO COMPRESS

The column partition containing the column use COLUMN format with autocompression.

NO AUTO COMPRESS

The column partition containing the column use COLUMN format without autocompression.

ADD ROW (column_name)

The column partition containing the column use ROW format, if it is not already defined with ROW format.

AUTO COMPRESS

The column partition containing the column use ROW format with autocompression.

NO AUTO COMPRESS

The column partition containing the column use ROW format without autocompression.

ADD SYSTEM (column_name)

Alters the column partition containing the column to have a system-determined format of COLUMN or ROW.

AUTO COMPRESS

Alters the column partition containing the column to have a system-determined format of COLUMN or ROW and adds autocompression.

NO AUTO COMPRESS

NO AUTO COMPRESS

alters the column partition containing the column to have a system-determined format of COLUMN or ROW and removes autocompression.

ADD (column_name) AUTO COMPRESS

Add autocompression for the column partition. You can add compression to a set of existing columns whether or not the table is empty or contains data. You can only specify autocompression for column-partitioned tables.

You cannot modify the autocompression or partition storage for a column partition more than once in the same ALTER TABLE request.

AUTO COMPRESS

Applies autocompression to the column or column partition. AUTO COMPRESS is the default. Vantage applies autocompression for a physical row on a per container basis. For efficiency, the system may use the autocompression method chosen for the previous container, including not using autocompression, if that is more effective.

NO AUTO COMPRESS

The column partition containing the column is not autocompressed.

ALTER TABLE Revalidation Options

You cannot modify either basic table parameters, the primary index, or partitioning in the same ALTER TABLE request you execute to revalidate a table or join index.

REVALIDATE

Update the data dictionary as needed. This also regenerates the table headers for a partitioned table.

You can optionally verify row partitioning and correct any errors that are found during the verification process for a row-partitioned table by specifying either the WITH DELETE or WITH INSERT INTO *save_table* options.

WITH DELETE

Delete any row for which a partitioning expression evaluates to a value outside the valid range of 1 through the number of partitions defined for that level.

This option is valid only for row-partitioned tables. You cannot specify the WITH clause for join indexes.

WITH INSERT

Insert into *save_table* any row for which a partitioning expression evaluates to a value outside the valid range of 1 through the number of partitions defined for that level. The non-valid row is inserted into the *save_table* and deleted from the source table.

This option is valid only for row-partitioned tables. You cannot specify the WITH clause for join indexes.

INTO

Optional keyword to precede save table specification.

save_table

save_table and the table being altered must be different tables. Also, the *save_table* must have the same number of columns as the table being modified and the column data types must match. You can use this option with row-level security-protected tables as long as the tables referenced in the request are all row-level security-protected and they are all be defined with the same row-level security constraints. If you do not specify the constraint values to be inserted into the target table, Vantage takes the constraint values for the target table from the source table.

ALTER TABLE Release Rows Options

Removes the logically deleted rows from a load isolated (LDI) table or a join index. You use this option to reclaim the space occupied by the deleted rows. You should collect statistics when you perform this operation on an LDI table or a join index. Reclaiming space on a regular basis improves query performance on the LDI table or a join index. This operation physically deletes those rows that are marked as logically deleted. The corresponding secondary indexes are maintained.

You must have the DROP TABLE privilege to perform this operation.

An EXCLUSIVE lock is placed on the table to perform the deletion.

DELETED

Optionally, the DELETED keyword can be specified for readability.

AND RESET LOAD IDENTITY

The RowLoadID of the rows is set to 0. The version number of the table is incremented and the CurrLoadID value is set to 0.

If this option is not specified, the version number of the table is not changed.

Alter Options

You must specify at least one table option or one alter option for a basic ALTER TABLE request.

You can specify only table options, only alter options, or you can specify table and alter options. If you specify table and alter options, you must first specify table options and then alter options.

You cannot modify or revalidate a table or join index in the same ALTER TABLE request you execute to modify basic table parameters.

ADD *column_name*

Add the column or modify the column attributes.

ADD *column_name* modifies the column, if the table already contains the column or creates the column, if the table does not contain the column.

column_name

The name of a column to add or change.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

You cannot ADD a column that has the same name as named collected statistics.

You cannot add an identity column to an existing table, nor can you add the identity column attribute to an existing column.

ADD and DROP cannot both be specified on the same column in the same ALTER TABLE request.

data_type

To add a column, you must specify a data type. To add a new column and, in certain cases, modify the data type and column attributes of an existing column, use this syntax:

```
ADD column_name
data type column attributes
```

You can add a column with the NUMBER data type, increase the precision of an existing fixed NUMBER column, or increase the precision and scale of a fixed NUMBER data type. You can only modify the scale and precision for fixed NUMBER columns.

For a floating NUMBER column, you cannot perform the following modifications:

- Decrease the precision.
- Decrease the scale.
- Increase the scale without also increasing the precision.
- Increase the scale and precision by different amounts.

If you do not specify explicit formatting, the new column assumes the default format for the data type, which can be specified by a custom data formatting specification (SDF) defined by the `tdlocaledef` utility. See *Teradata Vantage™ - Database Utilities*, B035-1102.

Explicit formatting applies to the parsing and to the retrieval of character strings.

For information on data types and data type attributes, see *Teradata Vantage™ - Data Types and Literals*, B035-1143. To increase the size of a BLOB, CLOB, or XML column, use this syntax:

```
ADD column_name data type column attributes
```

Note:

You cannot decrease the size of a column.

You cannot use ALTER TABLE to change the size, character set, or storage format of a column defined with the JSON data type.

column attributes

To modify the attributes of an existing column, use this syntax:

```
ADD column_name column attributes
```

INTO *column_name*

Add the column set as a new column partition for *table_name* or to add the specified column set to the existing column partition.

You use parentheses to group columns together into the same column partition.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

INTO *column_name* specifies a column in an existing column partition. The new columns are added to this column partition.

If you do not specify INTO *column_name*, the new partition contains all of the columns in the group.

You cannot use an ALTER TABLE request on a join index to add columns or column partitions.

You cannot include columns with the JSON data type in a column partition.

ADD COLUMN (*column_name*)

Add the column set as a new column partition.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

The table must be column-partitioned and the columns being added cannot already exist. You cannot alter a join index to add columns or column partitions using an ALTER TABLE request. Specifies column storage format for the *column_name* or *column_name* group. Alters the column partition containing the column to have COLUMN format, if it is not already defined with COLUMN format. If you do not specify COLUMN, ROW, or SYSTEM format and the column partition containing the column:

- Does not have a system-determined format, the column partition is not altered and the format remains as most recently defined.
- Has a system-determined format, the format remains system-defined.

If the altered column partition containing the column has a system-determined column partition format, the column partition is altered to a newly appropriate format if it does not already have that format.

data type

Data type for the column.

column attributes

Specify attributes for the column.

AUTO COMPRESS

Autocompress data in the column.

NO AUTO COMPRESS

Do not autocompress data in the column.

ADD ROW (*column_name*)

Add row storage format for the *column_name* or *column_name* group.

ADD ROW (*column_name*) alters the column partition containing the column to have ROW format, if it is not already defined with ROW format.

data type

Data type for the column.

column attributes

Specify attributes for the column.

AUTO COMPRESS

Autocompress data in the column.

NO AUTO COMPRESS

Do not autocompress data in the column.

ADD SYSTEM (*column_name*)

Adds a system-determined storage format of COLUMN or ROW for the *column_name* or *column_name* group.

ADD SYSTEM (*column_name*) alters the column partition in which the column exists to have a system-determined format of COLUMN or ROW, if it is not already defined with SYSTEM format.

If you do not specify a storage format, Vantage defaults to SYSTEM.

data type

Data type for the column.

column attributes

Specify attributes for the column.

AUTO COMPRESS

Autocompress data in the column.

NO AUTO COMPRESS

Do not autocompress data in the column.

ADD (*column_name*) AUTO COMPRESS

Add autocompression for the column partition.

You can add compression to a set of existing columns whether or not the table is empty or contains data. You can only specify autocompression for column-partitioned tables. AUTO COMPRESS applies autocompression to the column or column partition. AUTO COMPRESS is the default. Vantage applies autocompression for a physical row on a per container basis. For efficiency, the system may use the autocompression method chosen for the previous container, including not using autocompression, if that is more effective.

You cannot modify the autocompression or partition storage for a column partition more than once in the same ALTER TABLE request.

NO AUTO COMPRESS

The column partition containing the column is not autocompressed.

ADD PERIOD FOR

A derived period column.

A DATE or timestamp column can be part of only one derived period column definition and can only be used as the start of the period or end of the period, not both. The data type of begin and end columns in a derived period column must match, including the same properties, the same DATE or timestamp data type, and identical format column attribute.

A table can have more than one derived period column. For each derived period column you add, the maximum limit of 2048 non-derived period columns is reduced by one. You cannot change the NOT NULL attribute, if specified, of the begin and end columns in a derived period column.

A derived period column can only include a period arithmetic operation in a WHERE or ON clause.

A derived period column cannot be:

- Projected
- Part of a primary or secondary index
- Specified as part of a PARTITION BY COLUMN clause

period_name

Name of the derived period column. *period_name* cannot match the name of any other column in the table.

period_begin_column

The DATE or timestamp column to use as the start of the period.

period_end_column

The DATE or timestamp column to use as the end of the period.

You can specify UNTIL_CHANGED.

ADD CONSTRAINT *name*

Add a named constraint.

If you alter a nontemporal table that is defined with CHECK constraints into a temporal table with either valid time or transaction time, Vantage automatically converts the definitions for those existing constraints into CURRENT CHECK constraints.

However, if there are any existing UNIQUE or PRIMARY KEY constraints defined on the nontemporal table, the system returns an error to the requestor.

You cannot place a constraint on a column with the JSON data type.

ADD *row_level_security_constraint_name* CONSTRAINT

Add a row-level security constraint. The system generates a column with the same name as the security constraint.

The following rules apply when adding a security constraint column:

- You can ADD up to 5 row-level security constraint columns per ALTER TABLE request.
- The constraint object must already exist in the database.
- The specified constraint must not have the same name as another column in the table.
- You cannot specify any column attributes for the column (for example, name, data type or nullability). The constraint column uses the values specified for the constraint object.
- A row-level security constraint column cannot be specified as a component of the primary index or partitioning expression for a table.
- A row-level security constraint can be defined as a component of a secondary index.
- You cannot add a security constraint to a table with a hash index or a join index.
- A table can have a maximum of 5 security constraints.
- For newly added security constraint columns the system uses the session label for the ALTER TABLE request as the column value for each row.

- You cannot define a CHECK or UNIQUE constraint on a security constraint column.
- If a table protected by row-level security is defined with a CHECK or UNIQUE constraint, enforcement of the constraint bypasses any security constraints defined for the table.
- You cannot apply the COMPRESS phrase to a row-level security constraint column.

Note:

When using referential integrity (RI) on tables that define security constraint columns, the system does not recognize security constraints when checking referential integrity for either the parent and child RI table. Execution of any request that access such RI tables continues as if the tables had no row-level security constraints.

ADD FOREIGN KEY REFERENCES

Add a foreign key reference to a key in another table.

You must specify FOREIGN KEY *referencing_column_name* before you specify REFERENCES *referenced_table_name* (*referenced_column_name*).

For an unnamed FOREIGN KEY REFERENCES table constraint, use this syntax:

```
ADD FOREIGN KEY (referencing_column) REFERENCES
               referenced_table_name(referenced_column_name)
```

For a named FOREIGN KEY REFERENCES table constraint, use this syntax:

```
ADD CONSTRAINT constraint_name FOREIGN KEY
               (referencing_column) REFERENCES
               referenced_table_name (referenced_column_name)
```

(*column_name_list*)

One or more columns that reference a primary key or alternate key in *referenced_table_name*.

table_name

Referenced table name is the name of the table that contains the primary key or alternate key referenced by the *referencing_column* set.

You must either have the REFERENCES privilege on the referenced table or on all specified columns of the referenced table.

A table can have a maximum of 64 foreign keys and a maximum of 64 referential constraints.

A maximum of 64 other tables can reference a single table. A maximum of 128 reference indexes can be stored in the table header per table. However, only 64 of these, the

reference indexes that map the relationship between the table and its child tables, are stored per reference index subtable.

The table header limit on reference indexes includes both references to and from the table.

column_name

Referenced column name is the column set in *referenced_table_name* that is referenced by the *referencing_column* set.

CONSTRAINT

Keyword for a named constraint.

constraint_name

Name of the table attribute foreign key constraint. The *constraint_name* is optional. You can specify a foreign key REFERENCES constraint as a table attribute or a column attribute. Vantage uses a different syntax for the two forms of foreign key. Foreign key REFERENCES constraints can be:

- null
- unique, but this is rare. An example of a unique foreign key is a logical table that is vertically partitioned into multiple tables.

You cannot add foreign key constraints on columns having any of the following data types:

- BLOB
- CLOB
- ARRAY/VARRAY
- Period
- UDT
- XML
- Geospatial
- JSON
- DATASET

You cannot specify a foreign key constraint on a row-level security constraint column.

You can specify a mix of standard referential integrity constraints, batch referential integrity constraints, and Referential Constraints for the same table, but not for the same column sets.

For information on the various types of constraints, see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

Although you can create a child table before the parent table has been created, a REFERENCES constraint that makes a forward reference to a table that has not yet been created cannot qualify the parent table name with a database name.

The forward-referenced parent table that has not yet been created must be in the same database as child table being created.

Each column in the foreign key *referenced_column_name* list must correspond to a column of *referenced_table_name* in REFERENCES *table_name*, and you cannot specify the same column name more than once.

The foreign key column list must contain the same number of column names as the referenced primary or alternate key in *table_name*. The *i* th column of the referencing list corresponds to the *i* th column identified in the referenced list.

The data type of each foreign key referencing column must match the data type of the corresponding REFERENCES column.

Each foreign key can be defined on a maximum of 64 columns.

You can define a maximum of 100 table-level constraints for a table.

You cannot specify a foreign key REFERENCES constraint on:

- an identity column
- global temporary trace table
- global temporary table
- volatile table
- queue table

See [CREATE TABLE \(Queue Table Form\)](#).

Foreign key REFERENCES constraints cannot be copied to a new table using the CREATE TABLE AS syntax.

WITH CHECK OPTION

A referential integrity constraint. The integrity of the relationship is checked only when the entire transaction of which it is a component completes.

For a table attribute foreign key constraint, specify this clause:

```
FOREIGN KEY (referencing_column_name) REFERENCES
           referenced_table_name
           (referenced_column_name)
           WITH CHECK OPTION
```

If any of the rows in the transaction violate the Referential Integrity rule, then the entire transaction is rolled back.

See *Teradata Vantage™ - Database Design*, B035-1094 for information about the Referential Integrity rule.

This option is not valid for temporal tables. For details, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

This clause is a Teradata extension to the ANSI SQL:2011 standard.

ADD CHECK (*boolean_condition*)

An unnamed CHECK column constraint as a table attribute, where *column_name* is the left-hand side of *boolean_condition*. A table attribute CHECK constraint can compare columns in the table and can compare column contents with constants. You can define a maximum of 100 table-level constraints for a table.

You cannot add a CHECK constraint on a column with the JSON data type.

CONSTRAINT *constraint_name*

Named CHECK column constraint as a table attribute. Specify a *column_name* on the left-hand side of *boolean_condition*.

ADD UNIQUE (*column_name*)

A column set as the UNIQUE constraint column set for *table_name*. The columns must be defined as NOT NULL.

If you specify more than one *column_name*, the UNIQUE column set is based on the combined values of the specified column name.

For an unnamed UNIQUE table constraint, use this syntax:

```
ADD UNIQUE (column_name)
```

You cannot add a UNIQUE constraint on a column with the JSON data type.

CONSTRAINT *name*

For a named UNIQUE table constraint, use this syntax:

```
ADD CONSTRAINT constraint_name UNIQUE (column_name)
```

A UNIQUE table constraint can be defined on a maximum of 64 columns.

ADD PRIMARY KEY (*column_name*)

Column set is the primary key for *table_name*. The defined column set makes each row in the table unique. Columns in a PRIMARY KEY constraint must be defined as NOT NULL. You can specify a PRIMARY KEY constraint as a column attribute or as a table attribute. You can specify a PRIMARY KEY constraint on a single column or on a composite column set of up to 64 columns. If you specify more than one column, the primary key column set is based on the combined values of the specified columns. To add an unnamed PRIMARY KEY table constraint, use this syntax:

```
ADD PRIMARY KEY (column_name)
```

You cannot specify a PRIMARY KEY constraint for volatile tables. You cannot specify PRIMARY KEY constraints on columns with the following data types:

- BLOB
- BLOB UDT
- CLOB
- CLOB UDT
- VARIANT_TYPE
- ARRAY
- VARRAY
- Period
- XML
- Geospatial
- JSON
- DATASET

You cannot specify a PRIMARY KEY constraint on a row-level security constraint column.

You can only specify one primary key per table. The primary key enforces referential constraints. To specify candidate primary keys for referential integrity relationships with other tables, use the UNIQUE column attribute. This is not necessary for Referential Constraints, but is required for standard referential integrity constraints and batch referential integrity constraints.

To enforce the constraint, the PRIMARY KEY table attribute implicitly uses a unique secondary index or UPI for nontemporal tables and a single-table join index for most temporal tables. For details, see *Teradata Vantage™ - Temporal Table Support*, B035-1182. These system-defined secondary or single-table join indexes are included in the maximum of 32 secondary, hash, and join indexes per table.

Like UNIQUE constraints, PRIMARY KEY constraints ensure that the uniqueness of alternate keys when they are specified as part of a referential integrity relationship.

If you do not specify a primary index, the implicitly defined index is:

- A unique primary index for a nontemporal table.
- Not defined, and the table is a NoPI or a column-partitioned table.

If you specify a primary index, the implicitly defined index is a:

- Secondary index for a nontemporal table.
- System-defined single-table join index for a temporal table. For details, see *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

PRIMARY KEY constraints are valid for nontemporal and temporal tables. For details about temporal tables with PRIMARY KEY constraints, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

CONSTRAINT *name*

To add a named PRIMARY KEY table constraint, use this syntax:

```
ADD CONSTRAINT constraint_name PRIMARY KEY (column_name)
```

MODIFY CHECK (*boolean_condition*)

Change an existing CHECK constraint. The table or column constraint must already exist. Vantage scans existing rows of the table to validate that the current values conform to the specified constraint. Otherwise, the system returns an error to the requestor and does not change the table definition.

CONSTRAINT *name*

Named CHECK column constraint as a table attribute. Specify a *column_name* on the left-hand side of *boolean_condition*.

RENAME

Change a column or constraint name.

You can rename a column and add a new column to the same table using the old name for the renamed column. However, you must rename the old column before creating a new column with the previous name.

old_column_name

Current name of the column.

old_constraint_name

Current name of the constraint.

AS

Keyword that precedes new name.

TO

Keyword that precedes new name.

new_column_name

New name for the column.

new_constraint_name

New name for the constraint.

DROP PERIOD FOR

Remove the derived period column.

When you drop a derived period column from a table, the begin column and end column of the derived period column are not dropped.

period_name

Name of the derived period column.

DROP *column_name*

Removes the named column from the table.

You cannot drop the QITS column from a queue table. For a column-partitioned table, Vantage drops the specified column from its column partition.

If the specified column is the only column in the column partition, and the partition is not the only column partition, Vantage drops the partition. You cannot drop a column partition with a single column when it is the only column partition defined for the table or join index. You cannot drop a column in use as the begin column or end column of a derived period column.

For information about dropping temporal columns, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

IDENTITY

Removes the identity column attribute for the named column from the table.

The specification of IDENTITY is optional.

You can drop an identity column from an existing table.

When you use Teradata Unity, you cannot do a bulk data load insert of rows into a table that has an identity column. However, you can specify this option to drop only the identity column attribute while retaining the column and data. This is useful when the identity column for a table is also its unique primary index. Teradata Unity provides its own mechanism for generating identity column-like values for tables that require the identity column functionality. For additional information, see the Teradata Unity documentation.

DROP CONSTRAINT *name*

Drop a table-level named constraint on the specified table.

If the constraint is not named, you can drop the system-defined USI or single-table join index using a DROP INDEX or DROP JOIN INDEX request as appropriate. See [DROP INDEX](#) or [DROP JOIN INDEX](#).

DROP *row_level_security_constraint_name* CONSTRAINT

Drop a row-level security constraint column from the table.

Note:

Before you drop a row-level security constraint column from a table you must:

- As with other table columns, you must remove a security constraint column from any views that reference the base table before dropping the constraint column.
- Drop any indexes that reference the table and then recreate the indexes without the security constraint column.

Note:

Hash indexes and join indexes are not supported on row-level security tables.

DROP FOREIGN KEY REFERENCES

Drop a foreign key reference to a key in another table. For an unnamed FOREIGN KEY REFERENCES table constraint, use this syntax:

```
DROP FOREIGN KEY (referencing_column) REFERENCES
    referenced_table_name(referenced_column_name)
```

For a named FOREIGN KEY REFERENCES table constraint, use this syntax:

```
DROP CONSTRAINT constraint_name FOREIGN KEY
    (referencing_column) REFERENCES
    referenced_table_name (referenced_column_name)
```

(*column_name_list*)

Referencing column set that includes one or more columns in table that reference a primary key or alternate key in the referenced table.

***table_name* (*column_name*)**

Referenced table that contains the primary key or alternate key referenced by the referencing column set and the *column_name* that is referenced by the referencing column set.

CONSTRAINT *constraint_name*

Name of the table attribute foreign key constraint.

DROP CHECK

Drop all unnamed table-level constraints on the table.

CONSTRAINT *name*

Named CHECK constraint on the table to drop.

name

Drop a named CHECK constraint on the table.

DROP INCONSISTENT REFERENCES

Delete all inconsistent references defined on the table.

DROP INCONSISTENT REFERENCES is commonly used after a restore, when an ALTER TABLE DROP foreign key constraint might not work.

This clause is a Teradata extension to the ANSI SQL:2011 standard.

Table Options

The ALTER TABLE options are a Teradata extension to ANSI SQL. You can list the options in any order.

You must specify at least one table option or one column option for a basic ALTER TABLE request.

You can alter table options, column options, or both. If you alter both, you must alter table options, then column options.

You cannot modify basic table parameters and revalidate a table or join index in the same ALTER TABLE request.

FALLBACK

Add duplicate copy protection for the table or user-defined join index. Adding fallback creates and stores a duplicate copy of the table.

The FALLBACK default for a base data table is determined by the CREATE TABLE request when the table is created or the default fallback specification for the database or user containing the table. You can override these defaults with an ALTER TABLE request.

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

An ALTER TABLE request that changes a table with block-level compression and also adds FALLBACK does not pass the block-level compression characteristics of the primary table for the newly created fallback table by default.

Vantage assigns block-level compression to a newly created fallback table, depending on the request. If the request changes the row definition, adds fallback, and also specifies the BlockCompression query band, then the fallback table:

- Has block-level compression if the value for BlockCompression is set to YES.
- Does not have block-level compression if the value for BlockCompression is set to NO.

See SET QUERY_BAND in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

If the request does not specify the BlockCompression query band, the fallback table defaults to the system-wide compression characteristics defined by the compression columns of the DBS Control record. See *Teradata Vantage™ - Database Utilities*, B035-1102.

When a hardware read error occurs, the file system reads the fallback copy of the data and reconstructs the rows in memory on their home AMP. Read From Fallback applies to:

- Requests that do not attempt to modify data in the bad data block
- Primary subtable data blocks
- Reading the fallback data in place of the primary data. In some cases, Active Fallback can repair the damage to the primary data dynamically. In situations where the bad data block cannot be repaired, Read From Fallback substitutes an error-free fallback copy of the corrupt rows each time the read error occurs.

For a system-defined join index, the FALLBACK modification you make on a base table also applies to any system-defined join indexes defined on that table. You alter the FALLBACK for a system-defined join index directly.

PROTECTION

Optional default keyword.

NO

Fallback protection for the table is removed. Removing fallback deletes the existing duplicate copy.

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

WITH JOURNAL TABLE

A definition for the permanent journal table for the data table being altered.

The JOURNAL default for a base data table is determined either by the CREATE TABLE request that created the table or the default fallback specification for the database or user containing the table. You can override these defaults with an ALTER TABLE request.

Journal options are not supported for tables with row sizes greater than 64KB.

table_name

The permanent journal table *table_name* need not reside in the same database or user as the data table.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

database_name

The default database for the current session is assumed, unless you specify a database. The database must already exist and *table_name* must have been defined as its default permanent journal table.

user_name

The default database for the current user is assumed, unless you specify a user. The user must already exist and *table_name* must have been defined as its default permanent journal table.

JOURNAL

The number of BEFORE permanent journal change images to be maintained for the table. If the JOURNAL keyword is specified without NO or DUAL, then a single copy of the image is maintained unless FALLBACK is in effect or is also specified.

You cannot add or modify journal options for an nonpartitioned NoPI table or for a column-partitioned table.

Journal options are not supported for tables with row sizes greater than 64KB.

BEFORE

Optional keyword.

DUAL

If journaling is requested for a table that uses fallback protection, DUAL images are maintained automatically.

NO

Permanent journal change images are not maintained for the table.

LOG

Transient journaling is performed for global temporary and volatile tables. This option only pertains to global temporary and volatile tables.

Update, insert, or delete operations on the global temporary or volatile table are logged in the transient journal. This is the default.

You cannot change the LOG or NO LOG properties of a global temporary table if any materialized instances of the table exist.

NO

Transient journal logging of rows is not performed, reducing the system overhead of logging.

If an error or restart occurs and the table is defined as NO LOG, then any update, insert, or delete operations on the global temporary or volatile table cannot be recovered.

If the table is defined as NO LOG, a transient journal is generated for the transaction and the content of any materialized global temporary table or volatile table is emptied when a transaction aborts.

AFTER JOURNAL

The type of permanent journal image to be maintained for the table. Any existing images are not affected until the table is updated.

- If you specify the JOURNAL keyword without also specifying NO or DUAL, Vantage maintains a single copy of the image unless FALLBACK is in effect or is also specified.
- If you specify journaling for a database that uses fallback protection by default, Vantage maintains DUAL images automatically.

You cannot add or modify journal options for an nonpartitioned NoPI table or for a column-partitioned table.

NO

After-change images are not maintained for the table.

DUAL

Two after-change images are maintained for the table.

LOCAL

Single after-image journal rows for non-fallback data tables are written on the same virtual AMP as the changed data rows.

NOT LOCAL

Single after-image journal rows for non-fallback data tables are written on another virtual AMP in the cluster.

JOURNAL

The type of permanent journal image to be maintained for the table. The default is set by the CREATE DATABASE, CREATE USER, or MODIFY USER statement for the database in which the table resides. When you specify the JOURNAL keyword without also specifying AFTER or BEFORE, Vantage maintains both types of images. This option can appear twice in the same request: once to specify a BEFORE or AFTER image and again to specify the alternate type. You cannot add or modify journal options for a nonpartitioned NoPI table or for a column-partitioned table.

BEFORE JOURNAL

If you specify only one type of journaling, Vantage overrides only the default for that type. For example, if you specify AFTER, then BEFORE image journaling remains at its default setting. If you specify BEFORE and AFTER journaling, the two specifications must not conflict with one another.

ON COMMIT

You cannot change the ON COMMIT properties of a global temporary table if a materialized instance of the table exists.

DELETE ROWS

Delete the contents of an instance of a global temporary table when a transaction completes. DELETE ROWS clears an instance of a global temporary or volatile table of all rows. DELETE is the default.

PRESERVE ROWS

Preserve the contents of an instance of a global temporary table when a transaction completes. PRESERVE ROWS retains the rows in the table after the transaction commits.

CHECKSUM

A table-specific disk I/O integrity checksum for detection of hardware read errors. The checksum level setting applies to primary data rows, fallback data rows, and all secondary index rows for the table.

For a system-defined join index, the integrity checking setting you specify for a base table also applies to any system-defined join indexes defined on that table. You cannot specify integrity checking for a system-defined join index directly.

ON

Calculate checksums using the entire disk block. Sample 100% of the disk blocks to generate a checksum.

OFF

Disables checksum disk I/O integrity checks.

DEFAULT

The default setting is the current DBS Control checksum setting specified for this table type.

IMMEDIATE

The checksum setting is updated immediately for this table, including all data blocks and cylinder indexes. If you do not specify this option, the checksum setting is updated the next time the data blocks and cylinder indexes are updated and written to disk.

DEFAULT FREESPACE

The value for FREESPACE PERCENT is reset to the system default defined by DBS Control for this table.

Specifying this option does not immediately affect existing cylinders, but the new attribute value controls any new allocation of data blocks using this revised free space percentage.

Specifying the DEFAULT FREESPACE option when the percent freespace was previously set to a value other than the default value resets the size to the system default value.

FREESPACE

The percentage of free space to remain on a cylinder after a loading operation completes. This specification does not immediately affect existing cylinders, but the new attribute value controls any new allocation of data blocks using this revised free space percentage.

For additional information, see FREESPACE in CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

The FREESPACE setting you specify affects the following operations:

- FastLoad data loads
- MultiLoad data loads into unpopulated tables
- Table Rebuild

- System reconfiguration
- Ferret PACKDISK
- MiniCylPack

If few cylinders are available, and storage space is limited, MiniCylPack may not be able to honor the specified FREESPACE value.

- ALTER TABLE request that adds fallback protection to a table
- CREATE INDEX request that defines or redefines a secondary index on a populated table
- Creation of a fallback table during an INSERT ... SELECT operation into an empty table that is defined with fallback.
- Creation of a secondary index during an INSERT ... SELECT operation into an empty table that is defined with a secondary index.

integer

Integer is a value from 0 through 75.

integer PERCENT

Optional keyword to indicate value is a percentage.

MERGEBLOCKRATIO

This is the merge block ratio to be used for this table when Vantage combines smaller data blocks into a single larger data block.

DEFAULT

The value for MergeBlockRatio that is defined by the DBS Control record at the time a data block merge operation on this table begins.

You can specify DEFAULT MERGEBLOCKRATIO for global temporary and volatile tables.

Whether Vantage uses the merge block ratio you specify depends on the setting for the DBS Control parameter DisableMergeBlocks.

- If DisableMergeBlocks is FALSE and you specify a MERGEBLOCKRATIO, the system uses the value you specify. If you do not specify a MERGEBLOCKRATIO, the system uses the system-wide default setting for the DBS Control parameter MergeBlockRatio.
- If DisableMergeBlocks is TRUE, the system ignores all table-level settings for the merge block ratio and does not merge data blocks for any table in the system.

integer PERCENT

Merge block ratio when a data block merge operation occurs on this table, where *integer* is value from 1 through 100. The default value is 60.

You can only specify a numeric merge block ratio for permanent base tables and

permanent journal tables. You cannot specify a numeric merge block ratio for global temporary or volatile tables. This value of the merge block ratio does not affect the resulting block size when only a single block is modified. Setting the merge block ratio for a table to too high a value can cause the resulting merged block to require splitting during subsequent modifications.

PERCENT is an optional keyword to indicate value is a percentage.

NO

Data blocks for the table are not merged when Vantage combines smaller data blocks into a single larger data block.

You can specify NO MERGEBLOCKRATIO options for global temporary and volatile tables.

DATABLOCKSIZE

Maximum data block size for blocks that contain multiple rows as the value *data_block_size*. The calculation is the same as for the CREATE TABLE statement. For additional information, see DATABLOCKSIZE of CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184. For detailed information about setting system-wide default data block sizes using DBS Control settings, see *Teradata Vantage™ - Database Utilities*, B035-1102. For information about setting database-wide or user-wide default data block sizes, see [CREATE DATABASE](#) or [CREATE USER](#).

data_block_size

Decimal number or integer using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

BYTES

KBYTES

KILOBYTES

BYTES can be abbreviated as BYTE. The default is BYTES.

KILOBYTES can be abbreviated as KBYTE or KBYTES. For the rounding rules, see [CREATE TABLE and CREATE TABLE AS](#).

IMMEDIATE

Repackage data blocks into the newly specified size immediately instead of waiting until an affected block is modified or allocated. The new *data_block_size* value determines the maximum size of multiple-row data blocks subsequently modified or allocated.

- If you specify IMMEDIATE, the attribute value is changed and the rows in all existing data blocks are repacked into data blocks of the specified size.

- If you do not specify IMMEDIATE, the existing data blocks are not modified.

You cannot specify IMMEDIATE if you specify the BLOCKCOMPRESSION option.

MINIMUM

The smallest data block size for this table.

MINIMUM DATABLOCKSIZE sets the maximum data block size for blocks that contain multiple rows to the minimum value of 21,504 bytes (42 sectors) for systems with large cylinders or 9,216 bytes (18 sectors) for systems without large cylinders.

You can abbreviate MINIMUM as MIN.

For details about minimum data block sizes, see DATABLOCKSIZE of CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

MAXIMUM

The largest data block size for this table.

Systems that do not support block sizes 128KB and larger use a value of 127.5KB.

Systems that allow blocks larger than 127.5KB use a maximum of 1,048,064 bytes (2047 sectors) for large cylinders or 262,144 bytes (512 sectors) for small cylinders.

The value can be expressed either as a decimal number or integer number or using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

You can abbreviate MAXIMUM as MAX.

DEFAULT

The default data block size for this table. See *Teradata Vantage™ - Database Utilities*, B035-1102.

BLOCKCOMPRESSION

Table data is compressed at the block level.

You cannot specify this option to modify the definitions of global temporary tables or volatile tables.

For details, see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

AUTOTEMP

The file system determines the block-level compression setting for the table based on its Teradata Virtual Storage temperature.

Block-level compression is applied based on the temperature of the cylinders on which the data is stored. The file system determines the block-level compression setting

for the table based on its Teradata Virtual Storage temperature. The definitions of the various thresholds are determined by the DBS Control field TempBLCThresh. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102. You can adjust the temperature values for data being loaded using SET QUERY_BAND. See [Example: Setting BLOCKCOMPRESSION and TVSTEMPERATURE Query Bands](#).

MANUAL

Block-level compression is applied based on the default for the table at the time the table is created. The defaults for the table are determined by the settings in the Compression group of DBS Control fields. See *Teradata Vantage™ - Database Utilities*, B035-1102. You can override these values using SET QUERY_BAND. See [Example: Using the BLOCKCOMPRESSION Reserved Query Band](#).

Tables can be compressed or uncompressed at any time after loading by using the Ferret COMPRESS and UNCOMPRESS commands.

ALWAYS

The table and its subtables are always block-level compressed, even if a query band or the applicable DBS Control block-level compression settings indicate otherwise. The DBS Control field BlockLevelCompression must be enabled.

Tables can be compressed at any time after loading by using the Ferret COMPRESS command. This can be useful if this BLOCKCOMPRESSION option was set after the table was already populated.

NEVER

The table and its subtables are not block-level compressed, even if a query band or the applicable DBS Control block compression settings indicate otherwise.

Tables can be uncompressed at any time after loading by using the Ferret UNCOMPRESS command. This can be useful if this BLOCKCOMPRESSION option was set after the table was already populated.

DEFAULT

The table uses the block-level compression setting in the DBS Control field DefaultTableMode. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

BLOCKCOMPRESSIONALGORITHM

Specifies the algorithm to use for block-level compression (BLC).

This option only applies when the effective BLOCKCOMPRESSION is MANUAL, AUTOTEMP, or ALWAYS.

ZLIB

Block-level compression using the zlib software algorithm.

ELZS_H

Block-level compression using a hardware compression engine board in every system node.

DEFAULT

Uses the setting of the DBS Control field CompressionAlgorithm. See *Teradata Vantage™ - Database Utilities*, B035-1102.

BLOCKCOMPRESSIONLEVEL

Specifies a value to indicate a preference for compression speed or compression effectiveness. This option only applies when the BLOCKCOMPRESSIONALGORITHM option is set to ZLIB. Although this option is accepted in combination with BLOCKCOMPRESSIONALGORITHM = ELZS_H, this option is ignored. If BLOCKCOMPRESSIONALGORITHM is altered to ZLIB, this option takes effect.

value

Integer from 1 through 9, where 1 specifies the least processor-intensive compression speed with the lowest compression ratio and 9 specifies the most processor-intensive compression speed with highest compression ratio.

DEFAULT

The table uses the compression level setting of the DBS Control field CompressionLevel. See *Teradata Vantage™ - Database Utilities*, B035-1102.

WITH ISOLATED LOADING

Defines the table for load isolation (LDI). Load isolation enables concurrent read operations on committed rows while the table is being loaded.

The following types of tables cannot be defined as load isolated:

- Volatile table
- Error table
- Queue table
- Temporary table
- Global Temporary table
- Column partitioned table

NO

Defines the table as a non-load isolated table. This is equivalent to a table without this clause.

CONCURRENT ISOLATED LOADING

Enables concurrent read operations on committed rows while the table is being modified.

FOR ALL

All modifications can be concurrent load isolated.

FOR INSERT

Only INSERT operations can be concurrent load isolated.

FOR NONE

Concurrent load operations are not permitted.

USING FAST MODE

You can use this *table_option* with only two kinds of statements:

- ALTER TABLE statement with no *alter_option*
USING FAST MODE can only change the table format.
- ALTER TABLE statement whose only *alter_option* is ADD COLUMN
USING FAST MODE can change the table format and add columns.

The table format is either Fast Column Add (FCA) or non-FCA.

For a table in FCA format:

- Every ADD COLUMN statement on the table includes an implicit USE FAST MODE ON clause.
- If you update a column, columns with versions greater than the version of the updated column (*v*) do not materialize in the row if the row version is less than *v*.
- If you drop one or more columns, the table converts to non-FCA format. Missing values in remaining columns materialize in the rows, which may increase table size.

FCA format advantages:

- Dramatically reduces exclusive lock times that ADD COLUMN needs.
- Adds columns without increasing table space, because added columns do not materialize immediately in existing rows.

FCA format does not support the following:

- ADD COLUMN statements that increase row size from less than 64 KB to more than 1 MB
- Column-partitioned tables

- Dictionary tables
- VOLATILE tables
- GLOBAL TEMPORARY tables
- QUEUE tables
- ERROR tables
- Unity TD_Rowsize expression

Row size on managed servers can differ, and Unity does not support different values for the same field.

ON

Default when DBSControl flag FastAlterEnable has its default value, true.

If the table format is non-FCA, it becomes FCA.

If the ALTER TABLE statement includes one or more ADD COLUMN options, the statement adds the specified column or columns to the table without altering the existing rows and adds the default values of the new columns to the table header.

OFF

If the table format is FCA, it becomes non-FCA. FCA structures in the table header disappear, and missing values in remaining columns materialize in the rows, which may increase table size.

If the ALTER TABLE statement includes one or more ADD COLUMN options, the statement adds the specified column or columns to the table.

Column Attributes

You can specify the attributes listed below.

UPPERCASE

The column data is in uppercase format.

CASESPECIFIC

The column data is in case-specific format.

NOT

The column data is not in case-specific format.

FORMAT *quotestring*

The format string that must be valid for the external type of the UDT, the external type being its fromsql transform routine as defined either by default or by user definition using the CREATE TRANSFORM statement. See [CREATE TRANSFORM and REPLACE TRANSFORM](#).

If you do not specify a format, the system automatically applies the default display format of the external type.

TITLE *quotestring*

A title for reporting purposes.

The maximum size for a *column_name* TITLE is 256 characters.

This option is a Teradata extension to the ANSI standard.

NAMED *name*

Default name.

DEFAULT***number***

Value to use as a default.

USER

Default user.

DATE

Default date.

TIME

Default time.

NULL

Nullable column as the default.

WITH DEFAULT

All rows initially contain the system default in the field.

CHARACTER SET *server_character_set*

Character column definitions use the character set assigned as the default for the user with CREATE USER and MODIFY USER. You can explicitly override the default character set definitions by using the CHARACTER SET clause. For example, if the DEFAULT CHARACTER SET defined by CREATE USER is Unicode for a user, whenever that user creates or alters a table, the declaration CHARACTER(*n*) in CREATE TABLE or ALTER TABLE is equivalent to CHARACTER(*n*) CHARACTER SET UNICODE. See the information about the CHARACTER SET phrase in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

You cannot specify a character server data set of KANJI1.

NULL

The default is NULL except when you copy a table definition using the CREATE TABLE AS syntax, in which case the system carries the specification made for the source table over to the target table definition.

NOT

You must specify NOT NULL if you do not want nulls to be valid for the column.

AUTO COLUMN

Specifies that the column accepts shredded data not designated for the other columns in the table.

You can only specify the AUTO COLUMN option for a single JSON data type column in a table.

NOT

Removes the auto column capability from the column.

NO COMPRESS

The specified column is not compressed.

This is the default.

If you specify NO COMPRESS for an existing column defined with compression, then the compression attribute is dropped for the column.

You can specify NO COMPRESS for non-LOB based distinct and structured UDT columns, for ARRAY/VARRAY columns, and for Period data type columns.

COMPRESS

Use multivalue compression (MVC) for a set of distinct values in the column. You can also specify COMPRESS for a column in a volatile table. The default is to compress nulls only. You can specify multivalue and algorithmic compression in either order. For a detailed description of multivalue compression, see *Teradata Vantage™ - Database Design*, B035-1094.

For more information about the COMPRESS attribute, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

constant

Nulls and the specified constant values are compressed. You cannot compress nulls and constants if the column is defined as NOT NULL. Using COMPRESS on data can save space depending on the percentage of rows for which the compressed value is assigned. You can specify multivalue compression for all numeric types, BYTE, VARBYTE, DATE, CHARACTER, VARCHAR, GRAPHIC, VARGRAPHIC, TIME, TIME WITH TIME ZONE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE data types. See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for information about limits for this value.

NULL

Nulls are compressed for the column. The following rules apply:

- You cannot specify NULL if the column is defined as NOT NULL.
- LOB-based UDT nulls cannot be compressed.
- You can only specify NULL once per column.

COMPRESS USING

Algorithmic compression (ALC) for a column.

For a list of data types that can be algorithmically compressed, see the COMPRESS and DECOMPRESS phrases in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

You can also specify COMPRESS USING for a column in a volatile table. To specify COMPRESS USING *compress_udf_name*, the table cannot be populated with rows. If you specify COMPRESS USING, you must also specify DECOMPRESS USING. The COMPRESS USING and DECOMPRESS USING options can be specified in either order. When you specify algorithmic compression, the data is compressed, if possible, to a smaller amount of space than it currently requires. You can specify multivalue compression and algorithmic compression for the same column. Algorithmic compression is only applied to values that are not specified for multivalue compression. For a column-partitioned table, the column partition that includes the column can have autocompression. You can combine multivalue compression, algorithmic compression, autocompression (for a column-partitioned table), and block-level compression for the same table to achieve better compression, but as a general rule you should not use algorithmic compression with block-level compression because of the possibility of

a negative performance impact for other workloads. Teradata provides external UDFs for algorithmic compression and decompression. See the compression and decompression functions in *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

compress_UDF_name

Name of the UDF to algorithmically compress data in this column. The default containing database for *compress_UDF_name* is SYSUDTLIB. If *compress_UDF_name* is an embedded services UDF, then its default containing database is TD_SYSFNLIB. If *compress_UDF_name* is not contained in SYSUDTLIB or TD_SYSFNLIB, the system returns an error to the requestor.

DECOMPRESS USING

Algorithmically decompress data in this column. For a list of data types that can be algorithmically compressed, see the COMPRESS and DECOMPRESS phrases in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

To specify DECOMPRESS USING *compress_UDF_name*, the table cannot be populated with rows. The default containing database for *decompress_UDF_name* is SYSUDTLIB. If *decompress_UDF_name* is an embedded services UDF, then its default containing database is TD_SYSFNLIB. If *decompress_UDF_name* is not contained in SYSUDTLIB or TD_SYSFNLIB, Vantage returns an error to the requestor.

decompress_UDF_name

The name of the UDF to be used to algorithmically decompress data in this column.

UNIQUE

An unnamed UNIQUE constraint on a single column or a composite UNIQUE constraint on multiple columns of the table. A UNIQUE constraint enforces the rule that no two rows in the table can have the same value in a column set.

Columns must be defined as NOT NULL.

Any system-defined secondary or single-table join indexes used for this constraint are included in the maximum of 32 secondary, hash, and join indexes per table. This includes the system-defined secondary indexes used to implement UNIQUE constraints for nontemporal tables and the single-table join indexes used for UNIQUE constraints on temporal tables.

A UNIQUE constraint is a unique secondary index for nontemporal tables and a single-table join index for most temporal tables.

UNIQUE constraints ensure the uniqueness of alternate keys. Columns with UNIQUE constraints can be used to create referential integrity relationships with other tables. For details about UNIQUE constraints on temporal tables, see *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

You cannot specify UNIQUE constraints on columns with the following data types.

- BLOB
- CLOB
- ARRAY
- VARRAY
- Period
- XML
- Geospatial
- JSON
- DATASET

If the current data violates the uniqueness constraint, the system returns an error to the requestor and the table is not changed.

CONSTRAINT *constraint_name*

A named UNIQUE column constraint.

PRIMARY KEY

The column as the primary key for the table. The column must be defined as NOT NULL. For an unnamed PRIMARY KEY column constraint, use this syntax:

```
PRIMARY KEY
```

Use the column attribute form of PRIMARY KEY when the constraint applies only to the column and the primary key for the table is defined only on the single column. Tables can have only one primary key. To specify primary keys for referential integrity relationships with other tables, alter the column to have a UNIQUE constraint. Any system-defined secondary or single-table join indexes used to implement this constraint are included in the maximum of 32 secondary, hash, and join indexes per table. A PRIMARY KEY constraint is a unique secondary index or UPI for nontemporal tables and a single-table join index for most temporal tables. For details and examples of PRIMARY KEY constraint on temporal tables, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

You cannot include a column with the JSON data type in a PRIMARY KEY.

You cannot add a PRIMARY KEY constraint on a row-level security constraint column.

CONSTRAINT *constraint_name*

For a named PRIMARY KEY column constraint, use this syntax:

```
CONSTRAINT constraint_name PRIMARY KEY
```

CHECK (*boolean_condition*)

An unnamed CHECK column constraint, where *column_name* is the left-hand side of *boolean_condition*.

The following rules apply only to column attribute CHECK constraints.

- A column attribute CHECK constraint cannot reference other columns in the same table or in another table.
- You can specify multiple CHECK constraints on a single column. Multiple unnamed column-level CHECK constraints are combined into a single column-level CHECK constraint. Multiple named column-level CHECK constraints are applied individually.

You can specify column-level CHECK constraints on nontemporal and temporal tables. For details about how temporal tables support CHECK constraints, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

You can specify CHECK constraints as column attributes or as table attributes.

You cannot specify subqueries, aggregate, or ordered analytic functions in a CHECK constraint.

The search condition for a CHECK constraint cannot specify SET operators.

You cannot specify CHECK constraints for identity columns or for the columns of a volatile table.

When you add or modify a CHECK constraint, Vantage scans all existing rows of the table to validate that the current values conform to the specified search condition. Otherwise, the system returns an error to the requestor and does not change the table definition.

CHECK constraints are valid for nontemporal and temporal tables. For details about how temporal tables support CHECK constraints, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

This clause is a Teradata extension to the ANSI SQL:2011 standard.

A combination of table-level, column-level, and FOREIGN KEY WITH CHECK OPTION constraints on a table used for a view can create a constraint expression that is too large to be parsed for INSERT and UPDATE requests.

CHECK constraints for character columns use the current session collation. A CHECK constraint may be satisfied for one session collation, but not met for another, even though the same data is inserted or updated.

You cannot specify CHECK constraints on columns that are defined with any of the following data types.

- BLOB
- CLOB
- ARRAY/VARRAY
- Period
- UDT
- XML
- Geospatial
- JSON
- DATASET

You cannot add CHECK constraints that reference BLOB or CLOB columns.

You cannot specify multiple unnamed CHECK constraints with identical boolean conditions. However, the following ALTER TABLE request is valid because the case of *f1* and *F1* is different:

```
ALTER TABLE t1 (
  ADD f1 INTEGER, CHECK (f1 > 0), CHECK (F1 > 0));
```

The following request is not valid because the case of both *f1* specifications is identical.

```
ALTER TABLE t1 (
  ADD f1 INTEGER, CHECK (f1 > 0), CHECK (f1 > 0));
```

You can also use the BETWEEN ... AND operator as a form of CHECK constraint except for volatile table columns, identity columns, UDT columns, ARRAY, VARRAY, Geospatial columns, Period columns, BLOB columns, or CLOB columns.

The condition for a CHECK constraint can be any simple boolean search condition. Subqueries, aggregate expressions, and ordered analytic expressions are not valid in search conditions for CHECK constraint definitions.

You cannot specify CHECK constraints for:

- volatile table columns
- identity columns

You can use the following non-ANSI SQL constraint syntax for table attribute CHECK constraints.

```
BETWEEN value_1 AND value_2
```

The system treats the constraint as if it were the following ANSI-compliant constraint.

```
CHECK (column_name
  BETWEEN value_1
  AND value_2)
```

For information about using CHECK constraints to construct domains, see *Teradata Vantage™ - Database Design*, B035-1094.

CONSTRAINT *constraint_name*

Named boolean conditional expression to restrict the values that can be inserted into, or updated for, a column.

REFERENCES

A reference as a simple column attribute.

The following rules apply to column attribute REFERENCES constraints only.

- FOREIGN KEY is only used for table constraint foreign key specifications.
- If you do not specify *column_name*, the referenced table must have a simple primary key, and the specified foreign key column references that primary key column in the referenced table.

You cannot specify a REFERENCES constraint on a column with the JSON data type.

For an unnamed REFERENCES column constraint, use this syntax:

```
REFERENCES referenced_table_name
      (referenced_column_name)
```

referenced_table_name

The name of the table that contains the primary key or alternate key referenced by the specified *referencing_column*.

referenced_column_name

The name of the table that contains the primary key or alternate key referenced by *referencing_column*.

CONSTRAINT *constraint_name*

The name of the column attribute foreign key constraint.

For a named REFERENCES column constraint, use this syntax:

```
CONSTRAINT constraint_name REFERENCES referenced_table_name
      (referenced_column_name)
```

WITH CHECK OPTION

A referential integrity constraint. The integrity of the relationship is checked only when the entire transaction of which it is a component completes.

For a column attribute foreign key constraint, specify this clause:

```
REFERENCES WITH CHECK OPTION referenced_table_name
      (referenced_column_name)
```

If any of the rows in the transaction violate the Referential Integrity rule, the entire transaction is rolled back.

See *Teradata Vantage™ - Database Design*, B035-1094 for information about the Referential Integrity rule.

This option is not valid for temporal tables. For details, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

This clause is a Teradata extension to the ANSI SQL:2011 standard.

WITH NO CHECK OPTION

The constraint to be added or dropped is not to be used to validate referential integrity for the specified primary key-foreign key relationship.

For a column attribute foreign key constraint, specify this clause as follows.

```
REFERENCES WITH NO CHECK OPTION referenced_table_name
                                (referenced_column_name)
```

This is the only valid referential integrity option for temporal tables. For details, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

See *Teradata Vantage™ - Temporal Table Support*, B035-1182 for information about how to use this clause with temporal tables.

This clause is a Teradata extension to the ANSI SQL:2011 standard.

Alter Partitioning

The following section describes the partitioning options.

PARTITION BY

Add or change partitioning for a table. The table must be empty.

To change an nonpartitioned table to be partitioned or to change the partitioning for an existing partitioned table, specify **PARTITION BY** and then specify a valid partitioning. The partitioning is one or more partitioning expressions, a **COLUMN** specification, or a combination of both.

For 2-byte partitioning, you can specify a maximum of 15 partitioning levels.

For 8-byte partitioning, you can specify a maximum of 62 partitioning levels.

You cannot specify a row-level security constraint column as a partitioning column.

partitioning_expression

The result of a partitioning expression not based on a **RANGE_N** or **CASE_N** function must always be eligible to be implicitly cast to **INTEGER** type, if it is not already **INTEGER** value. A partitioning expression not based on a **RANGE_N** or **CASE_N** is only allowed if there is a single level of partitioning.

Use a **CASE_N** function to define a mapping between conditions to **INTEGER** numbers. The maximum number of partitions for a **CASE_N** partitioning level is limited by the maximum partitioning constraint text size, the request text size limit, and other factors.

You cannot alter a table to have a partitioning level that includes a row-level security constraint column.

A partitioning expression cannot specify external or SQL UDFs or columns with the following data types:

- BLOB
- CLOB
- ARRAY/VARRAY
- UDT
- Period
- XML
- Geospatial
- JSON
- DATASET

However, you can reference Period columns indirectly using the BEGIN and END bound functions. See [Example: CASE_N Partitioning Expression Using the END Bound Function](#) and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

COLUMN

A partitioning can only include a column partitioning level if the table does not have a primary index or ALTER TABLE specifies NO PRIMARY INDEX.

PARTITION BY COLUMN

Column partitioning.

You can define a large variety of partition levels with a large range in the number of combined partitions. However, complex partitioning can result in greater impact on performance and storage.

If you do not specify COLUMN or ROW for a column partition, the format is system-determined. For information about COLUMN format and ROW format, see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

AUTO COMPRESS

Vantage automatically determines and applies the best available compression method, if it can reduce the size of physical rows. AUTO COMPRESS is the default for column partitions.

(COLUMN (*column_name*, *column_name*))

Indicates COLUMN format for the column partition. Column grouping provides flexibility in specifying which columns are grouped into which partitions while still being able to specify the display order in the table element list. Column grouping in the column list for a

table allows for a simpler, but less flexible, specification of column groupings than you can specify in a partitioning level.

ROW (*column_name*, *column_name*)

Indicates ROW format for the column partition. ROW specifies that the column partition has ROW format. A ROW format means that only one column-partition value is stored in a physical row as a subrow.

WITH DELETE

Delete any row for which a new partitioning expression evaluates to a value outside the valid range of 1 through the number of partitions defined for that level.

Even though you can specify *PARTITION BY partitioning_level WITH DELETE*, the table must be empty so it has no effect. This option can be useful under certain circumstances if the table is not empty and *DROP RANGE* is specified.

WITH INSERT

Insert into *save_table* any row for which a new partitioning expression evaluates to a value outside the valid range of 1 through the number of partitions defined for that level. After it inserts the non-valid row, Vantage deletes it from the table. Even though you can specify with *PARTITION BY partitioning_level WITH INSERT*, the table must be empty so it has no effect. This option can be useful under certain circumstances if the table is not empty and *DROP RANGE* is specified. You can use this option with row-level security-protected tables as long as the tables referenced in the request are all row-level security-protected and they are all be defined with the same row-level security constraints.

If you do not specify the constraint values to be inserted into the target table, Vantage takes the constraint values for the target table from the source table.

INTO

Optional keyword preceding *save_table*.

save_table

save_table and the table being altered must be different tables, and *save_table* must have the same number of columns (with matching data types) as the table being modified.

NOT PARTITIONED

The table is not partitioned.

To change a partitioned table to a nonpartitioned table, specify **NOT PARTITIONED**.

DROP RANGE

Drop a set of ranges from the RANGE_N function on which a partitioning level for the table is based. You must specify the ranges to be dropped in ascending order. Use a conditional expression to drop a range set from the RANGE_N function on which a partitioning level for the table is based. You cannot DROP a condition from a CASE_N function in a partitioning. You must use PARTITION BY to redefine the entire partitioning for the table.

For information about the RANGE_N function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

DROP RANGE#L *n*

Represents a partition level number where *n* is an integer between 1 and 15, inclusive, for 2-byte partitioning and between 1 and 62, inclusive for 8-byte partitioning. You can only drop ranges from a character partition if the partitioning level for the table is derived exclusively from a RANGE_N function. You can only drop standard ranges from non-character partitioning levels. The term standard range refers to any range other than NO RANGE, UNKNOWN, or NO RANGE OR UNKNOWN. For example, a standard range could be a multilevel character partitioning with one or more non-character partitioning levels. You can only drop ranges in a character partitioning if the session collation matches the table collation and the session mode is the same as the session mode in effect when the table was created. You can only alter character partitioning levels that are defined using a RANGE_N function. You can only alter such character partitioning levels to add or drop a NO RANGE, UNKNOWN, or NO RANGE OR UNKNOWN partition. You can only drop standard ranges from non-character partitioning levels. For example, a multilevel character partitioning with one or more non-character partitioning levels. You can only drop ranges from a character partitioning if the session collation matches the table collation and the session mode is the same as the session mode in effect when the table was created. You can only alter character partitioning levels that are defined using RANGE_N functions, and you can only alter such partitions to add or drop a NO RANGE, UNKNOWN, or NO RANGE OR UNKNOWN partition.

BETWEEN *start_expression*

The *start_expression* must be defined according to the rules for the RANGE_N function. #L *n* represents a partition level number where *n* is an integer between 1 and 15, inclusive, for 2-byte partitioning and between 1 and 62, inclusive for 8-byte partitioning.

AND *end_expression*

Required only for the last range. The *end_expression* must be defined according to the rules for the RANGE_N function.

EACH *range_size*

A range with an EACH clause is equivalent to a series of *start_expressions* by adding multiples from 0 in increments of 1 of *range_size* to the specified *start_expression*.

A *range_size* must be less than or equal to *end_expression* or less than the next *start_expression*.

Each *range_size* must be a constant expression.

**NO RANGE
UNKNOWN**

You can drop NO RANGE, UNKNOWN, and NO RANGE OR UNKNOWN specifications from the definition for a RANGE_N function.

WHERE *conditional_expression*

You can specify a range expression or a conditional partition expression. For examples of dropping ranges based on a range expression or a conditional partition expression, see [Example: Dropping and Adding Partition Ranges](#). You must base *conditional_expression* on the system-derived PARTITION column or PARTITION#L *n* column, where *n* ranges from 1 through 62, and depends on the level being modified and whether one or multiple levels are changed.

ADD RANGE

Add a set of ranges to the RANGE_N function on which a partitioning level for the table is based. You must specify the ranges to be added in ascending order. For information about the RANGE_N function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Note:

You cannot ADD a range set to a CASE_N function in a partitioning level, but must use the MODIFY or MODIFY PRIMARY INDEX options to redefine the entire partitioning for the table.

You cannot change from 2-byte partitioning to 8-byte partitioning with ADD RANGE. You must use the MODIFY PARTITION BY or MODIFY PRIMARY INDEX options to redefine the entire partitioning for the table.

ADD RANGE#L *n*

Partition level number where *n* is an integer value from 1 through 15, for 2-byte partitioning, or from 1 through 62, for 8-byte partitioning. Note that there is no space between RANGE and #L *n* in a RANGE#L *n* specification. You can only add ranges if a partitioning level for the table is derived exclusively from RANGE_N functions. Use a RANGE_N function to define a mapping of ranges of the following types to INTEGER or BIGINT numbers:

- BIGINT
- BYTEINT
- CHARACTER
- DATE
- GRAPHIC
- INTEGER
- SMALLINT
- VARCHAR
- VARGRAPHIC

You can only add standard ranges from non-character partitioning levels. The term standard range here refers to any range other than NO RANGE, UNKNOWN, or NO RANGE OR UNKNOWN. For example, a multilevel character partitioning with one or more non-character partitioning levels.

You can only add ranges to a character partitioning if the session collation matches the table collation and the session mode matches the session mode in effect when the table was created.

You can only alter character partitioning levels that are defined using a RANGE_N function, and you can only alter such partitions to add or drop a NO RANGE, UNKNOWN, or NO RANGE OR UNKNOWN partition.

The maximum number of ranges, not including the NO RANGE, NO RANGE OR UNKNOWN, and UNKNOWN partitions for a RANGE_N partitioning level, is 9,223,372,036,854,775,805.

BETWEEN *start_expression*

The *start_expression* is defined according to the rules for the RANGE_N function..

AND *end_expression*

Required only for the last range.

The *end_expression* is defined according to the rules for the RANGE_N function. .

EACH *range_size*

Equivalent to a series of *start_expressions* by adding multiples from 0 in increments of 1 of *range_size* to the specified *start_expression*. A *range_size* must be less than or equal to *end_expression* or less than the next *start_expression*. The *range_size* variable must be a constant expression.

You cannot specify an EACH clause if the RANGE_N function specifies a character or graphic test value.

NO RANGE UNKNOWN

You can add NO RANGE or UNKNOWN to the definition for a RANGE_N function.

WHERE *conditional_expression*

You can specify a range expression or a conditional partition expression. For examples of adding ranges based on a range expression or a conditional partition expression, see [Example: Dropping and Adding Partition Ranges](#). You must base *conditional_expression* on the system-derived PARTITION column or PARTITION#L *n* column, where *n* ranges from 1 through 62, and depends on the level being modified and whether one or multiple levels are changed.

WITH DELETE

Delete any row for which a new partitioning expression evaluates to a value outside the valid range of 1 through the number of partitions defined for that level.

WITH INSERT

Insert into *save_table* any row for which a new partitioning level evaluates to a value outside the valid range of 1 through the number of partitions defined for that level. After it inserts the non-valid row, Vantage deletes it from the table.

You can use this option with row-level security-protected tables as long as the tables referenced in the request are all row-level security-protected and they are all be defined with the same row-level security constraints.

INTO

Optional keyword.

save_table

save_table and the table being altered must be different tables, and *save_table* must have the same number of columns (with matching data types) as the table being modified.

Examples

ALTER TABLE NORMALIZE Examples

Example: Adding a NORMALIZE Constraint to a Table

These examples show how to use an ALTER TABLE request to add a NORMALIZE constraint to a table. The system default condition for a NORMALIZE constraint is:

```
ON MEETS OR OVERLAPS
```

The examples all use the following table definition or an equivalent temporal table definition. The table definition does not specify a NORMALIZE constraint.

```
CREATE TABLE project (
  emp_id      INTEGER,
  project_name VARCHAR(20),
  dept_id     INTEGER,
  duration    PERIOD(DATE));
```

The following ALTER TABLE request modifies the *project* table to add a NORMALIZE constraint on the *duration* column. Rows are normalized only when the values of the *emp_id*, *project_name*, and *dept_id* columns are the same and the Period values for the *duration* column either meet or overlap.

```
ALTER TABLE project
ADD NORMALIZE ON duration;
```

The following ALTER TABLE request modifies the *project* table to add a NORMALIZE constraint on the *duration* column. The *dept_id* column is the only column to ignore for normalization.

Rows are normalized only when the values of columns *emp_id* and *project_name* are the same and the Period values for the *duration* column meet or overlap.

```
ALTER TABLE project
ADD NORMALIZE ALL BUT(dept_id) ON duration;
```

The following ALTER TABLE request modifies the *project* table to add a NORMALIZE constraint on the *duration* column that applies only when the Period values for the *duration* column overlap. The *dept_id* column is the only column to ignore for normalization.

Rows are normalized only when the values of columns *emp_id* and *project_name* are the same and the Period values for the *duration* column overlap.

```
ALTER TABLE project
ADD NORMALIZE ALL BUT(dept_id) ON duration ON OVERLAPS;
```

The following ALTER TABLE request modifies the *project* table to add a NORMALIZE constraint on the *duration* column that applies only when the Period values for the *duration* column overlap. The *dept_id* column is the only column to ignore for normalization.

Rows are normalized only when the values of columns *emp_id* and *project_name* are the same and the Period values for the *duration* column overlap.

```
ALTER TABLE project
ADD NORMALIZE ALL BUT(dept_id) ON duration ON OVERLAPS;
```

The next example defines the *project* table using a NORMALIZE condition of ON OVERLAPS on the *duration* column.

```
CREATE TABLE project (
  emp_id INTEGER,
  project_name VARCHAR(20),
  dept_id INTEGER,
  duration PERIOD(DATE),
  NORMALIZE ALL BUT (dept_id) ON duration ON OVERLAPS);
```

Modify the definition of *project* to add the *project_name* column to the ALL BUT *normalize_ignore* column list and to add the condition OR MEETS to the existing ON OVERLAPS condition.

```
ALTER TABLE
ADD NORMALIZE ALL BUT(dept_id,project_name) ON duration ON OVERLAPS
OR MEETS;
```

Vantage renormalizes the *project* table after you submit this ALTER TABLE request.

Example: Dropping a NORMALIZE Constraint From a Table

This example uses a nontemporal version of the *project* table from [Example: Adding a NORMALIZE Constraint to a Table](#) except that this table definition specifies a NORMALIZE ALL BUT constraint with the *dept_id* column specified in the *normalize_ignore* column list and the NORMALIZE constraint is defined on the *duration* column.

```
CREATE TABLE project (
  emp_id          INTEGER,
  project_name    VARCHAR(20),
  dept_id         INTEGER,
```

```
duration          PERIOD(DATE),
NORMALIZE ALL BUT(dept_id) ON duration);
```

The following ALTER TABLE request modifies the *project* table to remove the NORMALIZE constraint on the *duration* column.

```
ALTER TABLE project
DROP NORMALIZE;
```

ALTER TABLE MODIFY PRIMARY Examples

Example: Modify a Primary Index Table to a Primary Index-Column Partitioned Table

This example modifies the following table definition:

```
CREATE TABLE pi3 (a INTEGER, b INTEGER, c CHAR(10))
PRIMARY INDEX (a);
```

Execute any one of the following ALTER TABLE statements on the empty table:

```
ALTER TABLE pi3 MODIFY PARTITION BY COLUMN;
```

```
ALTER TABLE pi3 MODIFY PRIMARY INDEX PARTITION BY COLUMN;
```

```
ALTER TABLE pi3 MODIFY PRIMARY INDEX (a) PARTITION BY COLUMN;
```

The following table definition results:

```
CREATE TABLE pi3 (a INTEGER, b INTEGER, c CHAR(10))
PRIMARY INDEX (a) PARTITION BY COLUMN;
```

Example: Modify a Primary Index Table to a Primary Index-Column Partitioned-RP Table

This example modifies the following table definition:

```
CREATE TABLE pi4 (a INTEGER, b INTEGER, c CHAR(10))
PRIMARY INDEX (a);
```

Execute any one of the following ALTER TABLE statements on the empty table:

```
ALTER TABLE pi4 MODIFY
PARTITION BY (COLUMN, RANGE_N(b BETWEEN 1 AND 10 EACH 1));
```

```
ALTER TABLE pi4 MODIFY PRIMARY INDEX
PARTITION BY (COLUMN, RANGE_N(b BETWEEN 1 AND 10 EACH 1));
```

```
ALTER TABLE pi4 MODIFY PRIMARY INDEX (a)
PARTITION BY (COLUMN, RANGE_N(b BETWEEN 1 AND 10 EACH 1));
```

The following table definition results:

```
CREATE TABLE pi4 (a INTEGER, b INTEGER, c CHAR(10))
PRIMARY INDEX (a)
PARTITION BY (COLUMN, RANGE_N(b BETWEEN 1 AND 10 EACH 1));
```

Example: Modify a Primary Index-Column Partitioned Table to a Primary AMP Index-Column Partitioned Table

This example uses the following table definition:

```
CREATE TABLE p8 (a INTEGER, b INTEGER, c CHAR(10))
PRIMARY INDEX (a) PARTITION BY COLUMN;
```

Execute any one of the following ALTER TABLE statements on the empty table:

```
ALTER TABLE p8 MODIFY PRIMARY AMP INDEX;
```

```
ALTER TABLE p8 MODIFY PRIMARY AMP INDEX (a);
```

```
ALTER TABLE p8 MODIFY PRIMARY AMP INDEX (a) PARTITION BY COLUMN;
```

The following table definition results:

```
CREATE TABLE p8 (a INTEGER, b INTEGER, c CHAR(10))
PRIMARY AMP INDEX (b) PARTITION BY COLUMN;
```

Example: Modify a Primary Index-Column Partitioned Table to a NoPI-Column Partitioned Table

This example uses the following table definition:

```
CREATE TABLE p9 (a INTEGER, b INTEGER, c CHAR(10))
PRIMARY INDEX (a) PARTITION BY COLUMN;
```

Execute one of the following ALTER TABLE statements on the empty table:

```
ALTER TABLE p9 MODIFY NO PRIMARY INDEX;
```

```
ALTER TABLE p9 MODIFY NO PRIMARY INDEX PARTITION BY COLUMN;
```

The following table definition results:

```
CREATE TABLE p9 (a INTEGER, b INTEGER, c CHAR(10))
  NO PRIMARY INDEX PARTITION BY COLUMN;
```

Example: Modify a Primary Index Table to a Primary AMP Index-Column Partitioned Table

This example uses the following table definition:

```
CREATE TABLE p11 (a INTEGER, b INTEGER, c CHAR(10))
  PRIMARY INDEX (a);
```

Execute one of the following ALTER TABLE statements on the empty table:

```
ALTER TABLE p11 MODIFY PRIMARY AMP INDEX PARTITION BY COLUMN;
```

```
ALTER TABLE p11 MODIFY PRIMARY AMP INDEX (a) PARTITION BY COLUMN;
```

The following table definition results:

```
CREATE TABLE p11 (a INTEGER, b INTEGER, c CHAR(10))
  PRIMARY AMP INDEX (a) PARTITION BY COLUMN;
```

Example: Modify a Primary Index Table to a Primary AMP Index-Column Partitioned Table with a Different Index Column

This example uses the following table definition:

```
CREATE TABLE p12 (a INTEGER, b INTEGER, c CHAR(10))
  PRIMARY INDEX (a);
```

Execute the following ALTER TABLE statement on the empty table:

```
ALTER TABLE p12 MODIFY PRIMARY AMP INDEX (b) PARTITION BY COLUMN;
```

The following table definition results:

```
CREATE TABLE p12 (a INTEGER, b INTEGER, c CHAR(10))
  PRIMARY AMP INDEX (b) PARTITION BY COLUMN;
```

Example: Modify a Primary AMP Index Column Partitioned Table to a Primary Index Nonpartitioned Table

This example uses the following table definition:

```
CREATE TABLE pt3 (a INTEGER, b INTEGER, c CHAR(10))
  PRIMARY AMP INDEX (a) PARTITION BY COLUMN;
```

Execute one of the following ALTER TABLE statements on the empty table:

```
ALTER TABLE pt3 MODIFY PRIMARY INDEX (a) NOT PARTITIONED;
```

```
ALTER TABLE pt3 MODIFY PRIMARY INDEX NOT PARTITIONED;
```

The following table definition results:

```
CREATE TABLE pt3 (a INTEGER, b INTEGER, c CHAR(10))
  PRIMARY INDEX (a);
```

Example: Modify a Primary AMP Index Column Partitioned Table to a Primary Index Column Partitioned Table

This example uses the following table definition:

```
CREATE TABLE pt4 (a INTEGER, b INTEGER, c CHAR(10))
  PRIMARY AMP INDEX (a) PARTITION BY COLUMN;
```

Execute one of the following ALTER TABLE statements on the empty table:

```
ALTER TABLE pt4 MODIFY PRIMARY INDEX (b);
```

```
ALTER TABLE pt4 MODIFY PRIMARY INDEX (b) PARTITION BY COLUMN;
```

The following table definition results:

```
CREATE TABLE pt4 (a INTEGER, b INTEGER, c CHAR(10))
  PRIMARY INDEX (b) PARTITION BY COLUMN;
```

ALTER TABLE MODIFY NO PRIMARY Examples

Example: Modifying a Table to Not Have a Primary Index

The following example deletes all of the rows in the *orders* table, alters the table to not have a primary index, adds both column partitioning and row partitioning, and adds a secondary index.

```
DELETE orders ALL;
ALTER TABLE orders
  MODIFY NO PRIMARY INDEX
  PARTITION BY (COLUMN ADD 5,
                RANGE_N(o_ordertsz
                        BETWEEN TIMESTAMP '2003-01-01 00:00:00.000000+00:00'
                        AND      TIMESTAMP '2009-12-31 23:59:59.999999+00:00')
```



```

        EACH INTERVAL '1' MONTH) );
CREATE UNIQUE INDEX(o_orderkey) on orders;

```

This produces the following table definition for *orders*.

```

CREATE TABLE orders (
  o_orderkey    INTEGER NOT NULL,
  o_custkey     INTEGER,
  o_orderstatus CHARACTER(1) CASESPECIFIC,
  o_totalprice  DECIMAL(13,2) NOT NULL,
  o_ordertsz    TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_comment     VARCHAR(79) )
PARTITION BY (COLUMN ADD 5,
              RANGE_N(o_ordertsz
                    BETWEEN TIMESTAMP '2003-01-01 00:00:00.000000+00:00'
                    AND   TIMESTAMP '2009-12-31 23:59:59.999999+00:00'
                    EACH INTERVAL '1' MONTH) ),
  UNIQUE INDEX(o_orderkey);

```

Example: Modifying a Primary AMP Index Table to Not Have a Primary Index (NoPI)

This example uses the following table definition:

```

CREATE TABLE Orders
( o_orderkey INTEGER NOT NULL,
  o_custkey  INTEGER,
  o_orderstatus CHAR(1) CASESPECIFIC,
  o_totalprice DECIMAL(13,2) NOT NULL,
  o_ordertsz  TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_comment   VARCHAR(79) )
PRIMARY AMP INDEX (o_orderkey) PARTITION BY COLUMN
UNIQUE INDEX (o_orderkey);

```

Execute the following ALTER TABLE statement:

```

ALTER TABLE Orders MODIFY NO PRIMARY INDEX;

```

The following table results:

```

CREATE TABLE Orders
( o_orderkey INTEGER NOT NULL,
  o_custkey  INTEGER,
  o_orderstatus CHAR(1) CASESPECIFIC,
  o_totalprice DECIMAL(13,2) NOT NULL,
  o_ordertsz  TIMESTAMP(6) WITH TIME ZONE NOT NULL,

```

```
o_comment VARCHAR(79) )
NO PRIMARY INDEX PARTITION BY COLUMN
UNIQUE INDEX (o_orderkey);
```

ALTER TABLE MODIFY partitioning Examples

Example: Removing the Row Partitioning from a Column-Partitioned Table

This example deletes the rows in *orders* and then alters the table to remove the row partitioning.

```
DELETE orders ALL;
ALTER TABLE orders
MODIFY PARTITION BY COLUMN ALL BUT ((o_orderstatus,
                                     o_ordersubstatus),
                                     ROW(o_ship_addr, o_bill_addr)
                                     NO AUTO COMPRESS) ADD 3;
```

This results in the following table definition. Note the increase in the ADD option value, where all excess partitions are now assigned to level 1. Because of this modification, you could theoretically add 65,524 column partitions to the table; however, the limit on the maximum number of columns for a table would be reached first.

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_ordertsz      TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_salesperson   VARCHAR(5),
  o_ordersubstatus CHARACTER(1) CASESPECIFIC,
  o_ship_addr     VARCHAR(500),
  o_bill_addr     VARCHAR(200),
  o_item_count    INTEGER )
PARTITION BY COLUMN ALL BUT ((o_orderstatus, o_ordersubstatus)
                              ROW(o_ship_addr, o_bill_addr) NO AUTO COMPRESS)
ADD 65524;
```

Example: Adding and Dropping Columns in a Column-Partitioned Table

Suppose you create the following column-partitioned table *at3*.

```
CREATE TABLE at3 (
  a INTEGER,
```

```

    b INTEGER,
    c INTEGER,
    d INTEGER,
    e INTEGER,
    f INTEGER)
PARTITION BY COLUMN ((a, b), f, (d, e), c);

```

After you create table *at3* you decide to drop column *c* and replace it with a new column *g*.

Any of the following four equivalent ALTER TABLE statements can make this modification to *at3*.

```

ALTER TABLE at3 DROP c, ADD g INTEGER INTO c;
ALTER TABLE at3 ADD g INTEGER INTO c, DROP c;
ALTER TABLE at3 DROP c, ADD g INTEGER;
ALTER TABLE at3 ADD g INTEGER, DROP c;

```

Notice the use of the INTO clause in the first 2 examples above. When you specify an INTO clause for a column or a group of columns being added, the column set being added is added to the column partition that contains the column specified by the column name in the INTO clause. Both of these examples add the new column *g* into the partition that contains column *c*.

The preceding ALTER TABLE requests produce the following new definition for *at3*.

```

CREATE TABLE at3 (
    a INTEGER,
    b INTEGER,
    d INTEGER,
    e INTEGER,
    f INTEGER,
    g INTEGER)
PARTITION BY COLUMN ((a, b), (d, e), f, g);

```

After altering table *at3*, you decide to drop the existing columns *d* and *e* and replace them with new columns *i* and *h*, which are both to be contained within the same column partition.

Any of the following six equivalent ALTER TABLE requests can make this modification to *at3*.

```

ALTER TABLE at3 DROP d, DROP e, ADD h INTEGER INTO d, ADD i INTEGER
INTO e;
ALTER TABLE at3 ADD h INTEGER INTO e, DROP d, DROP e, ADD i INTEGER INTO d;
ALTER TABLE at3 ADD h INTEGER INTO d, DROP d, ADD i INTEGER INTO e, DROP e;
ALTER TABLE at3 DROP d, DROP e, ADD (h INTEGER, i INTEGER) INTO e;
ALTER TABLE at3 DROP d, DROP e, ADD (h INTEGER, i INTEGER);
ALTER TABLE at3 ADD (h INTEGER, i INTEGER), DROP d, DROP e;

```

Notice the use of the INTO clause in the first four examples above.

The preceding ALTER TABLE statements produce the following new definition for *at3*. Columns *d* and *e* no longer exist in this definition and columns *h* and *i* have been added.

```
CREATE TABLE at3 (
  a INTEGER,
  b INTEGER,
  f INTEGER,
  g INTEGER,
  h INTEGER,
  i INTEGER)
PARTITION BY COLUMN ((a, b), f, g, (h, i));
```

Suppose you want to alter table *at3* to add new columns *c* and *k* to the table. Either the first two ALTER TABLE statements below, submitted in sequence, or the third ALTER TABLE statement, can make this modification to *at3*.

```
ALTER TABLE at3 ADD c INTEGER;
```

and

```
ALTER TABLE at3 ADD k INTEGER INTO c;
```

or

```
ALTER TABLE at3 ADD (c INTEGER, k INTEGER);
```

The preceding ALTER TABLE requests produce the following new definition for *at3*.

```
CREATE TABLE at3 (
  a INTEGER,
  b INTEGER,
  c INTEGER,
  f INTEGER,
  g INTEGER,
  h INTEGER,
  i INTEGER,
  k INTEGER)
PARTITION BY COLUMN ((a, b), f, g, (h, i), (c, k));
```

Example: Modify a Primary Index-Column Partitioned Table to a Nonpartitioned or Primary Index-Nonpartitioned Table

This example uses the following table definition:

```
CREATE TABLE pi6 (a INTEGER, b INTEGER, c CHAR(10))
  PRIMARY INDEX (a) PARTITION BY COLUMN;
```

Execute any one of the following ALTER TABLE statements on the empty table:

```
ALTER TABLE pi6 MODIFY NOT PARTITIONED;
```

```
ALTER TABLE pi6 MODIFY PRIMARY INDEX NOT PARTITIONED;
```

```
ALTER TABLE pi6 MODIFY PRIMARY INDEX(a) NOT PARTITIONED;
```

The following table definition results:

```
CREATE TABLE pi6 (a INTEGER, b INTEGER, c CHAR(10)) PRIMARY INDEX (a);
```

Example: Modify a Primary Index-Column Partitioned Table to a No Primary Index-Nonpartitioned Table

This example uses the following table definition:

```
CREATE TABLE p7 (a INTEGER, b INTEGER, c CHAR(10))
  PRIMARY INDEX (a) PARTITION BY COLUMN;
```

Execute the following ALTER TABLE statement on the empty table:

```
ALTER TABLE p7 MODIFY NO PRIMARY INDEX NOT PARTITIONED;
```

The following table definition results:

```
CREATE TABLE p7 (a INTEGER, b INTEGER, c CHAR(10)) NO PRIMARY INDEX;
```

Example: Migrating Data to a New Time Zone

This example migrates data from a pre-13.10 system to a new system with the time zone set to the system default and default TIMEDATEWZCONTROL = 2.

Old system: 16/17 = 0, 57=0

Following is the table definition for this example:

```
CREATE SET TABLE t1 ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
```

```
CHECKSUM = DEFAULT,
DEFAULT MERGEBLOCKRATIO
(
  i INTEGER,
  j TIMESTAMP(6))
PRIMARY INDEX ( i );
```

Two rows of data are inserted.

```
SELECT * FROM t1;
```

The results display as follows.

i	j
1	2011-11-20 06:28:59.000000
2	9999-12-31 15:59:59.999999

The new system is set to this time zone: 18 = America Pacific, 57=2

Alter the table to the new time zone.

```
ALTER TABLE t1 FROM TIME ZONE='00:00',TIMEDATEWZCONTROL=0;
```

Set the time zone to Coordinated Universal Time (UTC) or Greenwich Mean Time.

```
SET TIME ZONE 'GMT';
```

Display the rows.

```
SELECT * FROM t1;
```

The converted date and timestamp information displays as follows.

i	j
1	2011-11-20 14:28:59.000000
2	9999-12-31 23:59:59.000000

Set the time zone to the user default for the session.

```
SET TIME ZONE USER;
```

Display the rows.

```
SELECT * FROM t1;
```

The converted date and timestamp information displays as follows.

i	j
1	2011-11-20 06:28:59.000000
2	9999-12-31 15:59:59.000000

Example: Using SET DOWN

The following request restricts or suspends access to *emp_table*. With the exception of fast path DELETE ALL requests, DML requests cannot be executed on this table until an ALTER TABLE ... RESET DOWN request makes *emp_table* accessible again.

```
ALTER TABLE emp_table SET DOWN;
```

Example: Using RESET DOWN

The following request re-enables access to *emp_table*, which had been set down either because of an excessive number of down regions in one of its data subtables on one of the AMPs or because someone had submitted a SET DOWN request against it.

```
ALTER TABLE emp_table RESET DOWN;
```

The following request returns an error because it attempts to alter *emp_table* to have fallback, and you cannot specify SET DOWN or RESET DOWN after specifying a table option.

```
ALTER TABLE emp_tablec, Fallback RESET DOWN;
```

ALTER TABLE Join Index Examples

Example: Altering a Column Partition to ROW Format for a Join Index with a Primary AMP Index

Assume this table definition:

```
CREATE TABLE Orders
( o_orderkey INTEGER NOT NULL,
```

```

    o_custkey INTEGER,
    o_orderstatus CHAR(1) CASESPECIFIC,
    o_totalprice DECIMAL(13,2) NOT NULL,
    o_ordertsz TIMESTAMP(6) WITH TIME ZONE NOT NULL,
    o_comment VARCHAR(79),
    o_salesperson VARCHAR(5),
    o_ordersubstatus CHAR(1) CASESPECIFIC,
    o_comment_ext1 VARCHAR(79),
    o_comment_ext2 VARCHAR(79),
    o_ship_addr VARCHAR(500),
    o_bill_addr VARCHAR(200),
    o_alt_ship_addr VARCHAR(500),
    o_alt_bill_addr VARCHAR(200),
    o_item_count INTEGER )
PRIMARY AMP INDEX (o_orderkey)
PARTITION BY (
    COLUMN AUTO COMPRESS ALL BUT (
        o_ordersubstatus NO AUTO COMPRESS,
        (o_comment_ext1, o_comment_ext2) NO AUTO COMPRESS,
        ROW(o_ship_addr, o_bill_addr) NO AUTO COMPRESS,
        COLUMN(o_alt_ship_addr, o_alt_bill_addr) ),
    UNIQUE INDEX (o_orderkey);

```

Create a join index as follows:

```

CREATE JOIN INDEX jOrders AS
    SELECT ROWID AS rw, o_custkey, o_salesperson, o_bill_addr
    FROM Orders
PRIMARY AMP INDEX (o_custkey)
PARTITION BY COLUMN NO AUTO COMPRESS
    ALL BUT (o_salesperson AUTO COMPRESS);

```

Alter the column partition to have ROW format:

```

ALTER TABLE jOrders ADD ROW(o_bill_addr);

```

This results in a join index definition as follows:

```

CREATE JOIN INDEX jOrders AS
    SELECT ROWID AS rw, o_custkey, o_salesperson, o_bill_addr
    FROM Orders
PRIMARY AMP INDEX (o_custkey)

```



```
PARTITION BY COLUMN NO AUTO COMPRESS
ALL BUT (o_salesperson AUTO COMPRESS, ROW(o_bill_addr));
```

Altering a Column Partition to AUTO COMPRESS for a Join Index with a Primary AMP Index

Assume the following table definition:

```
CREATE TABLE Orders
( o_orderkey INTEGER NOT NULL,
  o_custkey INTEGER,
  o_orderstatus CHAR(1) CASESPECIFIC,
  o_totalprice DECIMAL(13,2) NOT NULL,
  o_ordertsz TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_comment VARCHAR(79),
  o_salesperson VARCHAR(5),
  o_ordersubstatus CHAR(1) CASESPECIFIC,
  o_comment_ext1 VARCHAR(79),
  o_comment_ext2 VARCHAR(79),
  o_ship_addr VARCHAR(500),
  o_bill_addr VARCHAR(200),
  o_alt_ship_addr VARCHAR(500),
  o_alt_bill_addr VARCHAR(200),
  o_item_count INTEGER )
PRIMARY AMP INDEX (o_orderkey)
PARTITION BY (
  COLUMN AUTO COMPRESS ALL BUT (
    o_ordersubstatus NO AUTO COMPRESS,
    (o_comment_ext1, o_comment_ext2) NO AUTO COMPRESS,
    ROW(o_ship_addr, o_bill_addr) NO AUTO COMPRESS,
    COLUMN(o_alt_ship_addr, o_alt_bill_addr) ),
  UNIQUE INDEX (o_orderkey);
```

You create a join index with a primary AMP index:

```
CREATE JOIN INDEX jOrders AS
  SELECT ROWID AS rw, o_custkey, o_salesperson, o_bill_addr
  FROM Orders
PRIMARY AMP INDEX (o_custkey)
PARTITION BY COLUMN NO AUTO COMPRESS;
```

The following ALTER TABLE statement adds autocompression for the column partition:

```
ALTER TABLE jOrders ADD (o_salesperson) AUTO COMPRESS;
```

Example: Modifying a Single-Column Join Index Partition to Have Autocompression

The example uses the following definition for a join index defined on an *orders* table as used in the previous example. The system default is AUTO COMPRESS. However, the create table request overrides the default to set the columns to NO AUTO COMPRESS.

```
CREATE JOIN INDEX j_orders AS
  SELECT ROWID AS rw, o_custkey, o_salesperson, o_bill_addr
  FROM orders
  PARTITION BY COLUMN NO AUTO COMPRESS;
```

Modify the single-column partition *o_salesperson* to have autocompression.

```
ALTER TABLE j_orders
  ADD (o_salesperson) AUTO COMPRESS;
```

This request alters the definition of *j_orders* by changing the single-column partition consisting of column *o_salesperson* to have autocompression.

```
CREATE JOIN INDEX j_orders AS
  SELECT ROWID AS rw, o_custkey, o_salesperson, o_bill_addr
  FROM orders
  PARTITION BY COLUMN NO AUTO COMPRESS ALL BUT
    (o_salesperson AUTO COMPRESS);
```

Now modify the *o_bill_addr* single-column partition to have ROW storage format.

```
ALTER TABLE j_orders
  ADD ROW(o_bill_addr);
```

This request changes the definition of *j_orders* as follows.

```
CREATE JOIN INDEX j_orders AS
  SELECT ROWID AS rw, o_custkey, o_salesperson, o_bill_addr
  FROM Orders
  PARTITION BY COLUMN NO AUTO COMPRESS ALL BUT
    (o_salesperson AUTO COMPRESS,
     ROW(o_bill_addr));
```

ALTER TABLE RELEASE ROWS Examples

Example: Remove Logically Deleted Rows

```
ALTER TABLE lditab1 RELEASE ROWS;
```

Example: Remove Logically Deleted Rows and Reset RowLoadID

```
ALTER TABLE lditab12 RELEASE DELETED ROWS AND RESET LOAD IDENTITY;
```

ALTER TABLE ADD column_name Examples

Example: Adding or Modifying NUMBER Columns

Assume you create the following table.

```
CREATE TABLE num_tab (
  n1 NUMBER(*,3),
  n2 NUMBER,
  n3 NUMBER(*),
  n4 NUMBER(5,1));
```

This example modifies the precision of an existing NUMBER column in *num_tab*.

```
ALTER TABLE num_tab ADD n4 NUMBER(9,1);
```

This example modifies both the precision and the scale of an existing NUMBER column in the original definition of *num_tab*. The example is not valid if submitted for the revised definition of *num_tab* as it would be after Vantage processed the previous example where the precision of column *n4* was altered.

```
ALTER TABLE num_tab ADD n4 NUMBER(10,6);
```

This example adds a NUMBER column to *num_tab*.

```
ALTER TABLE num_tab ADD n6 NUMBER(12,4);
```

Example: Renaming a Column and Then Adding a New Column Using the Previous Name of the Renamed Column

You can rename an existing column, then create a new column, perhaps with a different data type, using the old name of the renamed column. For example, suppose you have the following table definition:

```
CREATE SET TABLE t1, NO FALLBACK, NO BEFORE JOURNAL,
                    NO AFTER JOURNAL,
    upi INTEGER NOT NULL,
    f1  FLOAT,
    d1  DECIMAL(7,2),
    i1  INTEGER)
UNIQUE PRIMARY INDEX(upi);
```

Rename column *f1* as *f2* and then add a new column named *f1* having the INTEGER data type rather than the FLOAT data type previously associated with the column now named *f2*:

```
ALTER TABLE t1
    RENAME f1 AS f2,
    ADD    f1 INTEGER;
*** Table has been modified.
*** Total elapsed time was 1 second.
```

Rename column *i1* as *i2* and then add a new column named *i1* having the DECIMAL data type rather than the INTEGER data type previously associated with the column now named *i2*:

```
ALTER TABLE t1
    RENAME i1 AS i2,
    ADD    i1 DECIMAL(8,3);
*** Table has been modified.
*** Total elapsed time was 1 second.
```

Display the new table definition using the SHOW TABLE statement. See [SHOW object](#).

```
SHOW TABLE t1;
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
CREATE SET TABLE  user_name.t1, NO FALLBACK,NO BEFORE JOURNAL,
                    NO AFTER JOURNAL,CHECKSUM = DEFAULT (
    upi INTEGER NOT NULL,
    f2  FLOAT,
    d1  DECIMAL(7,2),
    i2  INTEGER,
    f1  INTEGER,
    i1  DECIMAL(8,3)
UNIQUE PRIMARY INDEX (upi);
```

Example: Adding a LOB Column to a Table

The following example adds the CLOB column *extended_description* to a table named *partshistory*:

```
ALTER TABLE partshistory
ADD extended_description CLOB;
```

Example: Modifying a Table with JSON non-LOB and LOB Columns

Following is the table definition for a table with non-LOB and LOB JASON columns.

```
CREATE TABLE jsonTable (id INTEGER,
                        jsn1 JSON(1000) CHARACTER SET LATIN,
                        jsn2 JSON(1M) INLINE LENGTH 30000 CHARACTER SET LATIN);
```

This statement increases the maximum length of the non-LOB column.

```
ALTER TABLE jsonTable ADD jsn1 JSON(2000);
```

This statement increases the maximum length of the LOB column.

```
ALTER TABLE jsonTable
ADD jsn2 JSON(2M) INLINE LENGTH 30000 CHARACTER SET LATIN;
```

Example: Adding Columns to a Column Partition

This example adds two columns into the single-column partition for *o_comment* and modifies the partition to be a multicolumn partition with system-determined COLUMN format and autocompression.

Any of the following equivalent requests perform the same actions to alter the *orders* table.

```
ALTER TABLE orders
ADD (o_comment_ext1 VARCHAR(79), o_comment_ext2 VARCHAR(79))
INTO o_comment;
ALTER TABLE orders
ADD o_comment_ext1 VARCHAR(79) INTO o_comment,
ADD o_comment_ext2 VARCHAR(79) INTO o_comment;
ALTER TABLE orders
ADD (o_comment_ext1 VARCHAR(79)) INTO o_comment,
ADD (o_comment_ext2 VARCHAR(79)) INTO o_comment;
ALTER TABLE orders
ADD (o_comment_ext1 VARCHAR(79)) INTO o_comment,
ADD o_comment_ext2 VARCHAR(79) INTO o_comment;
ALTER TABLE orders
```

```
ADD o_comment_ext1 VARCHAR(79) INTO o_comment,
ADD (o_comment_ext2 VARCHAR(79)) INTO o_comment;
```

These all produce the following table definition with the new columns set to NULL in each row of the table.

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_ordertsz      TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_comment       VARCHAR(79),
  o_salesperson   VARCHAR(5),
  o_ordersubstatus CHARACTER(1) CASESPECIFIC,
  o_comment_ext1  VARCHAR(79),
  o_comment_ext2  VARCHAR(79) )
PARTITION BY (COLUMN
  ALL BUT ((o_orderstatus, o_ordersubstatus),
           (o_comment, o_comment_ext1, o_comment_ext2))
  ADD 4,
  RANGE_N(o_ordertsz
  BETWEEN TIMESTAMP '2003-01-01 00:00:00.000000+00:00'
  AND   TIMESTAMP '2009-12-31 23:59:59.999999+00:00'
  EACH INTERVAL '1' MONTH) ),
UNIQUE INDEX(o_orderkey);
```

Example: Adding a Column Partition to a Table with a Primary AMP Index

Assume the following table definition, with autocompression as the default.

```
CREATE TABLE Orders
( o_orderkey INTEGER NOT NULL,
  o_custkey  INTEGER,
  o_orderstatus CHAR(1) CASESPECIFIC,
  o_totalprice DECIMAL(13,2) NOT NULL,
  o_ordertsz  TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_comment   VARCHAR(79) )
PRIMARY AMP INDEX (o_orderkey), PARTITION BY COLUMN,
UNIQUE INDEX (o_orderkey);
```

The following ALTER TABLE statement adds a single-column partition, with system-determined COLUMN format and autocompression:

```
ALTER TABLE Orders ADD o_salesperson VARCHAR(5);
```

This results in a table definition as follows:

```
CREATE TABLE Orders
( o_orderkey INTEGER NOT NULL,
  o_custkey  INTEGER,
```

```

o_orderstatus CHAR(1) CASESPECIFIC,
o_totalprice DECIMAL(13,2) NOT NULL,
o_ordertsz TIMESTAMP(6) WITH TIME ZONE NOT NULL,
o_comment VARCHAR(79),
o_salesperson VARCHAR(5) )
PRIMARY AMP INDEX (o_orderkey) PARTITION BY COLUMN AUTO COMPRESS,
UNIQUE INDEX (o_orderkey);

```

Example: Adding Another Column Partition to a Table with a Primary AMP Index

The following ALTER TABLE statement adds another single-column partition, with system-determined COLUMN format and no autocompression as the default:

```
ALTER TABLE Orders ADD o_ordersubstatus CHAR(1) CASESPECIFIC;
```

This results in a table definition as follows:

```

CREATE TABLE Orders
( o_orderkey INTEGER NOT NULL,
  o_custkey INTEGER,
  o_orderstatus CHAR(1) CASESPECIFIC,
  o_totalprice DECIMAL(13,2) NOT NULL,
  o_ordertsz TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_comment VARCHAR(79),
  o_salesperson VARCHAR(5),
  o_ordersubstatus CHAR(1) CASESPECIFIC )
PRIMARY AMP INDEX (o_orderkey)
PARTITION BY
  COLUMN AUTO COMPRESS ALL BUT (o_ordersubstatus NO AUTO COMPRESS),
UNIQUE INDEX (o_orderkey);

```

Example: Adding a Two-Column Partition to a Table with a Primary AMP Index

The following ALTER TABLE statement adds a two-column partition, with system-determined COLUMN format and no autocompression as the default:

```
ALTER TABLE Orders ADD (o_comment_ext1 VARCHAR(79), o_comment_ext2 VARCHAR(79));
```

This results in a table definition as follows:

```

CREATE TABLE Orders
( o_orderkey INTEGER NOT NULL,

```

```

o_custkey INTEGER,
o_orderstatus CHAR(1) CASESPECIFIC,
o_totalprice DECIMAL(13,2) NOT NULL,
o_ordertsz TIMESTAMP(6) WITH TIME ZONE NOT NULL,
o_comment VARCHAR(79),
o_salesperson VARCHAR(5),
o_ordersubstatus CHAR(1) CASESPECIFIC,
o_comment_ext1 VARCHAR(79),
o_comment_ext2 VARCHAR(79) )
PRIMARY AMP INDEX (o_orderkey)
PARTITION BY
    COLUMN AUTO COMPRESS ALL BUT (
        o_ordersubstatus NO AUTO COMPRESS,
        (o_comment_ext1, o_comment_ext2) NO AUTO COMPRESS ),
UNIQUE INDEX(o_orderkey);

```

ALTER TABLE ADD COLUMN (*column_name*) Examples

Example: Adding a Two-Column Partition to a Table

This example uses a table definition with more columns than the version of *orders* used in [Example: Adding a Single-Column Partition to a Table](#). The system default is AUTO COMPRESS. The ALTER TABLE statements all add the same two-column partition, (*o_ship_addr*, *o_bill_addr*) to the *orders* table.

```

CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_ordertsz      TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_comment       VARCHAR(79),
  o_salesperson   VARCHAR(5),
  o_ordersubstatus CHARACTER(1) CASESPECIFIC,
  o_comment_ext1  VARCHAR(79),
  o_comment_ext2  VARCHAR(79))
PARTITION BY COLUMN AUTO COMPRESS ALL BUT (o_ordersubstatus
                                           NO AUTO COMPRESS,
                                           (o_comment_ext1,
                                            o_comment_ext2)
                                           NO AUTO COMPRESS),
UNIQUE INDEX(o_orderkey);

```


The following ALTER TABLE requests all add the same two-column partition (*o_ship_addr*, *o_bill_addr*) to the table with user-specified ROW format and autocompression.

```
ALTER TABLE orders
  ADD ROW(o_ship_addr VARCHAR(500), o_bill_addr VARCHAR(200))
  AUTO COMPRESS;
```

```
ALTER TABLE orders
  ADD ROW(o_ship_addr VARCHAR(500), o_bill_addr VARCHAR(200));
```

```
ALTER TABLE orders
  ADD (o_ship_addr VARCHAR(500), o_bill_addr VARCHAR(200));
```

```
ALTER TABLE orders
  ADD (o_ship_addr VARCHAR(500), o_bill_addr VARCHAR(200))
  AUTO COMPRESS;
```

```
ALTER TABLE orders
  ADD (o_ship_addr VARCHAR(500), o_bill_addr VARCHAR(200))
  NO AUTO COMPRESS;
```

```
ALTER TABLE orders
  ADD ROW(o_ship_addr VARCHAR(500), o_bill_addr VARCHAR(200))
  NO AUTO COMPRESS;
```

```
ALTER TABLE orders
  ADD o_bill_addr VARCHAR(200) INTO o_ship_addr,
  ADD ROW(o_ship_addr);
```

The resulting table has the following definition.

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_ordertsz      TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_comment       VARCHAR(79),
  o_salesperson   VARCHAR(5),
  o_ordersubstatus CHARACTER(1) CASESPECIFIC,
  o_comment_ext1  VARCHAR(79),
  o_comment_ext2  VARCHAR(79),
  o_ship_addr     VARCHAR(500),
```

```

    o_bill_addr      VARCHAR(200) )
PARTITION BY COLUMN AUTO COMPRESS ALL BUT (o_ordersubstatus
                                           NO AUTO COMPRESS,
                                           (o_comment_ext1,
                                           o_comment_ext2)
                                           NO AUTO COMPRESS,
                                           ROW(o_ship_addr, o_bill_addr)),
UNIQUE INDEX(o_orderkey);

```

Example: Adding a Two-Column and a Single-Column Partition to a Table with a Primary AMP Index

Assume the following table definition, with autocompression as the default.

```

CREATE TABLE Orders
( o_orderkey INTEGER NOT NULL,
  o_custkey INTEGER,
  o_orderstatus CHAR(1) CASESPECIFIC,
  o_totalprice DECIMAL(13,2) NOT NULL,
  o_ordertsz TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_comment VARCHAR(79) )
PRIMARY AMP INDEX (o_orderkey), PARTITION BY COLUMN,
UNIQUE INDEX (o_orderkey);

```

The following ALTER TABLE statement adds a two-column partition and a single-column partition to the table, with user-specified COLUMN format for the first partition, system-determined COLUMN format for the second partition, and autocompression by default.

```

ALTER TABLE Orders
ADD COLUMN(o_alt_ship_addr VARCHAR(500), o_alt_bill_addr VARCHAR(200)),
ADD o_item_count INTEGER;

```

This results in a table definition as follows:

```

CREATE TABLE Orders
( o_orderkey INTEGER NOT NULL,
  o_custkey INTEGER,
  o_orderstatus CHAR(1) CASESPECIFIC,
  o_totalprice DECIMAL(13,2) NOT NULL,
  o_ordertsz TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_comment VARCHAR(79),
  o_salesperson VARCHAR(5),
  o_ordersubstatus CHAR(1) CASESPECIFIC,
  o_comment_ext1 VARCHAR(79),

```

```

    o_comment_ext2 VARCHAR(79),
    o_ship_addr VARCHAR(500),
    o_bill_addr VARCHAR(200),
    o_alt_ship_addr VARCHAR(500),
    o_alt_bill_addr VARCHAR(200),
    o_item_count INTEGER )
PRIMARY AMP INDEX (o_orderkey)
PARTITION BY (
    COLUMN AUTO COMPRESS ALL BUT (
        o_ordersubstatus NO AUTO COMPRESS,
        (o_comment_ext1, o_comment_ext2) NO AUTO COMPRESS,
        ROW(o_ship_addr, o_bill_addr) NO AUTO COMPRESS,
        COLUMN(o_alt_ship_addr, o_alt_bill_addr) ),
    UNIQUE INDEX (o_orderkey);

```

ALTER TABLE ADD ROW (*column_name*) Examples

Example: Adding a ROW-Formatted Column Partition to a Column-Partitioned Table

The following example adds a two-column row partition to the column-partitioned *orders* table with user-specified ROW format and no autocompression.

```

ALTER TABLE orders
ADD ROW(o_ship_addr VARCHAR(500), o_bill_addr VARCHAR(200))
NO AUTO COMPRESS;

```

This produces the following a table definition with the new columns set to NULL.

```

CREATE TABLE orders (
    o_orderkey      INTEGER NOT NULL,
    o_custkey       INTEGER,
    o_orderstatus   CHARACTER(1) CASESPECIFIC,
    o_totalprice    DECIMAL(13,2) NOT NULL,
    o_ordertsz      TIMESTAMP(6) WITH TIME ZONE NOT NULL,
    o_comment       VARCHAR(79),
    o_salesperson   VARCHAR(5),
    o_ordersubstatus CHARACTER(1) CASESPECIFIC,
    o_comment_ext1  VARCHAR(79),
    o_comment_ext2  VARCHAR(79),
    o_ship_addr     VARCHAR(500),
    o_bill_addr     VARCHAR(200) )
PARTITION BY (COLUMN ALL BUT((o_ o_orderstatus, o_ordersubstatus),

```

```

(o_comment, o_comment_ext1,
  o_comment_ext2), ROW(o_ship_addr,
    o_bill_addr) NO AUTO COMPRESS)) ADD 3,
RANGE_N(o_ordertsz
  BETWEEN TIMESTAMP '2003-01-01 00:00:00.000000+00:00'
  AND      TIMESTAMP '2009-12-31 23:59:59.999999+00:00'
  EACH INTERVAL '1' MONTH) ),
UNIQUE INDEX(o_orderkey);

```

Example: Adding a Two-Column Partition with ROW Format to a Table with a Primary AMP Index

Assume the following table definition, with autocompression as the default.

```

CREATE TABLE Orders
  ( o_orderkey INTEGER NOT NULL,
    o_custkey INTEGER,
    o_orderstatus CHAR(1) CASESPECIFIC,
    o_totalprice DECIMAL(13,2) NOT NULL,
    o_ordertsz TIMESTAMP(6) WITH TIME ZONE NOT NULL,
    o_comment VARCHAR(79) )
PRIMARY AMP INDEX (o_orderkey), PARTITION BY COLUMN,
UNIQUE INDEX (o_orderkey);

```

The following ALTER TABLE statement adds a two-column partition to the table, with user-specified ROW format and autocompression by default:

```

ALTER TABLE Orders
  ADD ROW(o_ship_addr VARCHAR(500), o_bill_addr VARCHAR(200)) AUTO COMPRESS;

```

This results in a table definition as follows:

```

CREATE TABLE Orders
  ( o_orderkey INTEGER NOT NULL,
    o_custkey INTEGER,
    o_orderstatus CHAR(1) CASESPECIFIC,
    o_totalprice DECIMAL(13,2) NOT NULL,
    o_ordertsz TIMESTAMP(6) WITH TIME ZONE NOT NULL,
    o_comment VARCHAR(79),
    o_salesperson VARCHAR(5),
    o_ordersubstatus CHAR(1) CASESPECIFIC,
    o_comment_ext1 VARCHAR(79),
    o_comment_ext2 VARCHAR(79),
    o_ship_addr VARCHAR(500),

```

```

        o_bill_addr VARCHAR(200) )
PRIMARY AMP INDEX (o_orderkey)
PARTITION BY (
    COLUMN AUTO COMPRESS ALL BUT (
        o_orderstatus NO AUTO COMPRESS,
        (o_comment_ext1, o_comment_ext2) NO AUTO COMPRESS,
        ROW(o_ship_addr, o_bill_addr) ),
    UNIQUE INDEX(o_orderkey);

```

Example: Adding a Single-Column Partition to a Table

The system default is AUTO COMPRESS.

The example uses the following definition for an *orders* table.

```

CREATE TABLE orders (
    o_orderkey    INTEGER NOT NULL,
    o_custkey     INTEGER,
    o_orderstatus CHARACTER(1) CASESPECIFIC,
    o_totalprice  DECIMAL(13,2) NOT NULL,
    o_ordertsz    TIMESTAMP(6) WITH TIME ZONE NOT NULL,
    o_comment     VARCHAR(79) )
NO PRIMARY INDEX,
PARTITION BY COLUMN,
UNIQUE INDEX(o_orderkey);

```

The following ALTER TABLE requests all add the same single-column, o_salesperson, which by default is also a single-column partition with system-determined COLUMN format and autocompression to *orders*.

```

ALTER TABLE orders
    ADD o_salesperson VARCHAR(5);
ALTER TABLE orders
    ADD (o_salesperson VARCHAR(5));
ALTER TABLE orders
    ADD SYSTEM(o_salesperson VARCHAR(5)) AUTO COMPRESS;
ALTER TABLE orders
    ADD SYSTEM(o_salesperson VARCHAR(5));
ALTER TABLE orders
    ADD (o_salesperson VARCHAR(5)) AUTO COMPRESS;

```

The resulting table has the following definition.

```
CREATE TABLE orders (
  o_orderkey INTEGER NOT NULL,
  o_custkey INTEGER,
  o_orderstatus CHAR(1) CASESPECIFIC,
  o_totalprice DECIMAL(13,2) NOT NULL,
  o_ordertsz TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_comment VARCHAR(79),
  o_salesperson VARCHAR(5) )
PARTITION BY COLUMN AUTO COMPRESS,
UNIQUE INDEX(o_orderkey);
```

Example: Adding a Single-Column Partition

The following example adds a single-column partition using system-determined COLUMN format and autocompression to the *orders* table.

```
ALTER TABLE orders
ADD (o_salesperson VARCHAR(5));
```

The following equivalent request performs the same actions to alter the table.

```
ALTER TABLE orders
ADD o_salesperson VARCHAR(5);
```

Both requests alter the *orders* table to have the following definition with *o_salesperson* set to NULL in each row of the table.

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_ordertsz      TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_comment       VARCHAR(79),
  o_salesperson   VARCHAR(5) )
PARTITION BY (COLUMN ADD 4,
RANGE_N(o_ordertsz
  BETWEEN TIMESTAMP '2003-01-01 00:00:00.000000+00:00'
  AND   TIMESTAMP '2009-12-31 23:59:59.999999+00:00'
  EACH INTERVAL '1' MONTH) ),
UNIQUE INDEX(o_orderkey);
```

Example: Adding a Column to a Single-Column Partition

This example adds a column into the single-column partition for *o_orderstatus*, modifying the partition to be a multicolumn partition with system-determined COLUMN format and autocompression.

```
ALTER TABLE orders
ADD o_orderstatus CHAR(1) CASESPECIFIC INTO o_orderstatus;
```

The following equivalent request performs the same actions to alter the *orders* table.

```
ALTER TABLE orders
ADD (o_orderstatus CHAR(1) CASESPECIFIC) INTO o_orderstatus;
```

Both requests alter the *orders* table to have the following definition with *o_orderstatus* set to NULL in each row of the table.

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL
  o_custkey       INTEGER
  o_orderstatus   CHARACTER(1) CASESPECIFIC
  o_totalprice    DECIMAL(13,2) NOT NULL
  o_ordertsz      TIMESTAMP(6) WITH TIME ZONE NOT NULL
  o_comment       VARCHAR(79)
  o_salesperson   VARCHAR(5)
  o_orderstatus   CHARACTER(1) CASESPECIFIC)
PARTITION BY (COLUMN ALL BUT ((o_orderstatus, o_orderstatus)))
      ADD 4,
      RANGE_N(o_ordertsz BETWEEN TIMESTAMP
              '2003-01-01 00:00:00.000000+00:00'
              AND    TIMESTAMP
              '2009-12-31 23:59:59.999999+00:00'
              EACH INTERVAL '1' MONTH) ),
UNIQUE INDEX(o_orderkey);
```

Example: Attempting to Add and Drop Constraints in the Same Statement

The following request is not valid because you can add or drop only one constraint per ALTER TABLE request:

```
ALTER TABLE table_1
  DROP CONSTRAINT check_1,
  ADD CONSTRAINT check_2 CHECK (column_2 > 0);
```

The system returns the following message:

```
Only a check specification is allowed for the modification.
```

Example: Attempting to Add a Constraint with a Duplicate Name

The following request is not valid because the constraint named *dup_constr_name* already exists in *table_1*. You cannot add it to the table.

```
ALTER TABLE table_1
  ADD CONSTRAINT dup_constr_name
  FOREIGN KEY (column_3) REFERENCES table_2;
```

The system returns the following message:

```
Constraint with the same name 'dup_constr_name' already exists in table.
```

Example: Adding a Row-Level Security Constraint

Specify the name of a security constraint object to add a corresponding constraint column to a table. The table cannot have any hash indexes or join indexes and can contain a maximum of 5 security constraint columns.

```
ALTER TABLE table_name
  ADD security_constraint_name
  CONSTRAINT ;
```

Example: Adding a FOREIGN KEY Constraint

Add a foreign key constraint on the column named *column_2* in *table_1*:

```
ALTER TABLE table_1 ADD FOREIGN KEY (column_2) REFERENCES table_3;
```


Example: Adding or Dropping a Batch Referential Constraint

This example first adds a table-level batch referential constraint to column *d1* in table *drs.t2* referencing column *c1* in table *drs.t1* and then drops that same constraint.

```
CREATE SET TABLE drs.t1, NO FALLBACK, NO BEFORE JOURNAL,
                      NO AFTER JOURNAL (
    c1 INTEGER NOT NULL,
    c2 INTEGER NOT NULL,
    c3 INTEGER NOT NULL)
UNIQUE PRIMARY INDEX (c1);
CREATE SET TABLE drs.t2, NO FALLBACK, NO BEFORE JOURNAL,
                      NO AFTER JOURNAL (
    d1 INTEGER,
    d2 INTEGER,
    d3 INTEGER);
ALTER TABLE drs.t2
ADD CONSTRAINT fpk1
FOREIGN KEY (d1) REFERENCES WITH CHECK OPTION drs.t1 (c1);
ALTER TABLE drs.t2
DROP FOREIGN KEY (d1) REFERENCES WITH CHECK OPTION drs.t1 (c1);
```

Example: Adding a Table-Level Referential Constraint

The first request adds a table-level Referential Constraint relationship on foreign key column *a3* with column *e1* in table *e*. Referential integrity is not enforced for this relationship.

The second request drops the foreign key Referential Constraint that was created by the first.

```
ALTER TABLE a
ADD FOREIGN KEY (a3) REFERENCES WITH NO CHECK OPTION e(e1);

ALTER TABLE a
DROP FOREIGN KEY (a3) REFERENCES e(e1);
```

Example: Adding a NORMALIZE Constraint to a Table

These examples show how to use an ALTER TABLE request to add a NORMALIZE constraint to a table. The system default condition for a NORMALIZE constraint is:

```
ON MEETS OR OVERLAPS
```

The examples all use the following table definition or an equivalent temporal table definition. The table definition does not specify a NORMALIZE constraint.

```
CREATE TABLE project (
  emp_id      INTEGER,
  project_name VARCHAR(20),
  dept_id     INTEGER,
  duration    PERIOD(DATE));
```

The following ALTER TABLE request modifies the *project* table to add a NORMALIZE constraint on the *duration* column. Rows are normalized only when the values of the *emp_id*, *project_name*, and *dept_id* columns are the same and the Period values for the *duration* column either meet or overlap.

```
ALTER TABLE project
ADD NORMALIZE ON duration;
```

The following ALTER TABLE request modifies the *project* table to add a NORMALIZE constraint on the *duration* column. The *dept_id* column is the only column to ignore for normalization.

Rows are normalized only when the values of columns *emp_id* and *project_name* are the same and the Period values for the *duration* column meet or overlap.

```
ALTER TABLE project
ADD NORMALIZE ALL BUT(dept_id) ON duration;
```

The following ALTER TABLE request modifies the *project* table to add a NORMALIZE constraint on the *duration* column that applies only when the Period values for the *duration* column overlap. The *dept_id* column is the only column to ignore for normalization.

Rows are normalized only when the values of columns *emp_id* and *project_name* are the same and the Period values for the *duration* column overlap.

```
ALTER TABLE project
ADD NORMALIZE ALL BUT(dept_id) ON duration ON OVERLAPS;
```

The following ALTER TABLE request modifies the *project* table to add a NORMALIZE constraint on the *duration* column that applies only when the Period values for the *duration* column overlap. The *dept_id* column is the only column to ignore for normalization.

Rows are normalized only when the values of columns *emp_id* and *project_name* are the same and the Period values for the *duration* column overlap.

```
ALTER TABLE project
ADD NORMALIZE ALL BUT(dept_id) ON duration ON OVERLAPS;
```

The next example defines the *project* table using a NORMALIZE condition of ON OVERLAPS on the *duration* column.

```
CREATE TABLE project (
  emp_id INTEGER,
  project_name VARCHAR(20),
  dept_id INTEGER,
  duration PERIOD(DATE),
  NORMALIZE ALL BUT (dept_id) ON duration ON OVERLAPS);
```

Modify the definition of *project* to add the *project_name* column to the ALL BUT *normalize_ignore* column list and to add the condition OR MEETS to the existing ON OVERLAPS condition.

```
ALTER TABLE
ADD NORMALIZE ALL BUT(dept_id,project_name) ON duration ON OVERLAPS
OR MEETS;
```

Vantage renormalizes the *project* table after you submit this ALTER TABLE request.

Example: Adding a Foreign Key to a Column-Partitioned Table

Suppose that after defining tables *p* and *c*, you decide to add a foreign key relationship between these tables, with column-partitioned table *c* referencing table *p* using a table-level foreign key constraint.

Table *c* has the following definition.

```
CREATE TABLE c (
  c1 INTEGER,
  cb INTEGER,
  cd1 DATE,
  cc2 CHARACTER(30))
PARTITION BY (COLUMN,
              RANGE_N(cd1 BETWEEN DATE '2006-01-01'
                      AND DATE '2020-12-31'
                      EACH INTERVAL '1' MONTH));
```

Table *p* has the following definition.

```
CREATE TABLE p (
  p1 INTEGER,
  pb INTEGER NOT NULL UNIQUE,
  pc1 CHARACTER(10))
PRIMARY INDEX (p1);
```

You submit the following ALTER TABLE statement to add the desired table-level foreign key constraint to table *c*, with column *c.cb* referencing column *p.pb*.

```
ALTER TABLE c ADD FOREIGN KEY (cb) REFERENCES p (pb);
```

Because this foreign key only defines a relationship between a single column in the referencing and the referenced tables, you could just as well have defined it as a column-level constraint like the following.

```
ALTER TABLE c ADD cb REFERENCES p (pb);
```

These ALTER TABLE statements both produce the following error table definition.

```
CREATE TABLE c_0 (
  c1 INTEGER,
  cb INTEGER,
  cd1 DATE,
  cd2 CHARACTER(30))
NO PRIMARY INDEX;
```

Example: Adding a Named CHECK Constraint to a Table

Constraint name *check_1* must not be an existing constraint name in *table_1*.

```
ALTER TABLE table_1 ADD CONSTRAINT check_1 CHECK (column_1 > column_2);
```

Example: Adding an Unnamed CHECK Constraint

Add an unnamed CHECK constraint to ensure that values in column *column_2* are always greater than 100:

```
ALTER TABLE table_1 ADD CHECK (column_2 > 100);
```

Example: Adding a Column With an Unnamed CHECK Constraint

Add an unnamed CHECK constraint to ensure that values in column *column_1* are always greater than 0.

This example is valid only if there is not an existing unnamed column level CHECK for *column_1* in *table_1*.

```
ALTER TABLE table_1 ADD CONSTRAINT column_1 CHECK (column_1 > 0);
```

Example: Adding a Named UNIQUE Constraint to a Table

Constraint name *unique_1* must not be an existing constraint name in *table_1*.

```
ALTER TABLE table_1 ADD CONSTRAINT unique_1 UNIQUE (column_1, column_2);
```

Example: Adding a Multicolumn Unnamed UNIQUE Constraint

Add an unnamed uniqueness constraint to the columns named *column_3* and *column_4*:

```
ALTER TABLE table_1 ADD UNIQUE (column_3, column_4);
```

Example: Adding a PRIMARY KEY Constraint to a Table

table_1 must not already have a defined PRIMARY KEY constraint. Constraint name *primary_1* must not duplicate an existing constraint name in *table_1*.

```
ALTER TABLE table_1
  ADD CONSTRAINT primary_1 PRIMARY KEY (column_1, column_2);
```

column_1 and *column_2* must be NOT NULL or an error is returned.

Example: Modifying a Named CHECK Constraint

Replace the current boolean condition for CHECK constraint *check_1* with a new boolean condition.

Constraint name *check_1* must be an existing CHECK constraint.

```
ALTER TABLE table_1 MODIFY CONSTRAINT check_1 CHECK (column_2 > 0);
```

Example: Attempting to Modify a Nonexistent Constraint

The following request is not valid because the constraint named *no_such_constr* does not exist in *table_1*. You cannot modify or drop it.

```
ALTER TABLE table_1
  MODIFY CONSTRAINT no_such_constr
  CHECK (column_1 > 0);
```

Example: Adding a Derived Period Column to a Table

This example modifies a table to add a derived period column.

Here is the table definition for this example.

```
CREATE TABLE employee (
  eid INTEGER NOT NULL,
  name VARCHAR(100) NOT NULL,
  deptno INTEGER NOT NULL,
  jobstart DATE NOT NULL,
  jobend DATE NOT NULL
) PRIMARY INDEX(eid);
```

This following statement modifies the table, *employee*, to add a derived period column, *jobduration*, with *jobstart* as the period start column and *jobend* as the period end column.

```
ALTER TABLE employee ADD PERIOD FOR jobduration(jobstart, jobend);
```

Example: Dropping the IDENTITY Attribute From a Column Without Dropping the Column

Suppose you create the following identity column table named *id_phone*.

```
CREATE TABLE id_phone(
  id_num INTEGER GENERATED ALWAYS AS IDENTITY
          (START WITH 1000
           INCREMENT BY 10
           MINVALUE 0
           MAXVALUE 300000),
  phone INTEGER)
UNIQUE PRIMARY INDEX(idnum);
```

After some time, you decide to use Teradata Unity to manage multiple Vantage instances, including one that includes the *id_phone* table. Because Teradata Unity requires deterministic behavior to ensure data consistency at each Vantage instance, you must remove the identity column attribute from the *id_num* column, but you must also retain the column and its data with *id_phone* because *id_num* is also the unique primary index for the table. You can use the following statement:

```
ALTER TABLE id_phone
DROP id_num IDENTITY;
```

Column `id_num` in table `id_phone` is no longer an identity column, but it continues to be the unique primary index for the table.

Example: Dropping a Row-Level Security Constraint

Dropping a security constraint column from a table is similar to dropping any other column:

```
ALTER TABLE table_name
  DROP security_constraint_column_name
;
```

Example: Dropping a Column from a Column Partition and Modifying it to Become a Two-Column Partition

This example drops a column from a column partition of *orders* and modifies that partition to become a two-column partition with system-determined COLUMN format and autocompression.

```
ALTER TABLE orders
  DROP o_comment_ext2;
```

This results in the following table definition.

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_ordertsz      TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_comment       VARCHAR(79),
  o_salesperson   VARCHAR(5),
  o_ordersubstatus CHARACTER(1) CASESPECIFIC,
  o_comment_ext1  VARCHAR(79),
  o_ship_addr     VARCHAR(500),
  o_bill_addr     VARCHAR(200),
  o_alt_ship_addr VARCHAR(500),
  o_alt_bill_addr VARCHAR(200),
  o_item_count    INTEGER )
PARTITION BY (COLUMN ALL BUT ((o_orderstatus, o_ordersubstatus),
                              (o_comment, o_comment_ext1),
                              ROW(o_ship_addr, o_bill_addr) NO AUTO COMPRESS,
                              COLUMN(o_alt_ship_addr, o_alt_bill_addr)) ADD 1,
```

```

        RANGE_N(o_ordertsz
        BETWEEN TIMESTAMP '2003-01-01 00:00:00.000000+00:00'
        AND      TIMESTAMP '2009-12-31 23:59:59.999999+00:00'
        EACH INTERVAL '1' MONTH) ),
    UNIQUE INDEX(o_orderkey);

```

Example: Dropping a Column from a Column Partition and Modifying it to become a Single-Column Partition

This example drops a column from a column partition and modifies that column partition to become a single-column partition with system-determined COLUMN format and autocompression.

```

ALTER TABLE orders
DROP o_comment;

```

This results in the following table definition.

```

CREATE TABLE orders (
    o_orderkey      INTEGER NOT NULL,
    o_custkey       INTEGER,
    o_orderstatus   CHARACTER(1) CASESPECIFIC,
    o_totalprice    DECIMAL(13,2) NOT NULL,
    o_ordertsz      TIMESTAMP(6) WITH TIME ZONE NOT NULL,
    o_salesperson   VARCHAR(5),
    o_ordersubstatus CHARACTER(1) CASESPECIFIC,
    o_comment_ext1  VARCHAR(79),
    o_ship_addr     VARCHAR(500),
    o_bill_addr     VARCHAR(200),
    o_alt_ship_addr VARCHAR(500),
    o_alt_bill_addr VARCHAR(200),
    o_item_count    INTEGER )
PARTITION BY (COLUMN ALL BUT ( (o_orderstatus, o_ordersubstatus),
                                ROW(o_ship_addr, o_bill_addr) NO AUTO COMPRESS,
                                COLUMN(o_alt_ship_addr, o_alt_bill_addr)) ADD 1,
              RANGE_N(o_ordertsz
              BETWEEN TIMESTAMP '2003-01-01 00:00:00.000000+00:00'
              AND      TIMESTAMP '2009-12-31 23:59:59.999999+00:00'
              EACH INTERVAL '1' MONTH) ),
    UNIQUE INDEX(o_orderkey);

```


Example: Dropping a Column from a Column Partition Which Also Drops the Partition

This example drops a column from a column partition and, because it is the last column in the partition, also drops the partition.

```
ALTER TABLE orders
DROP o_comment_ext1;
```

This results in the following table definition for *orders*. Note that the ADD option value for level 1 increases to 2.

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_ordertsz      TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_salesperson   VARCHAR(5),
  o_ordersubstatus CHARACTER(1) CASESPECIFIC,
  o_ship_addr     VARCHAR(500),
  o_bill_addr     VARCHAR(200),
  o_alt_ship_addr VARCHAR(500),
  o_alt_bill_addr VARCHAR(200),
  o_item_count    INTEGER )
PARTITION BY (COLUMN ALL BUT ( (o_orderstatus, o_ordersubstatus),
                                ROW(o_ship_addr, o_bill_addr) NO AUTO COMPRESS,
                                COLUMN(o_alt_ship_addr, o_alt_bill_addr)) ADD 2,
              RANGE_N(o_ordertsz
                      BETWEEN TIMESTAMP '2003-01-01 00:00:00.000000+00:00'
                      AND   TIMESTAMP '2009-12-31 23:59:59.999999+00:00'
                      EACH INTERVAL '1' MONTH) ),
UNIQUE INDEX(o_orderkey);
```

Example: Dropping 2 Columns from a Column Partition Which Also Drops the Partition

This example drops two columns from a column partition, and because that leaves the partitions with no columns, also drops the column partition.

```
ALTER TABLE orders
DROP o_alt_ship_addr,
DROP o_alt_bill_addr;
```

This results in a table definition as follows. note that ADD option value for level 1 increases to 3.

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_ordertsz      TIMESTAMP(6) WITH TIME ZONE NOT NULL,
  o_salesperson   VARCHAR(5),
  o_ordersubstatus CHARACTER(1) CASESPECIFIC,
  o_ship_addr     VARCHAR(500),
  o_bill_addr     VARCHAR(200),
  o_item_count    INTEGER )
PARTITION BY (COLUMN ALL BUT ( (o_orderstatus, o_ordersubstatus),
                                ROW(o_ship_addr, o_bill_addr)
                                NO AUTO COMPRESS) ADD 3,
              RANGE_N(o_ordertsz
                BETWEEN TIMESTAMP '2003-01-01 00:00:00.000000+00:00'
                AND   TIMESTAMP '2009-12-31 23:59:59.999999+00:00'
                EACH INTERVAL '1' MONTH) ),
  UNIQUE INDEX(o_orderkey);
```

Example: Dropping All But 1 Column from a Column Partition

The following example drops 9 columns from the unpopulated *orders* table, leaving only 1 column in one column partition.

```
ALTER TABLE orders
DROP o_custkey,
DROP o_orderstatus,
DROP o_totalprice,
DROP o_ordertsz,
DROP o_salesperson,
DROP o_ordersubstatus,
DROP o_ship_addr,
DROP o_bill_addr,
DROP o_item_count;
```

This results in the following table definition.

```
CREATE TABLE orders (  
    o_orderkey INTEGER NOT NULL )  
PARTITION BY COLUMN ADD 65531;
```

Example: Dropping a Named Constraint From a Table

Constraint name *check_1* must exist in *table_1*.

```
ALTER TABLE table_1 DROP CONSTRAINT check_1;
```

Example: Dropping a Named CHECK Constraint From a Table

Constraint name *check_1* must exist in *table_1* as a CHECK constraint.

```
ALTER TABLE table_1 DROP CONSTRAINT check_1 CHECK;
```

Example: Dropping a UNIQUE Constraint From a Table

Constraint name *unique_1* must exist in *table_1*.

```
ALTER TABLE table_1 DROP CONSTRAINT unique_1;
```

Example: Dropping a Named Constraint From a Table

Constraint name *reference_1* must exist in *table_1*.

```
ALTER TABLE table_1 DROP CONSTRAINT reference_1;
```

Example: Dropping an Unnamed Column-Level CHECK Constraint

Drop the unnamed column level CHECK constraint from column *column_1* in *table_1*.

```
ALTER TABLE table_1 DROP column_1 CHECK;
```

Example: Adding an Unnamed CHECK Constraint

Modify a column by adding an unnamed CHECK constraint. Column *column_1* is an existing, already constrained column in *table_1*.

```
ALTER TABLE table_1 MODIFY column_1 CHECK (column_1 IS NOT NULL);
```

Example: Attempting to Drop a Nonexistent Constraint

The following request is not valid because the specified constraint does not exist and, therefore, cannot be dropped.

```
ALTER TABLE table_1 DROP CONSTRAINT no_such_constr;
```

The system returns the following message:

```
The specified constraint name does not exist in table.
```

Example: Adding a Named FOREIGN KEY Constraint to a Table

Constraint name *reference_1* must not be an existing constraint name in *table_1*.

```
ALTER TABLE table_1 ADD CONSTRAINT reference_1  
FOREIGN KEY (column_1) REFERENCES table_2;
```

Example: Dropping a FOREIGN KEY Constraint

Drop the foreign key constraint on the column named *column_2* in table *table_3*.

```
ALTER TABLE table_1 DROP FOREIGN KEY (column_2) REFERENCES table_3;
```

Example: Dropping All Unnamed Table-Level CHECK Constraints

Drop all unnamed table-level CHECK constraints.

```
ALTER TABLE table_1 DROP CHECK;
```

Example: Adding Fallback to a Table

Use the following to implement fallback protection for the *employee* table.

```
ALTER TABLE employee, Fallback;
```

Example: Adding a Single Before-Image Journal and Dual After-Image Journal to a Table

If a table named *newemp* is set for no journaling, but the database has a defined journal, the following request could be used to add a single before-image journal and a dual after-image journal:

```
ALTER TABLE newemp, BEFORE JOURNAL, DUAL AFTER JOURNAL;
```

Example: Changing a Single Before-Image Journal to a Dual Before-Image Journal and Dropping Two Columns

The following request changes the single before image journal to a dual image journal and also drops two columns:

```
ALTER TABLE newemp,  
    DUAL BEFORE JOURNAL  
    DROP phone,  
    DROP pref;
```

Example: Changing CHECKSUM to the Default

The following ALTER TABLE request changes the disk I/O integrity checksum value to the system default:

```
ALTER TABLE employee, CHECKSUM = DEFAULT;
```

Example: Changing FREESPACE

The following ALTER TABLE request changes the free space to 5 percent:

```
ALTER TABLE employee, FREESPACE = 5 PERCENT;
```

Example: Modifying MERGEBLOCKRATIO

These examples demonstrate how to use ALTER TABLE requests to modify the merge block ratio setting for an individual table.

The first example changes the default merge block ratio for emp_table so that the system retrieves the current value for the DBS Control parameter MergeBlockRatio and applies this value when merging smaller data blocks into a larger data block.

```
ALTER TABLE emp_table, DEFAULT MERGEBLOCKRATIO;
```

The next example modifies the current setting of the merge block ratio for emp_table to 25%.

```
ALTER TABLE emp_table, MERGEBLOCKRATIO = 25 PERCENT;
```

The final example disables all data block merging for emp_table.

```
ALTER TABLE emp_table, NO MERGEBLOCKRATIO;
```

Example: Changing DATABLOCKSIZE

The following ALTER TABLE request changes the maximum data block size to 1,048,064 bytes (2047 sectors) on a system with large cylinders and repacks existing data into blocks of that size:

```
ALTER TABLE employee, MAXIMUM DATABLOCKSIZE IMMEDIATE;
```

Example: Adding a TITLE Phrase to a Column

The following example changes the attributes of the dept_no column in the department table to include a TITLE phrase:

```
ALTER TABLE personnel.department ADD dept_no TITLE 'Depart';
```

Example: Adding the Null Attribute to Columns

The following requests add the NULL qualifier to columns training_skill and column_2 :

```
ALTER TABLE personnel ADD training_skill NULL;
```

```
ALTER TABLE abc ADD column_2 NULL;
```

Example: Making Composite Changes to a Column: NULL to NOT NULL

A composite change to a column implies changing more than one attribute of that column in one request. NULL columns can be changed to NOT NULL, along with other attribute changes, in one request. The following example changes both the case and NULL attribute of *column_2*. An error is returned if abc contains any rows with nulls in *column_2*.

```
ALTER TABLE abc ADD column_2 CASESPECIFIC NOT NULL;
```

Example: Making Composite Changes to a Column: NOT NULL to NULL

Composite changes to columns that involve changing the NOT NULL attribute to NULL are not valid. For example, the following cannot be done:

```
ALTER TABLE abc ADD column_2 TITLE 'newname' NULL;
```

To make these changes, you must perform two separate ALTER TABLE requests:

```
ALTER TABLE abc ADD column_2 TITLE 'newname';
ALTER TABLE abc ADD column_2 NULL;
```

Examples: Adding and Removing the JSON Auto Column Option

Here is the table definition for these examples.

```
CREATE TABLE MyTable (
  a INTEGER,
  b INTEGER,
  c JSON(64000) CHARACTER SET LATIN AUTO COLUMN);
```

This statement removes the auto column capability from column c.

```
ALTER TABLE MyTable
  c JSON(64000) CHARACTER SET LATIN NOT AUTO COLUMN;
```

This statement adds the column j with the JSON data type, a 4K byte length, BSON storage format, auto column capability, and a default value of NULL.

```
ALTER TABLE MyTable
  ADD j JSON(64000) STORAGE FORMAT BSON AUTO COLUMN NULL;
```

Example: Changing NO COMPRESS for an Existing Column

The following example alters a column named *qty_shipped_2006* in the *customer* table and makes that column uncompressible using multivalued compression. The column *qty_shipped* must already be defined for *customer* because you do not specify a data type for it. In that case, Vantage interprets the ADD specification to represent a request to modify an existing column.

```
ALTER TABLE customer ADD qty_shipped_2006 NO COMPRESS;
```

Example: Changing the Value Compression for a Compressed Column

Suppose a company that does business only in Santa Monica decides to expand. Previously, the city name Santa Monica was value-compressed. Now, the city name Los Angeles is to be value-compressed. The following example changes the single value-compressed city name value from Santa Monica to Los Angeles, which means that it should also be value-compressed:

```
ALTER TABLE customer ADD city_name COMPRESS 'Los Angeles';
```

If *city_name* had not been multivalued-compressed previously, this request would multivalued-compress all occurrences of Los Angeles. Note that this is just an example, because Santa Monica and Los Angeles can be multivalued-compressed.

Example: Adding a New Column With Multivalued Compression

The following ALTER TABLE request adds a new compressible column and specifies the maximum number of distinct values for compression.

```
ALTER TABLE mkw.x ADD b CHARACTER(2)
  COMPRESS (
    'a1','a2','a3','a4','a5','a6','a7','a8','a9','a10',
    'b1','b2','b3','b4','b5','b6','b7','b8','b9','b10',
    'c1','c2','c3','c4','c5','c6','c7','c8','c9','c10',
    'd1','d2','d3','d4','d5','d6','d7','d8','d9','d10',
    'e1','e2','e3','e4','e5','e6','e7','e8','e9','e10',
    'f1','f2','f3','f4','f5','f6','f7','f8','f9','f10',
    'g1','g2','g3','g4','g5','g6','g7','g8','g9','g10',
    'h1','h2','h3','h4','h5','h6','h7','h8','h9','h10',
    'i1','i2','i3','i4','i5','i6','i7','i8','i9','i10',
    'j1','j2','j3','j4','j5','j6','j7','j8','j9','j10',
    'k1','k2','k3','k4','k5','k6','k7','k8','k9','k10',
    'l1','l2','l3','l4','l5','l6','l7','l8','l9','l10',
```



```
'm1','m2','m3','m4','m5','m6','m7','m8','m9','m10',
'n1','n2','n3','n4','n5','n6','n7','n8','n9','n10',
'o1','o2','o3','o4','o5','o6','o7','o8','o9','o10',
'p1','p2','p3','p4','p5','p6','p7','p8','p9','p10',
'q1','q2','q3','q4','q5','q6','q7','q8','q9','q10',
'r1','r2','r3','r4','r5','r6','r7','r8','r9','r10',
's1','s2','s3','s4','s5','s6','s7','s8','s9','s10',
't1','t2','t3','t4','t5','t6','t7','t8','t9','t10',
'u1','u2','u3','u4','u5','u6','u7','u8','u9','u10',
'v1','v2','v3','v4','v5','v6','v7','v8','v9','v10',
'w1','w2','w3','w4','w5','w6','w7','w8','w9','w10',
'x1','x2','x3','x4','x5','x6','x7','x8','x9','x10',
'y1','y2','y3','y4','y5','y6','y7','y8','y9','y10',
'z1','z2','z3','z4','z5');
```

Example: Adding and Changing Algorithmic Compression

These examples use the following table definition.

```
CREATE TABLE Pendants (
  col_1 INTEGER,
  col_2 CHARACTER (10));
```

This example adds column *c3*, which specifies algorithmic compression, to table *Pendants*.

```
ALTER TABLE Pendants
ADD col_3 CHARACTER(30) COMPRESS USING compress_udf
DECOMPRESS USING decompress_udf;
```

This example adds column *c3*, which specifies algorithmic and multivalue compression, to table *Pendants*.

```
ALTER TABLE Pendants
ADD col_3 CHARACTER(30) COMPRESS ('amethyst', 'amber')
COMPRESS USING compress_udf
DECOMPRESS USING decompress_udf;
```

This example changes the definition of *Pendants.col_2* to add algorithmic compression. Because *Pendants* is empty, this ALTER TABLE request completes without error.

```
ALTER TABLE Pendants
ADD col_2 CHARACTER(10) COMPRESS
```

```

COMPRESS USING compress_udf
DECOMPRESS USING decompress_udf;

```

This example is the same as the previous example except that a row has been added to *Pendants*. Because *Pendants* is not empty, the system returns an error to the requestor.

```

INSERT INTO Pendants(1, 'amber');
ALTER TABLE Pendants
ADD col_2 CHARACTER(10) COMPRESS
                        COMPRESS USING compress_udf
                        DECOMPRESS USING decompress_udf;

```

Table Definitions for Examples

This topic provides the table definitions for the following examples:

[Example: Non-Valid MODIFY PRIMARY INDEX Statements](#) through [Example: Drop and Add Partition Ranges and Delete Rows Outside the Defined Ranges](#) and [Example: Using MODIFY to Repartition a Table and Saving Resulting Nonvalid Rows in a Save Table](#) through [Example: Revalidating the Partitioning for a Table](#) use the following tables.

The *orders* table has a primary index on *o_orderkey* and single-level partitioning. The table also has a USI defined on *o_orderkey*.

```

CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk         CHARACTER(16),
  o_shippriority  INTEGER,
  o_comment       VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_orderdate
                     BETWEEN DATE '1992-01-01'
                     AND      DATE '1998-12-31'
                     EACH INTERVAL '1' MONTH)
UNIQUE INDEX (o_orderkey);

```

The *orders_cp* table is multilevel column-partitioned with row partitioning on the second partitioning level that is defined using the same partitioning expression as the only partitioning level of *orders*. The table also has a USI defined on *o_orderkey*.

```

CREATE TABLE orders_cp (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk         CHARACTER(16),
  o_shippriority  INTEGER,
  o_comment       VARCHAR(79))
NO PRIMARY INDEX
PARTITION BY (COLUMN,
              RANGE_N(o_orderdate BETWEEN DATE '1992-01-01'
                        AND      DATE '1998-12-31'
                        EACH INTERVAL '1' MONTH))

UNIQUE INDEX (o_orderkey);

```

Example: Modifying Partition Ranges

This request modifies the table created in [Table Definitions for Examples](#). The request is valid if the table is empty.

```

ALTER TABLE orders
MODIFY
PARTITION BY RANGE_N(o_orderdate BETWEEN DATE '1993-01-01'
                      AND      DATE '2000-12-31'
                      EACH INTERVAL '1' MONTH);

```

The following request modifies the partitioning for the column-partitioned table *orders_cp*. The request is valid if the table is empty.

```

ALTER TABLE orders_cp
MODIFY
PARTITION BY (RANGE_N(o_orderdate BETWEEN DATE '1993-01-01'
                      AND      DATE '2000-12-31'
                      EACH INTERVAL '1' MONTH)

              COLUMN);

```

Example: Non-Valid MODIFY PRIMARY INDEX Statements

The following requests are not valid. These examples do not specify options that alter the table.

```
ALTER TABLE orders
MODIFY PRIMARY INDEX;

ALTER TABLE orders
MODIFY NOT UNIQUE PRIMARY INDEX;
```

The next example attempts to alter the table not to have a primary index and to add column and row partitioning. The request uses correct syntax but is not valid because *orders* is populated with rows.

```
ALTER TABLE orders
MODIFY NO PRIMARY INDEX
PARTITION BY (COLUMN ADD 5,
              RANGE_N(o_ordertsz BETWEEN TIMESTAMP
                      '2003-01-01 00:00:00.000000+00:00'
                      AND    TIMESTAMP
                      '2009-12-31 23:59:59.999999+00:00'
                      EACH INTERVAL '1' MONTH) );
```

Example: Changing a CASE_N-Based Row Partitioning Expression

You cannot use the ADD and DROP options to modify a partitioning expression for a table when the expression is based on a CASE_N function. For example, create the following table.

```
CREATE SET TABLE hyp_prd_tbls.dps_sum, NO FALLBACK,
NO BEFORE JOURNAL, NO AFTER JOURNAL,CHECKSUM = DEFAULT (
hyp_month          INTEGER NOT NULL,
hyp_year           INTEGER NOT NULL,
churn_ctgry VARCHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC
NOT NULL,
scenario VARCHAR(6) CHARACTER SET LATIN NOT CASESPECIFIC
NOT NULL,
in_contract_term   VARCHAR(30) CHARACTER SET LATIN
NOT CASESPECIFIC NOT NULL,
promo_type VARCHAR(50) CHARACTER SET LATIN NOT CASESPECIFIC
NOT NULL,
access_tier_lvl1   VARCHAR(50) CHARACTER SET LATIN
NOT CASESPECIFIC NOT NULL,
arpu_tier_lvl1     VARCHAR(50) CHARACTER SET LATIN
NOT CASESPECIFIC NOT NULL,
hrchy_segmt        VARCHAR(50) CHARACTER SET LATIN
NOT CASESPECIFIC NOT NULL,
channel_type       VARCHAR(40) CHARACTER SET LATIN
```

```

        ps_mkt_cd          NOT CASESPECIFIC NOT NULL,
                           VARCHAR(6) CHARACTER SET LATIN
        tenure            NOT CASESPECIFIC NOT NULL,
                           VARCHAR(12) CHARACTER SET LATIN
                           NOT CASESPECIFIC NOT NULL,
        tot_pplan_access_amt  DECIMAL(15,2),
        tot_feat_access_amt  DECIMAL(15,2),
        tot_occ_amt          DECIMAL(15,2),
        tot_airtime_call_amt DECIMAL(15,2),
        churn_tot_pplan_access_amt DECIMAL(15,2),
        churn_tot_feat_access_amt DECIMAL(15,2),
        churn_tot_occ_amt    DECIMAL(15,2),
        churn_tot_airtime_call_amt DECIMAL(15,2))
PRIMARY INDEX (hyp_month, hyp_year, churn_ctgry, scenario,
in_contract_term, promo_type, access_tier_lvl1,
               arpu_tier_lvl1, hrchy_segmt, channel_type,
PARTITION BY CASE_N(hyp_month = 200901, hyp_month = 200902,
                    hyp_month = 200903, hyp_month = 200904,
                    hyp_month = 200905, hyp_month = 200906,
                    hyp_month = 200907, hyp_month = 200908,
                    hyp_month = 200909, hyp_month = 200910,
                    hyp_month = 200911, hyp_month = 200912,
                    NO CASE);
ps_mkt_cd, tenure)

```

You can only use the DROP and ADD options row partitioning expressions that are based on a RANGE function. To alter a partitioning expression based on a CASE_N function, you must use the PARTITION BY clause.

For example, to add the CASE_N options (*hyp_month* = 201001, *hyp_month* = 201002, *hyp_month* = 201003, *hyp_month* = 201004) to table *hyp_prd_tbls.dps_sum*, you would use the following ALTER TABLE request using the PARTITION BY clause.

```

ALTER TABLE hyp_prd_tbls.dps_sum
PARTITION BY CASE_N(hyp_month = 200901, hyp_month = 200902,
                    hyp_month = 200903, hyp_month = 200904,
                    hyp_month = 200905, hyp_month = 200906,
                    hyp_month = 200907, hyp_month = 200908,
                    hyp_month = 200909, hyp_month = 200910,
                    hyp_month = 200911, hyp_month = 200912,
                    hyp_month = 201001, hyp_month = 201002,
                    hyp_month = 201003, hyp_month = 201004,
                    NO CASE);

```

The table must not contain any rows if you modify the partitioning using this method. You would have to copy the table to a new table definition, for example, using a CREATE TABLE AS ... WITH DATA request or create a new table and use an INSERT ... SELECT or MERGE request to copy the rows from the old table to the new table. See CREATE TABLE (AS Clause) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

Suppose *hyp_prd_tbls_sum* was created as the following column-partitioned table.

```
CREATE SET TABLE hyp_prd_tbls.dps_sum, NO FALLBACK,
      NO BEFORE JOURNAL, NO AFTER JOURNAL, CHECKSUM=DEFAULT (
hyp_month          INTEGER NOT NULL,
hyp_year           INTEGER NOT NULL,
churn_ctgry VARCHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC
                  NOT NULL,
scenario VARCHAR(6) CHARACTER SET LATIN NOT CASESPECIFIC
                  NOT NULL,
in_contract_term  VARCHAR(30) CHARACTER SET LATIN
                  NOT CASESPECIFIC NOT NULL,
promo_type VARCHAR(50) CHARACTER SET LATIN NOT CASESPECIFIC
                  NOT NULL,
access_tier_lvl1  VARCHAR(50) CHARACTER SET LATIN
                  NOT CASESPECIFIC NOT NULL,
arpu_tier_lvl1    VARCHAR(50) CHARACTER SET LATIN
                  NOT CASESPECIFIC NOT NULL,
hrchy_segmt       VARCHAR(50) CHARACTER SET LATIN
                  NOT CASESPECIFIC NOT NULL,
channel_type      VARCHAR(40) CHARACTER SET LATIN
                  NOT CASESPECIFIC NOT NULL,
ps_mkt_cd         VARCHAR(6) CHARACTER SET LATIN
                  NOT CASESPECIFIC NOT NULL,
tenure            VARCHAR(12) CHARACTER SET LATIN
                  NOT CASESPECIFIC NOT NULL,
tot_pplan_access_amt DECIMAL(15,2),
tot_feat_access_amt DECIMAL(15,2),
tot_occ_amt        DECIMAL(15,2),
tot_airtime_call_amt DECIMAL(15,2),
churn_tot_pplan_access_amt DECIMAL(15,2),
churn_tot_feat_access_amt DECIMAL(15,2),
churn_tot_occ_amt   DECIMAL(15,2),
churn_tot_airtime_call_amt DECIMAL(15,2))
NO PRIMARY INDEX
PARTITION BY (COLUMN,
              CASE_N(hyp_month = 200801, hyp_month = 200802,
                    hyp_month = 200803, hyp_month = 200804,
```

```

hyp_month = 200805, hyp_month = 200806,
hyp_month = 200807, hyp_month = 200808,
hyp_month = 200809, hyp_month = 200810,
hyp_month = 200811, hyp_month = 200812,
NO CASE);

```

The same rules for using the DROP and ADD options for row partitioning based on a RANGE_N or CASE_N function still apply.

For example, to modify the existing row partitioning expression in a column-partitioned version of *hyp_prd_tbls.dps_sum* with the second level row partitioning based on a CASE_N function to add the CASE_N partitioning (*hyp_month* = 201001, *hyp_month* = 201002, *hyp_month* = 201003, *hyp_month* = 201004) to table *hyp_prd_tbls.dps_sum*, you would use the following ALTER TABLE request.

```

ALTER TABLE hyp_prd_tbls.dps_sum
MODIFY
PARTITION BY (COLUMN,
CASE_N(hyp_month = 200901, hyp_month = 200902,
hyp_month = 200903, hyp_month = 200904,
hyp_month = 200905, hyp_month = 200906,
hyp_month = 200907, hyp_month = 200908,
hyp_month = 200909, hyp_month = 200910,
hyp_month = 200911, hyp_month = 200912,
hyp_month = 201001, hyp_month = 201002,
hyp_month = 201003, hyp_month = 201004,
NO CASE));

```

Example: Adding a Name to a Previously Unnamed Primary Index

To name the primary index of the *orders* table defined in [Table Definitions for Examples](#), you could perform the following request:

```

ALTER TABLE orders
MODIFY PRIMARY INDEX ordpi;

```

Example: Modifying Partitioning for Multilevel Partitioned Tables

Assume you have created a table named *orders* using the following CREATE TABLE request.

```

CREATE TABLE orders (
o_orderkey    INTEGER NOT NULL,
o_custkey     INTEGER,

```

```

    o_orderstatus CHARACTER(1) CASESPECIFIC,
    o_totalprice  DECIMAL(13,2) NOT NULL,
    o_orderdate   DATE FORMAT 'yyyy-mm-dd' NOT NULL)
PRIMARY INDEX (o_orderkey)
PARTITION BY (
RANGE_N(o_custkey   BETWEEN 0
                AND 49999
                EACH 100),
RANGE_N(o_orderdate BETWEEN DATE '2000-01-01'
                AND   DATE '2006-12-31'
                EACH INTERVAL '1' MONTH))
UNIQUE INDEX (o_orderkey);

```

This table has 2-byte partitioning. If *orders* is empty, you could submit the following ALTER TABLE MODIFY request to modify it to have a single-level row-partitioned primary index:

```

ALTER TABLE orders
MODIFY
PARTITION BY RANGE_N(
    o_orderdate BETWEEN DATE '2000-01-01'
                AND   DATE '2006-12-31'
                EACH INTERVAL '1' MONTH);

```

Assume the previously defined CREATE TABLE definition for *orders*. If there are no rows with *o_orderdate* between the dates 2000-01-01 and 2000-12-31, you could submit any one of the following equivalent ALTER TABLE requests to modify the partitioning expression for level 2.

```

ALTER TABLE orders
MODIFY
DROP RANGE#L2   BETWEEN DATE '2000-01-01'
                AND   DATE '2000-12-31'
ADD RANGE      BETWEEN DATE '2007-01-01'
                AND   DATE '2007-12-31'
                EACH INTERVAL '1' MONTH;

```

or

```

ALTER TABLE orders
MODIFY
DROP RANGE#L2
WHERE orders.PARTITION#L2 BETWEEN 1
                AND   12
ADD RANGE#L2      BETWEEN DATE '2007-01-01'

```



```

AND      DATE '2007-12-31'
EACH INTERVAL '1' MONTH;

```

or

```

ALTER TABLE orders
  MODIFY
  DROP RANGE#L2
  WHERE PARTITION#L2 IN (1,2,3,4,5,6,7,8,9,10,11,12)
  ADD RANGE BETWEEN DATE '2007-01-01'
                AND      DATE '2007-12-31'
                EACH INTERVAL '1' MONTH;

```

Assume the previously defined CREATE TABLE definition for *orders*. If there are zero, one, or more rows with *o_orderdate* between 2000-01-01 and 2000-12-31, you could submit the following ALTER TABLE MODIFY request to alter the row partitioning. In this case, the system deletes the rows from the dropped ranges.

```

ALTER TABLE orders
  MODIFY
  DROP RANGE#L2  BETWEEN DATE '2000-01-01'
                  AND      DATE '2000-12-31'
  ADD RANGE      BETWEEN DATE '2007-01-01'
                  AND      DATE '2007-12-31'
                  EACH INTERVAL '1' MONTH
  WITH DELETE;

```

Assume you created one table using the previously defined CREATE TABLE definition for *orders* and another table using the following CREATE TABLE request.

```

CREATE TABLE old_orders (
  o_orderkey    INTEGER NOT NULL,
  o_custkey     INTEGER,
  o_orderstatus CHARACTER(1) CASESPECIFIC,
  o_totalprice  DECIMAL(13,2) NOT NULL,
  o_orderdate   DATE FORMAT 'yyyy-mm-dd' NOT NULL)
  UNIQUE PRIMARY INDEX (o_orderkey);

```

The following ALTER TABLE MODIFY request is one way to alter the partitioning if there are zero, one, or more rows with *o_orderdate* between 2000-01-01 and 2000-12-31. In this case, the system saves the rows in *old_orders* prior to deleting them from *orders*.

```

ALTER TABLE orders
  MODIFY
    DROP RANGE#L2 BETWEEN DATE '2000-01-01'
                      AND     DATE '2000-12-31'
    ADD RANGE#L2  BETWEEN DATE '2007-01-01'
                      AND     DATE '2007-12-31'
                      EACH INTERVAL '1' MONTH
  WITH INSERT INTO old_orders;

```

The following ALTER TABLE MODIFY request is not valid for a populated table because adding more partitions to level 2 exceeds the maximum defined for level 2. This is because there was no ADD option for level 2, and all the excess combined partitions were assigned to level 1 as its default ADD option.

```

ALTER TABLE orders
  MODIFY
    DROP RANGE#L2 BETWEEN DATE '2000-01-01'
                      AND     DATE '2000-12-31'
    ADD RANGE      BETWEEN DATE '2007-01-01'
                      AND     DATE '2008-12-31'
                      EACH INTERVAL '1' MONTH
  WITH DELETE;

```

Assume the previously defined CREATE TABLE definition for *orders_cp*. You could submit the following ALTER TABLE MODIFY request to alter the row partitioning expressions at both levels:

```

ALTER TABLE orders_cp
  MODIFY PRIMARY INDEX
    DROP RANGE BETWEEN      0
                      AND    99
    ADD RANGE  BETWEEN 50000
                      AND    50199
                      EACH 100,
    DROP RANGE WHERE PARTITION#L2 = 1
  WITH DELETE;

```

The number of partitions at level 1 is increased by 2 and the number of partitions at level 2 decreases by 1. At the same time, the ADD value for level 2 increases from 0 to 1.

Example: Modifying Row Partitioning for Multilevel Column-Partitioned Tables

Assume you create *orders_cp*, a column-partitioned version of the *orders* table that has column partitioning at level 1.

If *orders_cp* is not populated with data, you could submit the following ALTER TABLE MODIFY request to modify it to have only one level of row partitioning.

```
ALTER TABLE orders_cp
  MODIFY
    PARTITION BY (COLUMN,
                  RANGE_N(o_orderdate BETWEEN DATE '2000-01-01'
                        AND DATE '2006-12-31'
                        EACH INTERVAL '1' MONTH));
```

Table *orders_cp* has 2-byte partitioning.

If there are no rows with *o_orderdate* in *orders_cp* between the dates 2000-01-01 and 2000-12-31, you could submit any one of the following equivalent ALTER TABLE MODIFY requests to modify the partitioning expression for level 2.

```
ALTER TABLE orders_cp
  MODIFY
    DROP RANGE#L2 BETWEEN DATE '2000-01-01'
                  AND DATE '2000-12-31'
    ADD RANGE#L2 BETWEEN DATE '2007-01-01'
                  AND DATE '2011-12-31'
                  EACH INTERVAL '1' MONTH;
```

or

```
ALTER TABLE orders_cp
  MODIFY
    DROP RANGE
    WHERE orders.cp.PARTITION#L2 BETWEEN 1
                  AND 12
    ADD RANGE#L2 BETWEEN DATE '2007-01-01'
                  AND DATE '2011-12-31'
                  EACH INTERVAL '1' MONTH;
```

or

```

ALTER TABLE orders_cp
  MODIFY
  DROP RANGE#L2
  WHERE PARTITION#L2 IN (1,2,3,4,5,6,7,8,9,10,11,12)
  ADD RANGE BETWEEN DATE '2007-01-01'
                AND    DATE '2011-12-31'
                EACH INTERVAL '1' MONTH;

```

If there are 0, 1, or more rows with *o_orderdate* between 2000-01-01 and 2000-12-31 in *orders_cp*, you could submit the following ALTER TABLE request to alter the row partitioning level of *orders_cp*. In this case, Vantage deletes the rows in the dropped ranges because of the WITH DELETE specification.

This ALTER TABLE MODIFY request is valid for a populated table because adding more partitions to level 2 does not exceed the maximum defined for level 2. This is because all the excess combined partitions were assigned to level 2 as its default ADD option.

```

ALTER TABLE orders_cp
  MODIFY
  DROP RANGE#L2 BETWEEN DATE '2000-01-01'
                AND    DATE '2000-12-31'
  ADD RANGE BETWEEN DATE '2007-01-01'
                AND    DATE '2011-12-31'
                EACH INTERVAL '1' MONTH
  WITH DELETE;

```

Assume you create 1 table using the previously defined *orders_cp* and another table called *old_orders*.

```

CREATE TABLE old_orders (
  o_orderkey    INTEGER NOT NULL,
  o_custkey     INTEGER,
  o_orderstatus CHARACTER(1) CASESPECIFIC,
  o_totalprice  DECIMAL(13,2) NOT NULL,
  o_orderdate   DATE FORMAT 'yyyy-mm-dd' NOT NULL)
  UNIQUE PRIMARY INDEX (o_orderkey);

```

The following ALTER TABLE request is one way to alter the row partitioning if there are 0, 1, or more rows with *o_orderdate* between 2000-01-01 and 2000-12-31. In this case, the system saves the rows in *old_orders* prior to deleting them from *orders_cp*.

```

ALTER TABLE orders_cp
  MODIFY
  DROP RANGE BETWEEN DATE '2000-01-01'
                AND    DATE '2000-12-31'
  ADD RANGE BETWEEN DATE '2007-01-01'

```

```

AND      DATE '2011-12-31'
EACH INTERVAL '1' MONTH
WITH INSERT INTO old_orders;

```

Assume the previously defined CREATE TABLE definition for *orders* (see [Example: Modifying Partitioning for Multilevel Partitioned Tables](#)) with column partitioning added at level one and the primary index has been removed. The new table is named *orders_cp*. Table *orders_cp* has 8-byte partitioning. You could submit the following ALTER TABLE request to alter the partitioning expressions at both levels:

```

ALTER TABLE orders_cp
MODIFY
DROP RANGE BETWEEN      0
AND      399
ADD RANGE BETWEEN 50000
AND      50899
EACH 100,
DROP RANGE WHERE PARTITION#L3 = 1
ADD RANGE BETWEEN DATE '2007-01-01'
AND      DATE '2007-01-31'
WITH DELETE;

```

The number of partitions at level 2 is increased by 5 and the number of partitions at level 3 must remain the same because the default for level 3 is ADD 0, and the excess combined combinations are assigned to level 2, not leaving enough for level 3 to be able to increase its number of partitions.

Example: Modifying Character Partitioning for a Row-Partitioned Table

Assume you have created the following table:

```

CREATE TABLE orders (
  o_orderkey INTEGER NOT NULL,
  o_custkey INTEGER,
  o_orderstatus CHAR(1) NOT CASESPECIFIC,
  o_totalprice DECIMAL(13,2) NOT NULL,
  o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHAR(21),
  o_comment VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY (RANGE_N(o_custkey BETWEEN 0
AND 49999
EACH 1000),
RANGE_N(o_orderdate BETWEEN DATE '2000-01-01'
AND DATE '2006-12-31'

```

```

                                EACH INTERVAL '1' MONTH) ADD 24,
RANGE_N(o_orderpriority BETWEEN 'high'
                                AND    'highest',
                                'low'
                                AND    'lowest',
                                'medium'
                                AND    'medium',
                                NO RANGE OR UNKNOWN))
UNIQUE INDEX (o_orderkey);

```

The following ALTER TABLE request is valid only if *orders* is empty:

```

ALTER TABLE orders
MODIFY
PARTITION BY (RANGE_N(o_custkey  BETWEEN 0
                                AND 49999
                                EACH 1000),
RANGE_N(o_orderdate BETWEEN DATE '2000-01-01'
                                AND    DATE '2006-12-31'
                                EACH INTERVAL '1' MONTH ),
CASE_N(o_orderstatus = 'S',
        o_orderstatus = 'C',
        o_orderstatus = 'F',
        o_orderstatus = 'T',
        NO CASE OR UNKNOWN) );

```

Assuming the preceding table definition for *orders* , you could submit any one of the following equivalent requests if there are no rows with an *o_orderdate* value between 2000-01-01 and 2000-12-31 to alter the row partitioning expression for level 2:

```

ALTER TABLE orders
MODIFY
DROP RANGE#L2 BETWEEN DATE '2000-01-01'
                        AND    DATE '2000-12-31'
ADD RANGE BETWEEN DATE '2007-01-01'
                        AND    DATE '2007-12-31'
                        EACH INTERVAL '1' MONTH;
ALTER TABLE orders
MODIFY
DROP RANGE#L2 WHERE orders.PARTITION#L2 BETWEEN 1
                                                AND    12
ADD RANGE#L2 BETWEEN DATE '2007-01-01'
                        AND    DATE '2007-12-31'

```

```

        EACH INTERVAL '1' MONTH;
ALTER TABLE orders
MODIFY
DROP RANGE#L2 WHERE PARTITION#L2 IN (1,2,3,4,5,6,7,8,9,10,11,12)
ADD RANGE BETWEEN DATE '2007-01-01'
        AND      DATE '2007-12-31'
        EACH INTERVAL '1' MONTH;

```

Assuming the table definition for *orders* , the following ALTER TABLE request is valid even if the table is populated with rows. However, the range being dropped must be empty because the request does not specify a WITH clause.

```

ALTER TABLE orders
MODIFY PRIMARY INDEX
DROP RANGE#L1 BETWEEN 0
        AND      999
ADD RANGE BETWEEN 50000
        AND      50999;

```

These ALTER TABLE MODIFY ADD/DROP RANGE requests only execute if the session collation matches the PPI collation and the session mode matches the session mode in effect when the character PPI was created.

The following example, which is based on the same original schema, returns an error:

```

ALTER TABLE orders
MODIFY
DROP RANGE#L3 BETWEEN 'medium'
        AND      'medium'
ADD RANGE BETWEEN      'medium'
        AND      'urgent';

```

The following ADD/DROP request involving the character row partitioning level of the *orders* table is valid because it involves only the special row partitions NO RANGE OR UNKNOWN and UNKNOWN, with the NO RANGE OR UNKNOWN row partition being replaced with the UNKNOWN partition.

```

ALTER TABLE orders
MODIFY PRIMARY INDEX
DROP RANGE#L3 WHERE PARTITION#L3 = 4
ADD RANGE#L3 UNKNOWN;

```

Example: Modifying Character Partitioning for a Column-Partitioned Table

Assume you have created a column-partitioned version of the *orders* table from the previous example named *orders_cp* with column partitioning at level 1.

The following ALTER TABLE request is valid only if *orders_cp* is not populated with data.

```
ALTER TABLE orders_cp
MODIFY
PARTITION BY (COLUMN,
              RANGE_N(o_custkey    BETWEEN 0
                      AND 49999
                      EACH 1000),
              RANGE_N(o_orderdate BETWEEN DATE '2000-01-01'
                      AND   DATE '2006-12-31'
                      EACH INTERVAL '1' MONTH) ADD 48,
              RANGE_N(o_orderstatus = 'S',
                      o_orderstatus = 'C',
                      o_orderstatus = 'F',
                      o_orderstatus = 'T',
                      NO CASE OR UNKNOWN) );
```

Assuming the previous request succeeded because *orders_cp* was empty, you could submit any one of the following equivalent requests if there are no rows with an *o_orderdate* value between 2000-01-01 and 2000-12-31 to alter the partitioning expression for level 3.

```
ALTER TABLE orders_cp
MODIFY
DROP RANGE#L3 BETWEEN DATE '2000-01-01'
                        AND   DATE '2000-12-31'
ADD RANGE BETWEEN DATE '2007-01-01'
                AND   DATE '2008-12-31'
                EACH INTERVAL '1' MONTH;
ALTER TABLE orders_cp
MODIFY
DROP RANGE#L3 WHERE orders_cp.PARTITION#L2 BETWEEN 1
                                                AND   12
ADD RANGE#L3 BETWEEN DATE '2007-01-01'
                AND   DATE '2008-12-31'
                EACH INTERVAL '1' MONTH;
ALTER TABLE orders_cp
MODIFY
```



```

DROP RANGE#L3 WHERE PARTITION#L3 IN (1,2,3,4,5,6,7,8,9,10,11,12)
ADD RANGE BETWEEN DATE '2007-01-01'
                AND    DATE '2008-12-31'
                EACH INTERVAL '1' MONTH;

```

Assuming that *orders_cp* is column-partitioned, the following ALTER TABLE MODIFY request is valid even if the table is populated with data.

```

ALTER TABLE orders_cp
MODIFY
DROP RANGE#L2 BETWEEN 0
                AND    999
ADD RANGE BETWEEN 50000
                AND    54999
                EACH    100;

```

These ALTER TABLE MODIFY ADD/DROP RANGE requests only execute if the session collation and the session mode are the same as when the table was created.

The following example, which is based on the same original schema, returns an error if either the session collation or session mode are different than when *orders_cp* was created.

```

ALTER TABLE orders_cp
MODIFY
DROP RANGE#L3 BETWEEN 'medium'
                AND    'medium'
ADD RANGE BETWEEN    'medium'
                AND    'urgent';

```

The following ADD/DROP request involving the character partitioning level of the *orders_cp* table is valid because it involves only the special row partitions NO RANGE OR UNKNOWN and UNKNOWN, with the NO RANGE OR UNKNOWN row partition being replaced with the UNKNOWN partition. Vantage deletes any rows that do not belong to any of the defined ranges for *orders_cp*

```

ALTER TABLE orders_cp
MODIFY
DROP RANGE#L4 WHERE PARTITION#L3 = 4
ADD RANGE#L4 UNKNOWN
WITH DELETE;

```

Example: Dropping and Adding Partition Ranges

The following ALTER TABLE request modifies the tables *orders* and *orders_cp* created in [Table Definitions for Examples](#). The outcome of the requests are identical to the table alterations performed in later ALTER TABLE requests in this example, but using different syntax.

The request is valid if there are no rows with values for *o_orderdate* between January 1, 1992 (DATE '1992-01-01') and December 31, 1992 (DATE '1992-12-31').

```
ALTER TABLE orders
MODIFY
  DROP RANGE BETWEEN DATE '1992-01-01'
                        AND   DATE '1992-12-31'
  ADD  RANGE BETWEEN DATE '1999-01-01'
                        AND   DATE '2000-12-31'
      EACH INTERVAL '1' MONTH;
```

The following request modifies the partitioning for the column-partitioned table *orders_cp*.

```
ALTER TABLE orders_cp
MODIFY
  DROP RANGE BETWEEN DATE '1992-01-01'
                        AND   DATE '1992-12-31'
  ADD  RANGE BETWEEN DATE '1999-01-01'
                        AND   DATE '2000-12-31'
      EACH INTERVAL '1' MONTH;
```

The following ALTER TABLE request modifies the table created in [Table Definitions for Examples](#). The result is equivalent to the outcome of other requests in this example set using different syntax, and is valid if there are no rows with values for *o_orderdate* between January 1 1992 (DATE '1992-01-01') and December 31 1992 (DATE '1992-12-31').

```
ALTER TABLE orders
MODIFY
  DROP RANGE WHERE PARTITION BETWEEN 1
                                    AND   12
  ADD  RANGE BETWEEN DATE '1999-01-01'
                        AND   DATE '2000-12-31'
      EACH INTERVAL '1' MONTH;
```

The following request modifies the partitioning for the column-partitioned table *orders_cp*.

```

ALTER TABLE orders_cp
MODIFY
  DROP RANGE BETWEEN DATE '1992-01-01'
                        AND   DATE '1992-12-31'
  ADD  RANGE BETWEEN DATE '1999-01-01'
                        AND   DATE '2000-12-31'
                        EACH INTERVAL '1' MONTH;

```

The following ALTER TABLE request modifies the table created in [Table Definitions for Examples](#). The result is equivalent to the outcome of other requests in this example set and is valid if there are no rows with values for *o_orderdate* between January 1 1992 (DATE '1992-01-01') and December 31 1992 (DATE '1992-12-31').

```

ALTER TABLE orders
MODIFY
  DROP RANGE WHERE Orders.PARTITION IN (1,2,3,4,5,6,7,8,9,10,11,12)
  ADD  RANGE BETWEEN DATE '1999-01-01'
                        AND   DATE '2000-12-31'
                        EACH INTERVAL '1' MONTH;

```

The following request modifies the partitioning for the column-partitioned table *orders_cp*.

```

ALTER TABLE orders_cp
MODIFY
  DROP RANGE BETWEEN DATE '1992-01-01'
                        AND   DATE '1992-12-31'
                        EACH INTERVAL '1' MONTH
  ADD  RANGE BETWEEN DATE '1999-01-01'
                        AND   DATE '2000-12-31'
                        EACH INTERVAL '1' MONTH;

```

Example: Drop and Add Partition Ranges and Delete Rows Outside the Defined Ranges

The following ALTER TABLE request modifies the table created in [Table Definitions for Examples](#). It is valid if there are 0 or more rows with *o_orderdate* values between January 1 1992 (DATE '1992-01-01') and December 31 1992 (DATE '1992-12-31'). As a result of the WITH DELETE specification, those rows, if any, are deleted from orders because they do not belong to any partition in the new partitioning expression.

The specified primary index is the same as the existing primary index for the table because the request does not include changes to the primary index. The EACH clause after the DROP clause is ignored, but the EACH clause after the ADD clause is processed.

```

ALTER TABLE orders
MODIFY
  DROP RANGE BETWEEN DATE '1992-01-01'
                        AND   DATE '1992-12-31'
                        EACH INTERVAL '1' MONTH
  ADD  RANGE BETWEEN DATE '1999-01-01'
                        AND   DATE '2000-12-31'
                        EACH INTERVAL '1' MONTH
WITH DELETE;

```

The following request modifies the partitioning for the column-partitioned table *orders_cp*.

```

ALTER TABLE orders_cp
MODIFY
  DROP RANGE BETWEEN DATE '1992-01-01'
                        AND   DATE '1992-12-31'
  ADD  RANGE BETWEEN DATE '1999-01-01'
                        AND   DATE '2000-12-31'
                        EACH INTERVAL '1' MONTH
WITH DELETE;

```

Example: Dropping Row Range Partition When Partitioning is Defined with RANGE_N Function and NO RANGE Partition

Suppose you have the following PPI base table definition.

```

CREATE SET TABLE sales_table, NO FALLBACK, CHECKSUM = DEFAULT ,
                        NO BEFORE JOURNAL, NO AFTER JOURNAL (
  product_code  CHARACTER(8) CHARACTER SET LATIN NOT CASESPECIFIC
                        NOT NULL,
  sales_date    DATE FORMAT 'YYYY-MM-DD' NOT NULL NOT NULL,
  agent_id      CHARACTER(8) CHARACTER SET LATIN NOT CASESPECIFIC
                        NOT NULL,
  quantity_sold INTEGER,
  product_desc  VARCHAR(50) CHARACTER SET LATIN NOT CASESPECIFIC)
PRIMARY INDEX (product_code, sales_date, agent_id)
PARTITION BY RANGE_N(sales_date BETWEEN DATE '2001-01-01'
                        AND   DATE '2003-12-31'
                        EACH INTERVAL '1' MONTH ,
                        NO RANGE);

```

You then define the following PPI base table to act as the save table for an ALTER TABLE request in which you will drop one or more row partitions from *sales_table*.

```
CREATE SET TABLE sales_table_1, NO FALLBACK,
  NO BEFORE JOURNAL, NO AFTER JOURNAL, CHECKSUM = DEFAULT (
    product_code  CHARACTER(8) CHARACTER SET LATIN NOT CASESPECIFIC
                      NOT NULL,
    sales_date    DATE FORMAT 'YYYY-MM-DD' NOT NULL,
    agent_id      CHARACTER(8) CHARACTER SET LATIN NOT CASESPECIFIC,
    quantity_sold INTEGER,
    product_desc  VARCHAR(50) CHARACTER SET LATIN NOT CASESPECIFIC
                      NOT NULL)
PRIMARY INDEX (product_code, sales_date, agent_id)
PARTITION BY RANGE_N(sales_date
  BETWEEN DATE '2001-01-01'
  AND      DATE '2001-12-31'
  EACH INTERVAL '1' MONTH , NO RANGE);
```

You then populate *sales_table* with the following rows.

```
INSERT INTO sales_table
  VALUES ('PC2',DATE '2001-01-10','AG2',5,'PC');
INSERT INTO sales_table
  VALUES ('PC3',DATE '2001-03-10','AG2',5,'PC');
INSERT INTO sales_table
  VALUES ('PC4',DATE '2002-05-10','AG2',5,'PC');
INSERT INTO sales_table
  VALUES ('PC5',DATE '2003-07-10','AG2',5,'PC');
INSERT INTO sales_table
  VALUES ('PC5',DATE '2004-07-10','AG2',5,'PC');
```

The following SELECT request indicates that the 5 intended rows were successfully inserted into *sales_table*:

```
SELECT partition, product_code, sales_date, agent_id,
       quantity_sold, product_description
FROM sales_table
ORDER BY 1;
PARTITION product_code sales_date agent_id quantity_sold product_desc
-----
```

1	PC2	2001-01-10	AG2	5	PC
3	PC3	2001-03-10	AG2	5	PC
17	PC4	2002-05-10	AG2	5	PC

31 PC5	2003-07-10 AG2	5 PC
37 PC5	2004-07-10 AG2	5 PC

Use an ALTER TABLE MODIFY request to drop a range of row partitions from *sales_table* with the intent of saving any deleted rows in *sales_table1*, by specifying a WITH INSERT INTO *sales_table1* clause.

```
ALTER TABLE sales_table
MODIFY
DROP RANGE BETWEEN DATE '2001-01-01'
AND DATE '2001-12-31'
WITH INSERT INTO sales_table1;
*** Table has been modified.
*** Total elapsed time was 1 second.
```

Select the contents of *sales_table* to check whether the rows within the row partition you dropped were deleted from the table.

```
SELECT PARTITION, product_code, sales_date, agent_id,
quantity_sold, product_desc
FROM sales_table
ORDER BY 1;
*** Query completed. 5 rows found. 6 columns returned.
*** Total elapsed time was 1 second.
PARTITION product_code sales_date agent_id quantity_sold product_desc
-----
5 PC4 2002-05-10 AG2 5 PC
19 PC5 2003-07-10 AG2 5 PC
25 PC2 2001-01-10 AG2 5 PC<<<<<
25 PC3 2001-03-10 AG2 5 PC<<<<<
25 PC5 2004-07-10 AG2 5 PC<<<<<
```

Because the base table *sales_table* includes a NO RANGE partition, rows are not deleted when you drop the range of row partitions. Those rows are moved to the NO RANGE partition, which is identified by partition number 25, and retained in *sales_table*.

To do what you had intended, you should have either defined *sales_table* without specifying a NO RANGE partition or deleted the rows before submitting the ALTER TABLE request.

Now assume that you create a column-partitioned version of *sales_table*, *sales_table_cp*, in another database, and also assume that you create a column-partitioned version of *sales_table1*, *sales_table1_cp*, in the same database. You submit an ALTER TABLE MODIFY request to drop a range of partitions from *sales_table_cp* with the intent of saving any dropped rows in *sales_table1_cp*, specifying a WITH INSERT INTO *sales_table1_cp* clause to do so.

```

ALTER TABLE sales_table_cp
  MODIFY
    DROP RANGE BETWEEN DATE '2001-01-01'
                      AND   DATE '2001-12-31'
  WITH INSERT INTO sales_table1_cp;
*** Table has been modified.
*** Total elapsed time was 1 second.

```

Submit the same SELECT PARTITION request that you used for the PPI version of *sales_table* (except retrieve the rows from *sales_table1_cp* rather than *sales_table1*) to check whether the rows within the partition that was dropped were deleted from the table, and the request returns the identical result set.

```

SELECT PARTITION, product_code, sales_date, agent_id,
           quantity_sold, product_desc
FROM sales_table_cp
ORDER BY 1;

```

Example: Dropping Ranges Without an EACH Clause

Assume the following table definition.

```

CREATE TABLE t1 (
  i INTEGER,
  d DATE)
PRIMARY INDEX (i)
PARTITION BY RANGE_N(d BETWEEN DATE '2000-01-01'
                     AND   DATE '2000-12-31'
                     EACH INTERVAL '1' MONTH);

```

The following ALTER TABLE request is valid. After the EACH clause is expanded, *t1* has 12 existing ranges of one month each. The DROP RANGE operation is an attempt to drop a single four-month range that covers 4 of the existing one-month ranges. Vantage accepts this as a valid request to drop the 4 ranges within the range specified by the DROP clause.

```

ALTER TABLE t1
  MODIFY
    DROP RANGE BETWEEN DATE '2000-01-01'
                      AND   DATE '2000-04-30'
  WITH DELETE;

```

The new partitioning expression for *t1* after this ALTER TABLE request completes is the following.

```
RANGE_N(d BETWEEN DATE '2000-05-01'
        AND      DATE '2000-12-31'
        EACH INTERVAL '1' MONTH)
```

If table *t1* were column-partitioned and named *t1_cp*, the preceding row-partitioned table example would have the same effect.

Example: Using MODIFY to Repartition a Table and Saving Resulting Nonvalid Rows in a Save Table

Assume that the *orders* table defined in [Table Definitions for Examples](#) exists and the following save table has been created to handle rows that are no longer valid when you change the row partitioning for the *orders* table.

```
CREATE TABLE old_orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk         CHARACTER(16),
  o_shippriority  INTEGER,
  o_comment       VARCHAR(79))
UNIQUE PRIMARY INDEX (o_orderkey);
```

The following ALTER TABLE request is valid if there are 0 or more rows with *o_orderdate* between January 1, 1992 and December 31, 1992 (DATE '1992-01-01 AND DATE '1992-12-31').

The MODIFY option moves any rows having those values into the *old_orders* save table and then deleted from *orders*.

```
ALTER TABLE orders
MODIFY
  DROP RANGE BETWEEN DATE '1992-01-01'
                    AND      DATE '1992-12-31'
  ADD  RANGE BETWEEN DATE '1999-01-01'
                    AND      DATE '2000-12-31'
                    EACH INTERVAL '1' MONTH
WITH INSERT INTO old_orders;
```

Suppose *orders* had been defined as a column-partitioned table, *orders_cp*.

The following ALTER TABLE request is valid if there are 0 or more rows with *o_orderdate* between January 1, 1992 and December 31, 1992 (DATE '1992-01-01 AND DATE '1992-12-31').

The MODIFY option moves any rows having those values into the *old_orders* save table and then deleted from *orders_cp*.

```
ALTER TABLE orders_cp
MODIFY
  DROP RANGE BETWEEN DATE '1992-01-01'
                        AND   DATE '1992-12-31'
  ADD  RANGE BETWEEN DATE '1999-01-01'
                        AND   DATE '2000-12-31'
                        EACH INTERVAL '1' MONTH
WITH INSERT INTO old_orders;
```

Example: Revalidating the Partitioning for a Table

Suppose that one of the following events causes you to suspect that incorrect partitioning of the *orders* table has occurred.

- You suspect that partitioning might not be correct after a restore.
- You copy the table to a system with different hardware or operating system.
- A system malfunction has occurred.
- The data dictionary has not been updated for tables created in an earlier release.

You can validate the partitioning of *orders* table rows using either of the following ALTER TABLE requests, with the first request deleting any problematic rows and the second moving them into a new table.

```
ALTER TABLE orders
REVALIDATE WITH DELETE;

ALTER TABLE orders
REVALIDATE WITH INSERT INTO old_orders;
```

Now assume that *orders* is column-partitioned and is named *orders_cp*. The following ALTER TABLE requests validate the partitioning of *orders_cp* table rows in the same way that they were validated in the previous part of this example.

```
ALTER TABLE orders_cp
REVALIDATE WITH DELETE;

ALTER TABLE orders_cp
REVALIDATE WITH INSERT INTO old_orders;
```

Examples: USING FAST MODE

Adding Columns with USING FAST MODE ON

Adds columns custid and custaddr to existing, populated table sales_hist, converting sales_hist to FCA format if it is in non-FCA format.

If DBSControl parameters FastAlterEnable and FastAlterDefault are enabled (the default), you can omit USING FAST MODE ON.

```
ALTER TABLE sales_hist
USING FAST MODE ON
ADD custid INT DEFAULT 9999,
ADD custaddr VARCHAR(1000);
```

Adding Columns with USING FAST MODE OFF

Adds columns custid and custaddr to existing, populated table sales_hist, converting sales_hist to non-FCA format if it is in FCA format.

```
ALTER TABLE sales_hist
USING FAST MODE OFF
ADD custid INT DEFAULT 9999,
ADD custaddr VARCHAR(1000);
```

Converting Table from FCA Format to Non-FCA Format without Adding Columns

```
ALTER TABLE sales_hist USING FAST MODE OFF;
```

Related Information

For more information about table definitions and primary index partitioning, see [CREATE TABLE and CREATE TABLE AS](#) and [CREATE TABLE \(Queue Table Form\)](#) and *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

You can use the ALTER TABLE TO CURRENT statement to reconcile the partitioning of a partitioned primary index for a temporal table or uncompressed join index defined on a temporal table. See [ALTER TABLE TO CURRENT](#).

For more information about partitioned primary indexes and the system-derived PARTITION and PARTITION#Ln columns, see *Teradata Vantage™ - Database Design*, B035-1094.

For information about the various forms of partition elimination, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

For information about using ALTER TABLE to modify temporal tables, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

For information about Teradata Unity, see the Teradata Unity documentation.

ALTER TABLE (Map and Colocation Form)

Moves a table from one map to another. You can also change the colocation name for a sparse map.

Tables that have the same sparse map and colocation name can be joined using the same primary index or primary AMP index columns without redistributing or duplicating rows. For example, you can colocate two tables, then join the tables on the primary index or primary AMP index columns.

You can also use SQL procedures provided in the TDMaps system database to move tables from one map to another. See *Teradata Vantage™ - Database Administration*, B035-1093.

Required Privileges

You must have the DROP TABLE privilege for the table and you must have been granted the specified map, except when the map you specify is the current map for the table or you specify the same map determined to be the default map according to the following order of precedence:

- If the immediate owner is not the creator:
 - Default map, if defined, for the profile of the immediate owner.
 - Default map, if defined, for the immediate owner.
 - System-default map.
- Default map, if defined, for the profile of the creator.
- Default map, if defined, for the creator.
- System-default map.

See the CREATE USER [DEFAULT MAP](#) option, CREATE DATABASE [DEFAULT MAP](#) option, or CREATE PROFILE [DEFAULT MAP](#) option.

See GRANT MAP in *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

ALTER TABLE Syntax (Map and Colocation Form)

```
ALTER TABLE [ database_name. | user_name. ] table_name ,
  MAP = map_name [ COLOCATE USING colocation_name ] [ ; ]
```

ALTER TABLE Syntax Elements (Map and Colocation Form)

map_name

Name of an existing contiguous or sparse map.

You cannot specify TD_DataDictionaryMap or TD_GlobalMap.

COLOCATE USING *colocation_name*

Name for colocating the table on the same AMPs with other tables, join indexes, or hash indexes. For example, you can colocate two tables, then join the tables on the primary index or primary AMP index columns.

You cannot specify this option if you are altering the table to use a contiguous map. For a contiguous map, a colocation name is not needed for colocation and *colocation_name* is set to NULL.

If you do not want to colocate the table, do not specify this option.

Usage Notes

Altering the Table to a Sparse Map Without Specifying the Colocation Option

If you alter the table to use a sparse map without specifying the COLOCATE USING clause and the table previously used a sparse map, the colocation name does not change. Otherwise, the colocation name defaults to *database_table*, where *database* is the name of the database or user followed by an underscore (_) and *table* is the name of the table. If *database* exceeds 63 characters, *database* is truncated to 63 characters. If *table* exceeds 64 characters, *table* is truncated to 64 characters.

Locks During the Alter Table Map Operation

A read lock is placed on the table during the copy to the new map. The lock is upgraded to an exclusive lock while DBC.TVM is updated and the transaction is committed.

Indexes and the Alter Table Map Operation

Secondary indexes for the table are moved or rebuilt using the new map.

Join indexes or hash indexes on the base table are not moved when the table is moved to a new map. You must use [ALTER JOIN INDEX](#) or [ALTER HASH INDEX](#), as appropriate.

System-defined join indexes for the table are moved or rebuilt using the new map.

The table can have a primary index, primary AMP index (PA), or no primary index (NoPI). The table can have column partitioning, row partitioning, or both. NoPI and PA table rows are redistributed into the new map.

Join indexes and hash indexes that have the ROWID field from a PA, NoPI table, or from a column-partitioned PI table that is altered to a new map are invalidated. You must drop and recreate the join index or hash index.

Secure Zones and Sparse Maps

For a sparse map, you must be in the same secure zone as the sparse map.

Permanent Journaling and Altering the Table Map

If permanent journaling is enabled for a table, the map for the table must be a contiguous map and the map must be the same as the map of the journal table associated with the table. A journal table must have a contiguous map.

To alter the map of a table that has journaling enabled, you must alter the table to have NO JOURNAL and alter the table to another contiguous map. Then, you can enable journaling again. If you alter a table to a sparse map, you cannot enable journaling.

Examples

Alter a Table to a Sparse Map

In this example, a table is populated with only 5 rows. A sparse map named OneAMPMap has been created. Following is the table definition:

```
CREATE SET TABLE MyDatabase.Tab1, FALLBACK,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT,
  DEFAULTMERGEBLOCKRATIO,
  MAP = TD_Map1
( A1 INTEGER,
  B1 INTEGER,
  C1 INTEGER
) PRIMARY INDEX(A1);
```

This statement moves the table to the sparse map and the colocation name defaults to MyDatabase_Tab1:

```
ALTER TABLE Tab1, MAP=OneAMPMap;
```

Alter Two Tables to a Sparse Map

In this example, two tables are populated with only 5 rows each and often joined on A1=A2. A sparse map named OneAMPMap has been defined. This example uses the following table definitions:

```
CREATE SET TABLE MyDatabase.Tab1, FALLBACK,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT,
  DEFAULTMERGEBLOCKRATIO,
  MAP = TD_Map1
( A1 INTEGER,
  B1 INTEGER,
```

```

    C1 INTEGER
) PRIMARY INDEX(A1);

CREATE SET TABLE MyDatabase.Tab2, FALLBACK,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULTMERGEBLOCKRATIO,
    MAP = TD_Map1
( A2 INTEGER,
  B2 INTEGER,
  C2 INTEGER
) PRIMARY INDEX(A2);

```

Because the tables are joined on the primary index, you want to colocate the tables. For table Tab1, the colocation name defaults to MyDatabase_Tab1. For table Tab2, you set the colocation name to MyDatabase_Tab1. The following statements colocate the tables:

```

ALTER TABLE Tab1, MAP=OneAMPMap;

ALTER TABLE Tab2, MAP=OneAMPMap COLOCATE USING MyDatabase_Tab1;

```

ALTER FOREIGN TABLE

ALTER FOREIGN TABLE can change the schema or attributes of a specified foreign table.

To see the foreign table definition, use [SHOW TABLE](#).

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the DROP TABLE privilege on the foreign table that you want to alter.

ALTER FOREIGN TABLE Syntax

```

ALTER FOREIGN TABLE table_specification
[ , table_option [, ...] ]
[ , external_security_clause ]
[ column_option ]
[ UPDATE update_specification ]

```

table_specification

```
[ database_name. | user_name. ] table_name
```

external_security_clause

```
EXTERNAL SECURITY [ { INVOKER | DEFINER } TRUSTED ]  
[ database_name. | user_name. ] authorization_name
```

column_option

```
{ ADD { column change |  
      (named_column_specification [,...]) INTO column_2 |  
      [ COLUMN | ROW ] (named_column_specification [,...]) [ [NO]  
      AUTO COMPRESS ] |  
      [ COLUMN | ROW | SYSTEM ] (column) [ [NO] AUTO COMPRESS ]  
    } |  
  DROP column [ IDENTITY ] |  
  RENAME column { AS | TO } new_name  
}
```

update_specification

```
{ [ LOCATION (location_column) ]  
  [ PATHPATTERN (path_pattern) ]  
  [ MANIFEST [ ( { 'TRUE' | 'FALSE' } ) ] ]  
  [ ROWFORMAT [ (row_format) ] ]  
  [ STOREDAS [ ( { 'TEXTFILE' | 'PARQUET' } ) ] ]  
  [ STRIP_EXTERIOR_SPACES [ ( { 'TRUE' | 'FALSE' } ) ] ]  
  [ STRIP_ENCLOSING_CHAR [ ( { 'NONE' | '' } ) ] ]  
}
```

change

```
{ nameless_column_specification [ INTO column_2 ] |  
  data_type_attributes |  
  [ CONSTRAINT constraint_name ] column_constraint |  
  NULL |
```

```
security_constraint_column_declaration |
}
```

named_column_specification

```
column data_type_declaration [ data_type_attributes ]
```

nameless_column_specification

```
data_type_declaration [ data_type_attributes ]
```

ALTER FOREIGN TABLE Syntax Elements

table_option

Any table option allowed for CREATE FOREIGN TABLE (see [table_option](#)).

database_name

Name of the database that contains the foreign table.

Default: current database

user_name

Name of the user who owns the foreign table.

Default: current user

table_name

Name of the existing foreign table to alter.

normalize_option

For syntax, see *normalize* in [ALTER TABLE Syntax \(Basic\)](#). For syntax element description, see [NORMALIZE](#).

period_specification

For syntax, see *add_option* in [ALTER TABLE Syntax \(Basic\)](#). For syntax element description, see [ADD PERIOD FOR](#).

column

Name of *table_name* column to alter.

column_2

Name of *table_name* column other than *column*. See INTO *column_name* in [ADD column_name](#).

new_name

New name for *column*.

data_type_declaration

For syntax, see *data_type* in [CREATE GLOBAL TEMPORARY TRACE TABLE Syntax](#).
For syntax element description, see [CREATE GLOBAL TEMPORARY TRACE TABLE Syntax Elements](#).

data_type_attributes

For syntax, see *data_type_attributes* in [CREATE GLOBAL TEMPORARY TRACE TABLE Syntax](#). For syntax element description, see [CREATE GLOBAL TEMPORARY TRACE TABLE Syntax Elements](#).

location_column

See [location_column](#).

path_pattern

See [PATHPATTERN](#).

MANIFEST

Specifies whether *location_column* points to a manifest file. If you specify 'FALSE', *location_column* points to a key prefix. The key prefix must include the full path and file name.

Default: 'FALSE'

row_format

See [ROWFORMAT](#).

You cannot specify ROWFORMAT with STOREDAS ('PARQUET').

STOREDAS

See [STOREDAS](#).

Default: 'TEXTFILE'

STRIP_EXTERIOR_SPACES

[Optional] Specify whether to strip exterior (leading and trailing) spaces.

STRIP_EXTERIOR_SPACES is disallowed with any of the following:

- Payload defined with data type DATASET CSV or JSON
- STOREDAS ('PARQUET')
- PARTITION BY COLUMN

Default: 'FALSE'

STRIP_ENCLOSING_CHAR

[Optional] Specify either 'NONE' or the enclosing character to skip when there are no exterior spaces. The *enclosing_character* must be double quotation mark (").

STRIP_ENCLOSING_CHAR is disallowed with any of the following:

- Payload defined with data type DATASET CSV or JSON
- STOREDAS ('PARQUET')
- PARTITION BY COLUMN

Default: 'NONE'

constraint_name

Name of constraint to add to *column*.

column_constraint

See [column_constraint_attribute](#).

Usage Notes

- You cannot use ALTER FOREIGN TABLE to drop the location or payload column.
- You cannot use ALTER FOREIGN TABLE to change the location column.
- ALTER FOREIGN TABLE enforces the following rules for Parquet foreign columnar tables:
 - No row partitioning (*column_option* cannot specify ROW and *update_specification* cannot specify ROWFORMAT)
 - No subrow partitioning
 - No multicolumn partitions
 - No autocompression (*column_option* cannot specify AUTOCOMPRESS)

Examples

Tables for Examples

The examples use these foreign tables.

JSON

CREATE FOREIGN TABLE Statement

```
CREATE FOREIGN TABLE JSONTab1
USING (LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/JSONDATA/'));
```

SHOW TABLE Statement

```
SHOW TABLE JSONTab1;
```

```
CREATE MULTISET FOREIGN TABLE JSONTab1 ,FALLBACK ,
  MAP = TD_MAP1
  (
    Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
    Payload JSON(16776192) INLINE LENGTH 64000 CHARACTER SET LATIN)
USING
  (
    LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/JSONDATA/')
    MANIFEST ('FALSE')
    PATHPATTERN ('$var1/$var2/$var3/$var4/$var5')
    ROWFORMAT ('{"record_delimiter":"\n","character_set":"LATIN"}')
    STOREDAS ('TEXTFILE')
  )
NO PRIMARY INDEX ;
```

CSV

CREATE FOREIGN TABLE Statement

```
CREATE FOREIGN TABLE CSVTab1
USING (LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/CSVDATA/'));
```

SHOW TABLE Statement

```
SHOW TABLE CSVTab1;
```

```
CREATE MULTISET FOREIGN TABLE CSVTab1 ,FALLBACK ,
  MAP = TD_MAP1
  (
    Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
    GageHeight2 DECIMAL(3,2),
```

```

    Flow DECIMAL(3,2),
    site_no INTEGER,
    datetime TIMESTAMP(0) FORMAT 'Y4-MM-DDBHH:MI',
    Precipitation DECIMAL(3,2),
    GageHeight DECIMAL(3,2))
USING
(
    LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/CSVDATA/')
    MANIFEST ('FALSE')
    PATHPATTERN ('$var1/$site_no/$var3/$var4/$var5')
    ROWFORMAT
('{"field_delimiter":",","record_delimiter":"\n","character_set":"LATIN"}')
    STOREDAS ('TEXTFILE')
    HEADER ('TRUE')
    STRIP_EXTERIOR_SPACES ('FALSE')
    STRIP_ENCLOSING_CHAR ('NONE')
)
NO PRIMARY INDEX ;

```

Parquet

CREATE FOREIGN TABLE Statement

```

CREATE FOREIGN TABLE ParquetTab1 (
    Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
    GageHeight2 FLOAT,
    Flow FLOAT,
    site_no BIGINT,
    datetime VARCHAR(16) CHARACTER SET UNICODE NOT CASESPECIFIC,
    Precipitation FLOAT,
    GageHeight FLOAT
)
USING (LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/PARQUETDATA/'));

```

SHOW TABLE Statement

```
SHOW TABLE ParquetTab1;
```

```

CREATE MULTISET FOREIGN TABLE ParquetTab1 ,FALLBACK ,
    MAP = TD_MAP1
(
    Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
    GageHeight2 FLOAT,

```

```

    Flow FLOAT,
    site_no BIGINT,
    datetime VARCHAR(16) CHARACTER SET UNICODE NOT CASESPECIFIC,
    Precipitation FLOAT,
    GageHeight FLOAT)
USING
(
    LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/PARQUETDATA/')
    MANIFEST ('FALSE')
    PATHPATTERN ('$var1/$var2/$var3/$var4/$var5')
    STOREDAS ('PARQUET')
)
NO PRIMARY INDEX
PARTITION BY COLUMN ADD 65525;

```

Column-Level Alterations

Example: Add Column to Table

```
ALTER FOREIGN TABLE ParquetTab1 ADD GageHeight FLOAT;
```

Example: Change Character Set of Payload Column to UNICODE

```
ALTER FOREIGN TABLE JSONTab1
ADD Payload JSON(8388096) INLINE LENGTH 32000 CHARACTER SET UNICODE;
```

Example: Change Character Set of Payload Column to LATIN

```
ALTER FOREIGN TABLE JSONTab1
ADD Payload JSON(16776192) INLINE LENGTH 64000 CHARACTER SET LATIN;
```

Example: Drop Column from Table

```
ALTER FOREIGN TABLE ParquetTab1 DROP GageHeight;
```

Table-Level Alterations

Example: Update Map

```
ALTER FOREIGN TABLE ParquetTab1, MAP = TD_MAP2;
```

Example: Update Simplified Authorization

```
ALTER FOREIGN TABLE ParquetTab1, EXTERNAL SECURITY s3_auth1;
```

Example: Update Legacy Authorization

```
ALTER FOREIGN TABLE ParquetTab1, EXTERNAL SECURITY DEFINER
TRUSTED TDAWSEDBS_AUTH2;
```

Example: Update FALLBACK Attribute and Authorization

```
ALTER FOREIGN TABLE JSONTab1, FALLBACK,
EXTERNAL SECURITY DEFINER TRUSTED TDAWSEDBS_AUTH2;
```

Example: Update LOCATION

```
ALTER FOREIGN TABLE ParquetTab1
UPDATE
LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/PARQUETDATA/09394500');
```

Example: Update PATHPATTERN and ROWFORMAT

```
ALTER FOREIGN TABLE CSVTab1
UPDATE
PATHPATTERN ('$year/$month/$day')
ROWFORMAT ('{"record_delimiter": "\n", "character_set": "UTF8"}')
;
```

Example: Alter Both Column-Level and Table-Level Attributes

DROP is a column-level alteration; UPDATE is a table-level alteration.

```
ALTER FOREIGN TABLE ParquetTab1
DROP Precipitation
UPDATE LOCATION ('/s3/td-usgs-public.s3.amazonaws.com/PARQUETDATA/09394500')
PATHPATTERN ('$Var1/$Var2/$var3');
```

ALTER TABLE TO CURRENT

Reconciles the row partitioning for a table or uncompressed join index to a newly resolved date or timestamp when its partitioning is based on the DATE, CURRENT_DATE, or CURRENT_TIMESTAMP functions.

Required Privileges

To execute ALTER TABLE TO CURRENT, you must have the DROP TABLE privilege on that table or join index or on its containing database or user.

To execute ALTER TABLE TO CURRENT on a join index, you must also have the INDEX or DROP TABLE privilege on any base table that a CURRENT_DATE or CURRENT_TIMESTAMP condition is defined on.

Privileges Granted Automatically

None.

ALTER TABLE TO CURRENT Syntax

```
ALTER TABLE { table_name | join_index_name } TO CURRENT
[ WITH [ INSERT [INTO] save_table | DELETE ] ] [;]
```

ALTER TABLE TO CURRENT Syntax Elements

table_name

Name of the table whose row partitioning is to be reconciled to a new current date or timestamp value.

join_index_name

Name of the join index whose row partitioning is to be reconciled to a new current date or timestamp value.

WITH INSERT INTO *save_table*

Table into which to insert any row whose row partition number evaluates to a value outside the valid range.

The WITH INSERT INTO *save_table* clause is sometimes referred to as a null partition handler. You cannot specify a null partition handler for a join index.

The *save_table* and the table being altered must be different tables with different names or they must be in different databases.

WITH DELETE

Delete any row whose row partition number evaluates to null or to a value outside the valid range.

The WITH DELETE clause is sometimes referred to as a null partition handler. You cannot specify a null partition handler for a join index.

Usage Notes

ALTER TABLE TO CURRENT Processing

During ALTER TABLE TO CURRENT processing, if the starting expression with the newly resolved DATE, CURRENT_DATE, or CURRENT_TIMESTAMP value does not fall on a partition boundary, Vantage must scan all of the rows before repartitioning the rows based on the new partitioning expression. Column partitioned tables must be empty before using ALTER TABLE TO CURRENT.

For an ALTER TABLE TO CURRENT operation where an all-row scan is unnecessary, the table can contain data.

ALTER TABLE TO CURRENT Examples

Example: Altering Table Row Partitioning with Partitioning Expression Based on Multiple CURRENT_DATE Functions

Assume that the table is partitioned with the latest 2 quarters in separate row partitions and all other data in another row partition. The example below reconciles CURRENT_DATE at the beginning of each quarter. Assume that the table is created on January 1, 2009; therefore, CURRENT_DATE is resolved to January 1, 2009 at creation time.

```
CREATE SET TABLE sales, NO FALLBACK (
  storeID    INTEGER,
  amount     DECIMAL(10,2),
  sale_date  DATE FORMAT 'YYYY/MM/DD' NOT NULL)
PRIMARY INDEX (storeID)
PARTITION BY CASE_N(sale_date>=CURRENT_DATE/*latest quarter data*/,
  sale_date<CURRENT_DATE
  AND sale_date>=CURRENT_DATE-INTERVAL '3' MONTH,
  NO CASE);
```

The table contains the following rows with a resolved CURRENT_DATE of January 1, 2009:

sales			
StoreID	Amount	Sale_Date	PARTITION
1	1000.00	2009-01-31	1
1	2000.00	2009-01-01	1
1	3500.00	2009-01-15	1
1	500.00	2008-09-15	2

sales			
StoreID	Amount	Sale_Date	PARTITION
1	2000.00	2008-12-15	2
1	5000.00	2009-04-01	3

Assume that you submit the following request on April 1, 2009:

```
ALTER TABLE sales TO CURRENT;
```

The resolved CURRENT_DATE is changed to April 1, 2009 and the rows in the table are reconciled as follows.

sales			
StoreID	Amount	Sale_Date	PARTITION
1	1000.00	2009-01-31	2
1	2000.00	2009-01-01	2
1	3500.00	2009-01-15	2
1	500.00	2008-09-15	3
1	2000.00	2008-12-15	3
1	5000.00	2009-04-01	3

Suppose you submit the following CREATE TABLE request on January 1, 2009.

```
CREATE SET TABLE customer, NO FALLBACK (
  cust_name          CHARACTER(10),
  cust_no            INTEGER,
  policy_expiration_date DATE FORMAT 'YYYY/MM/DD' NOT NULL)
PRIMARY INDEX (cust_no)
PARTITION BY CASE_N(policy_expiration_date>=CURRENT_DATE,
                     policy_expiration_date<CURRENT_DATE
                     AND policy_expiration_date>=CURRENT_DATE-
                           INTERVAL '3' MONTH);
```

The rows in the table are as follows, with the resolved CURRENT_DATE as January 1, 2009.

customer			
cust_name	cust_no	policy_expiration_date	PARTITION
Li	1	2009-01-31	1
Khan	2	2009-01-01	1
Reddy	3	2009-01-15	1
Smith	5	2008-12-15	2

You submit the following ALTER TABLE ... TO CURRENT request on April 1, 2009.

```
ALTER TABLE customer TO CURRENT;
```

Because the following row cannot be moved to any row partition based on the revised CURRENT_DATE value, the system returns an error message to the requestor.

customer			
cust_name	cust_no	policy_expiration_date	PARTITION
Smith	5	2008-12-15	2

To avoid this error, you instead submit the following ALTER TABLE ... TO CURRENT request on April 1, 2009:

```
ALTER TABLE customer TO CURRENT WITH DELETE;
```

As a result of this ALTER TABLE ... TO CURRENT request, Vantage deletes the following row from the table because it is no longer of interest.

customer			
cust_name	cust_no	policy_expiration_date	PARTITION
Smith	5	2008-12-15	2

The rows in the table are then the following, with the resolved CURRENT_DATE as April 1, 2009:

customer			
cust_name	cust_no	policy_expiration_date	PARTITION
Li	1	2009-01-31	2
Khan	2	2009-01-01	2
Reddy	3	2009-01-15	2

Example: Non-optimized Use of a CURRENT_DATE Function in a CREATE TABLE Request

This example specifies the CURRENT_DATE function in a CASE_N condition in a way that reconciliation using an ALTER TABLE ... TO CURRENT request cannot be optimized to scan fewer than all of the partitions. As a result, each time you submit an ALTER TABLE ... TO CURRENT request on *customer*, Vantage must perform a full-table scan to reconcile the row partitioning for the newly resolved date.

```
CREATE SET TABLE customer, NO FALLBACK (
  cust_name          CHARACTER(10),
  cust_no            INTEGER,
  policy_expiration_date DATE FORMAT 'YYYY/MM/DD')
PRIMARY INDEX(cust_no)
PARTITION BY CASE_N(policy_expiration_date=CURRENT_DATE,
                     NO CASE);
```

Example: Non-optimized Use of CURRENT_DATE Functions in CREATE TABLE Requests

This example also specifies the CURRENT_DATE function as an argument for CASE_N conditions in a way that reconciliation using an ALTER TABLE ... TO CURRENT request cannot be optimized to scan fewer than all of the row partitions of *customer*.

The request specifies a CURRENT_DATE function for two different partitioning columns, so Vantage cannot determine when all rows in certain row partitions are not impacted because there is no known relationship between the *policy_expiration_date* and *birth_date* columns that can be used to determine the impacts of a newly resolved date. As a result, each time you submit an ALTER TABLE ... TO CURRENT request on *customer*, Vantage must perform a full-table scan to reconcile the row partitioning for the new resolution date.

```
CREATE SET TABLE customer, NO FALLBACK (
  cust_name          CHARACTER(10),
  cust_no            INTEGER,
  birth_date          DATE,
  policy_expiration_date DATE FORMAT 'YYYY/MM/DD')
PRIMARY INDEX (cust_no)
PARTITION BY CASE_N(policy_expiration_date >= CURRENT_DATE,
                     policy_expiration_date >= CURRENT_DATE,
                     birth_date < CURRENT_DATE - INTERVAL '20' YEAR,
                     NO CASE);
```

Example: Altering the Row Partitioning of a Join Index

Assume that for the table in “Example: Altering the Row Partitioning of a Table When the Partitioning Expression Is Based on Multiple CURRENT_DATE Functions,” you define a row-partitioned sparse join

index *j_sales* on January 1, 2009 to contain the data of the current quarter in one row partition and all other data in another row partition for *sale_amt* greater than 2000.00.

```
CREATE JOIN INDEX j_sales AS
  SELECT *
  FROM sales
  WHERE sale_amt >= 2000.00
  PRIMARY INDEX (store_ID)
  PARTITION BY CASE_N(sale_date >= CURRENT_DATE, NO CASE);
```

Join index *j_sales* contains the following rows assuming that the resolved *CURRENT_DATE* is January 1, 2009:

j_sales			
store_ID	amount	sale_date	PARTITION
1	2000.00	2009-01-01	1
1	3500.00	2009-01-15	1
1	2000.00	2008-12-15	2
1	5000.00	2008-04-01	1

On April 1, 2009, you submit the following ALTER TABLE TO CURRENT request.

```
ALTER TABLE j_sales TO CURRENT;
```

The rows in join index *j_sales* are reconciled as follows:

j_sales			
store_ID	amount	sale_date	PARTITION
1	2000.00	2009-01-01	2
1	3500.00	2009-01-15	2
1	2000.00	2008-12-15	2
1	5000.00	2008-04-01	2

Example: Altering the Row Partitioning

This example demonstrates how using *CURRENT_TIMESTAMP* or *CURRENT_DATE* functions in a partitioning expression is most appropriate when the data must be partitioned as one or more current row partitions and one or more history row partitions, where current and history are defined with respect to the resolved *CURRENT_TIMESTAMP* or *CURRENT_DATE* function in the partitioning expression.

The example shows how you can periodically reconcile the table to move older data from the current row partition into one or more history partitions using an ALTER TABLE ... TO CURRENT request rather than redefining the row partitioning for the table using explicit dates that must be determined each time you submit an ALTER TABLE ... DROP RANGE or ALTER TABLE ... ADD RANGE request. You should be take care to evaluate your intended use of CURRENT_DATE and CURRENT_TIMESTAMP functions in a partitioning expression before you define the expressions on a table or join index.

Assume that the following table definition is created in the year 2009 (the CURRENT year at the time).

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_comment       VARCHAR(79))
PRIMARY INDEX(o_orderkey)
PARTITION BY RANGE_N(o_orderdate BETWEEN DATE '2004-01-01'
                      AND      DATE '2010-12-31'
                      EACH INTERVAL '1' MONTH)
UNIQUE INDEX(o_orderkey);
```

Assume that the table is row-partitioned to record 5 years of historical data, data for the current year, and data for one future year.

If you decide to alter the row partitioning for orders next year to maintain 5 years of history data, data for the current year, and data for one future year, you can submit an ALTER TABLE request such as this in 2011:

```
ALTER TABLE orders
MODIFY PRIMARY INDEX(o_orderkey)
DROP RANGE WHERE PARTITION BETWEEN 1 AND 12
ADD  RANGE BETWEEN DATE '2011-01-01'
                      AND      DATE '2011-12-31'
                      EACH INTERVAL '1' MONTH
WITH DELETE;
```

For this case, you must compute the new dates and specify them explicitly in the ADD RANGE clause of the request. This requires manual intervention every year you submit the request.

Suppose that instead of creating *orders* as was done the first time, you create it using a CURRENT_DATE function in the partitioning expression to simplify altering its partitions. This request assumes that as of the CREATE TABLE date, the last 5 years of historical data, data for the current year, and data for one future year are to be stored in the table for a total of 7 years, which is identical to the earlier case example.

The CREATE TABLE DDL for this case looks like this.

```

CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_comment       VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_orderdate BETWEEN
                     CAST(((EXTRACT(YEAR FROM
CURRENT_DATE)-5-1900)*10000+0101)                      AS DATE)
                     AND
                     CAST(((EXTRACT(YEAR FROM
CURRENT_DATE)+1-1900)*10000+1231)
                     AS DATE)
                     EACH INTERVAL '1' MONTH)
UNIQUE INDEX (o_orderkey);

```

You can schedule the following ALTER TABLE ... TO CURRENT request to be submitted annually. This request rolls the row partition window forward by efficiently dropping and adding partitions.

```
ALTER TABLE orders TO CURRENT WITH DELETE;
```

This statement does not need to be changed each time the data needs to be repartitioned based on the new dates, as would be the case if the row partitioning expression had not been specified using a CURRENT_DATE function.

In all of the preceding cases, the row partitioning begins on a year boundary.

The first case alters the partitioning in such a way that it continues to begin on a year boundary, but could be altered to roll forward to start on some month in a year by specifying the desired dates in the ALTER TABLE request.

The second case is designed only to roll forward to start on a year boundary.

You can use the following CREATE TABLE request to roll forward to begin partitioning on the first of a month. The case assumes that, as of the CREATE TABLE date, the last 71 months of history, the current month, and 12 future months are to be stored in the table, for a total of 84 months.

```

CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,

```

```

    o_orderpriority CHARACTER(21),
    o_comment        VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_orderdate BETWEEN
                        CAST(((EXTRACT(YEAR FROM
CURRENT_DATE)-1900)*10000 +
EXTRACT(MONTH FROM
CURRENT_DATE)*100+01) AS DATE) -
                        INTERVAL '71' MONTH
AND
                        CAST(((EXTRACT(YEAR FROM
CURRENT_DATE)+1-1900)*10000 +
EXTRACT(MONTH FROM
CURRENT_DATE)*100+01) AS DATE)+
                        INTERVAL '13' MONTH -
                        INTERVAL '1' DAY
                        EACH INTERVAL '1' MONTH)
UNIQUE INDEX (o_orderkey);

```

You can schedule the following ALTER TABLE ... TO CURRENT request to be submitted monthly or less frequently, but at some time before you run out of future months.

The request rolls the partition window forward by efficiently dropping and adding partitions so that, as of the ALTER TABLE TO CURRENT date, the last 71 months of history data, the data for the current month, and data for 12 months in the future can be contained in the table for a total of 84 months.

```
ALTER TABLE orders TO CURRENT WITH DELETE;
```

The following case uses simpler row partitioning, but possibly is not optimized because the entire table might need to be scanned in order to reconcile its rows when you submit an ALTER TABLE ... TO CURRENT request to change its row partitioning. The case assumes that, as of the CREATE TABLE date, approximately 2,191 days of history data, the data for the current day, and approximately 365 days of future data can be contained in the table, for a total of about 7 years.

```

CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_comment       VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_orderdate BETWEEN CURRENT_DATE -

```

```

                                INTERVAL '6' YEAR
                                AND CURRENT_DATE +
                                INTERVAL '1' YEAR
                                EACH INTERVAL '1' MONTH)
UNIQUE INDEX (o_orderkey);

```

You could schedule the following ALTER TABLE ... TO CURRENT request to be submitted daily or less frequently, but at some time before you run out of future days. The request rolls the partition window forward by dropping and adding row partitions only if CURRENT_DATE is the same day of the month as the last CREATE TABLE or ALTER TABLE ... TO CURRENT request was submitted. Otherwise, the entire table must be scanned to reconcile the rows.

```
ALTER TABLE orders TO CURRENT WITH DELETE;
```

This request can be very inefficient if you do not submit the ALTER TABLE ... TO CURRENT request on the same day of the month as the last CREATE TABLE or ALTER TABLE ... TO CURRENT request was submitted.

Performance degrades as a function of the increase in the number of days between the last resolved date and the new resolved date because of the increasing number rows that must be moved. For example, if the last resolved date was January 1, 2010 and the next ALTER TABLE ... TO CURRENT request is submitted on February 2, 2010, then Vantage would have to move all of the *orders* table rows to new partitions.

Also note that using the partitioning specified for this case, the system returns an error message reporting a non-valid date error if the CREATE TABLE or ALTER TABLE ... TO CURRENT request is submitted on February 29.

Related Information

- ALTER TABLE TO CURRENT in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- [CREATE JOIN INDEX](#)

For information about temporal tables, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

RENAME TABLE

Renames an existing table.

You cannot rename a temporal table if it has a system-defined single-table join index. You must first drop the system-defined join index and then rename the table. See *Teradata Vantage™ - Temporal Table Support*, B035-1182 for further information.

ANSI Compliance

RENAME TABLE is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have DROP privileges on the table to be renamed, and the appropriate CREATE privileges on its containing database or user.

RENAME TABLE Syntax

```
RENAME TABLE [ database_name. | user_name. ] old_table_name
               { TO | AS } [ database_name. | user_name. ] new_table_name [;]
```

RENAME TABLE Syntax Elements

database_name

user_name

Optional name of the containing database or user for the table to be renamed if other than the current database or user.

You cannot use this statement to change the database or user qualifier for the table.

old_table_name

Existing name for the table.

new_table_name

New name for the table.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

new_table_name

New name for the table.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

database_name

user_name

Optional name of the containing database or user for the renamed table if other than the current database or user.

RENAME TABLE Example

The following request renames the *emp* table as *employee*.

```
RENAME TABLE emp TO employee;
```

Related Information

- [CREATE TABLE and CREATE TABLE AS](#)
- [CREATE TABLE \(Queue Table Form\)](#)
- [DROP TABLE](#)

DROP TABLE

Drops the definition for a specified table from the Data Dictionary and from the containing database or user, depending on the keyword specified.

You can drop a base table or materialized global temporary table, depending on the keyword specified.

Keyword Specified	Database Object Dropped
TABLE	Base table definition and all table rows. In this case, the definition of a base table is restricted to persistent base tables, queue tables, and volatile tables. Also, drops definition of global temporary table (if TEMPORARY keyword is not specified). To drop a materialized global temporary table, you must specify the keyword TEMPORARY.
TEMPORARY TABLE	Materialized global temporary table.

To drop a data table that has an associated error table, you must first drop its error table. See [DROP ERROR TABLE](#).

ANSI Compliance

DROP TABLE is ANSI SQL:2011-compliant.

Required Privileges

You must have the appropriate DROP privilege on the specified base table or queue table.

You do not need the DROP privilege to drop materialized global temporary or volatile tables.

DROP TABLE Syntax

```
DROP [TEMPORARY] [FOREIGN] TABLE [ database_name. | user_name. ] table_name
[ALL] [;]
```

DROP TABLE Syntax Elements

TEMPORARY

The table to be dropped is the materialized instance of a global temporary table.

If you do not specify TEMPORARY and *table_name* identifies a global temporary table, then the base global temporary definition table is dropped.

If any materialized instances of the base global temporary table exist anywhere in the system, you cannot drop the table and an error is returned.

TEMPORARY is not valid when specified for any database object other than a global temporary table.

TEMPORARY is not valid when specified with the ALL keyword.

FOREIGN

Optional keyword you can specify for readability when dropping a foreign table.

When you drop a foreign table, the external file source is not modified in any way. Only the Vantage local information is removed.

database_name

user_name

Name of the containing database or user for the specified base table or queue table.

This specification is required only if the table to be dropped is contained in a different database than the current database.

table_name

Name of the base table or queue table to be dropped.

The *table_name* specification cannot identify a journal table.

If the table to be dropped has triggers, hash indexes, or join indexes associated with it, you must first drop those triggers, hash indexes, and join indexes else the request aborts.

ALL

The base global temporary table definition and all its materialized instances are to be dropped when they are not being accessed by any other transactions.

ALL is not valid when specified for any database object other than a global temporary table.

ALL is not valid when specified with the TEMPORARY keyword.

Take care in using this option because the time required for all open transactions accessing this object to commit might be significant.

If you do not specify ALL and if the global temporary table is materialized in any other session, Vantage aborts the request.

DROP TABLE Examples

Example: Dropping a Table

This request drops a table named *order_ids*.

```
DROP TABLE order_ids;
```

Example: Dropping a Foreign Table

This statement drops the foreign table *table1*.

```
DROP FOREIGN TABLE MyDB.table1;
```

Related Information

- CREATE TABLE, CREATE TABLE (Queue Table Form), and HELP TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - Database Design*, B035-1094

DROP ERROR TABLE

Deletes the specified error table object definition from the Data Dictionary and from the containing database or user.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the DROP TABLE privilege on the specified error table.

DROP ERROR TABLE Syntax

```
DROP ERROR TABLE {
  FOR [ database_name. | user_name. ] data_table_name |
  [ database_name. | user_name. ] error_table_name
} [;]
```

DROP ERROR TABLE Syntax Elements

FOR

This syntax is useful if you did not define a name for the error table to be dropped and you do not know its system-assigned default name.

database_name

user_name

The containing database or user for the specified error table if it is defined in a database other than the current database.

data_table_name

Name of the data table for which the error table to be dropped is defined.

error_table_name

Name of the error table to be dropped.

Example: Dropping an Error Table

This example deletes the error table defined on the *orders* table.

```
DROP ERROR TABLE FOR orders;
```

Related Information

- CREATE ERROR TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- [DROP MACRO](#)
- DROP MACRO in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184

HELP COLUMN

Displays the attributes of a column, including whether it is a single-column primary or secondary index and, if so, whether it is unique.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must either own the table, join index, or hash index in which the column is defined or have at least one privilege on that table.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against a specified table, join index, or hash index.

HELP COLUMN Syntax

Column or Column List from Table

```
HELP COLUMN column_name [,...] FROM source [,...] [;]
```

All Columns from Table

```
HELP COLUMN * FROM source [,...] [;]
```

Table Columns

```
HELP COLUMN column_spec [,...] [;]
```

All Columns in Tables or Indexes

```
HELP COLUMN column_spec [,...] [;]
```

Column Expression

```
HELP COLUMN expression [,...] [;]
```

Column Expression from Table or Index

```
HELP COLUMN column_spec [,...] [;]
```

Column from Error Table for Data Table

```
HELP COLUMN column_name FROM ERROR TABLE FOR [ database_name. | user_name. ]
data_table_name [;]
```

Column from Error Table

```
HELP COLUMN column_name FROM [ database_name. | user_name. ] error_table_name [;]
```

source

```
[ database_name. | user_name. ]
{ table_name | join_index_name | hash_index_name }
```

column_spec

```
source.column_name
```

HELP COLUMN Syntax Elements**column_name**

Name of the column for which to display attributes.

*

Attribute information for all the columns in the named table, join index, or hash index is requested.

expression

An expression, either based on a column or not, for which the data type is requested.

column_name**database_name****user_name**

Containing database or user for *table_name*, *error_table_name*, *data_table_name*, *hash_index_name*, or *join_index_name* if different from the current database or user.

data_table_name

Name of the data table to which the error table is attached.

error_table_name
Name of the error table.

table_name
Name of the table for which attribute information is requested. You cannot request help for a journal table.

join_index_name
Name of the join index for which attribute information is requested.

hash_index_name
Name of the hash index for which attribute information is requested.

Examples

Example: HELP COLUMN Sample Output

The following shows a sample output display for HELP COLUMN on the table defined by the following CREATE TABLE statement.

```
CREATE TABLE table_1 (  
    column_1 INTEGER NOT NULL PRIMARY KEY,  
    column_2 INTEGER CHECK (column_2>0) CHECK (column_2<100));  
HELP COLUMN table_1.column_2;
```

Column Name	column_2
Type	I
Nullable	Y
Format	-(10)9
Max Length	4
Decimal Total Digits	?
Decimal Fractional Digits	?
Range Low	?
Range High	?
Uppercase	N
Table/View?	T

Indexed?	N
Unique?	?
Primary?	?
Title	?
Column Constraint	CHECK ((column_2>0) AND (column_2<100))
Char Type	?
IdCol Type	?
UDT Name	?
Temporal Column	N
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	N
Derived_UDT	?
Derived_UDTFieldID	?
Column Dictionary Name	column_2
Column SQL Name	column_2
Column Name UEscape	?
Dictionary Title	?
SQL Title	?
Title UEscape	?
UDT Database Dictionary Name	?
UDT Database SQL Name	?
UDT Database Name UEscape	?
UDT Dictionary Name	?

UDT SQL Name	?
UDT Name UEscape	?
Without Overlaps Unique	?
Storage Format	?

In BTEQ, a question mark (?) character indicates a null value.

Example: HELP COLUMN column_name

The following two requests return information about the attributes of the named column set in the named table or view.

```
HELP COLUMN column_name
FROM table_name;
HELP COLUMN table_name.column_name;
```

Example: HELP COLUMN for All Columns

The following two requests return attribute information for all the columns of the named table or view. All column attributes are returned, except any default values and comments (for these, use HELP TABLE table_name).

```
HELP COLUMN * FROM table_name;
HELP COLUMN table_name.*;
```

Example: HELP COLUMN for Fully Qualified Columns

The following query requests information about particular columns in the employee and department tables (note that each column reference is fully qualified with its table name).

```
HELP COLUMN employee.name, employee.dept_no, department.dept_no
FROM employee, department;
```

The return from the preceding query lists each column in the order in which it was specified; therefore, the first *dept_no* column listed is for the employee table, the second is for the department table.

Column Name	Type	Nullable
-----	----	-----
Name	CV	N

DeptNo	I2	Y
DeptNo	I2	N

Example: HELP COLUMN for a Table

A KanjiEBCDIC user creates a table using four character data types:

```
CREATE TABLE kanji_user.table_1,
  NO FALLBACK,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL (
    clatin  CHARACTER(5) CHARACTER SET LATIN,
    cgraphic CHARACTER(5) CHARACTER SET GRAPHIC,
    csjis   CHARACTER(5) CHARACTER SET KANJI SJIS,
    cunicode CHARACTER(5) CHARACTER SET UNICODE)
PRIMARY INDEX (clatin);
```

This HELP COLUMN request displays information for all columns:

```
HELP COLUMN table_1.*;
```

Following is the abbreviated BTEQ output formatted using the SIDETITLES and FOLDLINE options.

Column Name	clatin
Type	CF
Format	X(5)
Max Length	5
Title	?
Column Constraint	?
Char Type	1
IdCol Type	?
UDT Name	?
Temporal Column	?
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
Nonsequenced ValidTime Unique	?
Current TransactionTime Unique	?

Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	N
Derived_UDT	?
Derived_UDTFieldID	?

Column Name	cgraphic
Type	CF
Format	X(5)
Max Length	10
Title	?
Column Constraint	?
Char Type	4
IdCol Type	?
Temporal Column	N
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
ConstraintID	?
Derived_UDT	?
Derived_UDTFieldID	?

Column Name	csjis
-------------	-------

Type	CF
Format	X(5)
Max Length	5
Title	?
Column Constraint	?
Char Type	3
IdCol Type	?
UDT Name	?
Temporal Column	N
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	?
Derived_UDT	?
Derived_UDTFieldID	?

Column Name	cunicode
Type	CF
Format	X(5)
Max Length	10
Title	?
Column Constraint	?
Char Type	2
IdCol Type	?
UDT Name	?

Temporal Column	N
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	N
Derived_UDT	?
Derived_UDTFieldID	?

Example: HELP COLUMN On Non-Column Expressions Only

The following request executes HELP COLUMN on 3 expressions, but no actual columns. Note that report does not include column names, but instead lists the data types, nullability, and output format.

```
HELP COLUMN 1+10, 10.1+1, 'asdf' || 'asd1';
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.
Column Name          Type Nullable Format
-----
                I      Y      -(10)9
                D      Y      -----.9
                CV     Y      X(8)
```

Example: HELP COLUMN On Multiple *table_name.** Expressions

The following request executes HELP COLUMN on all columns of two separate tables.

```
HELP COLUMN d.*, transaction_detail.*;
*** Help information returned. 8 rows.
*** Total elapsed time was 1 second.
Column Name          Type Nullable Format
-----
```

transaction_header_key	I	Y	-(10)9
vendor_key	I	Y	-(10)9
item_key	I	Y	-(10)9
stuff	CF	Y	X(300)
transaction_header_key	I	Y	-(10)9
vendor_key	I	Y	-(10)9
item_key	I	Y	-(10)9
stuff	CF	Y	X(300)

Example: HELP COLUMN On a Mixture of Non-Column Expressions and table_name .* Expressions

The following request executes HELP COLUMN on an integer expression and all columns of the d table. Note that the report does not report a column name for the integer expression.

```
HELP COLUMN 1+1, d.*;
*** Help information returned. 5 rows.
*** Total elapsed time was 1 second.
```

Column Name	Type	Nullable	Format
-----	I	Y	-(10)9
transaction_header_key	I	Y	-(10)9
vendor_key	I	Y	-(10)9
item_key	I	Y	-(10)9
stuff	CF	Y	X(300)

Example: Partitioned Primary Index Table

This example uses the following table definition.

```
CREATE TABLE Orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk         CHARACTER(16),
  o_shippriority  INTEGER,
  o_comment       VARCHAR(79))
PRIMARY INDEX orders_PI (o_orderkey)
PARTITION BY RANGE_N(o_orderdate BETWEEN DATE '1992-01-01'
```

```

AND      DATE '1998-12-31'
EACH INTERVAL '1' MONTH)

UNIQUE INDEX (o_orderkey)
INDEX (o_custkey)
INDEX (o_orderdate) ORDER BY VALUES
INDEX (o_orderdate)
INDEX (o_orderdate, o_custkey) ORDER BY VALUES(o_orderdate)
INDEX (o_orderdate, o_orderstatus);

```

HELP COLUMN generates the following report on this table. The BTEQ commands are included to show how the report format was derived. Attributes of particular interest for this example are highlighted in boldface.

Note the following things about this report.

- *o_orderstatus* is a component of a composite secondary index.
- *o_totalprice* is not a component of any index.

```

BTEQ -- Enter your DBC/SQL request or BTEQ command:
.SIDETITLES
BTEQ -- Enter your DBC/SQL request or BTEQ command:
.FOLDLINE
BTEQ -- Enter your DBC/SQL request or BTEQ command:
HELP COLUMN * FROM orders;
*** Help information returned. 9 rows.
*** Total elapsed time was 1 second.

```

Column Name	o_orderkey
Type	I
Nullable	N
Format	-(10)9
Max Length	4
Decimal Total Digits	?
Decimal Fractional Digits	?
Range Low	?
Range High	?
UpperCase	N
Table/View?	T
Indexed?	Y

Unique?	N
Primary?	P
Title	?
Column Constraint	?
Char Type	?
IdCol Type	?
UDT Name	?
Temporal Column	?
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	?
Derived_UDT	?
Derived_UDTFieldID	?
Column Dictionary Name	o_orderkey
Column SQL Name	o_orderkey
Column Name UEscape	?
Dictionary Title	?
SQL Title	?
Title UEscape	?
UDT Database Dictionary Name	?
UDT Database SQL Name	?
UDT Database Name UEscape	?
UDT Dictionary Name	?
UDT SQL Name	?

UDT Name UEscape	?
Without Overlaps Unique	?
Storage Format	?

Column Name	o_custkey
Type	I
Nullable	Y
Format	-(10)9
Max Length	4
Decimal Total Digits	?
Decimal Fractional Digits	?
Range Low	?
Range High	?
UpperCase	N
Table/View?	T
Indexed?	Y
Unique?	N
Primary?	S
Title	?
Column Constraint	?
Char Type	?
IdCol Type	?
UDT Name	?
Temporal Column	N
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N
Column Partition Number	0

Column Partition Format	NA
Column Partition AC	N
Security Constraint	?
Derived_UDT	?
Derived_UDTFieldID	?
Column Dictionary Name	o_custkey
Column SQL Name	o_custkey
Column Name UEscape	?
Dictionary Title	?
SQL Title	?
Title UEscape	?
UDT Database Dictionary Name	?
UDT Database SQL Name	?
UDT Database Name UEscape	?
UDT Dictionary Name	?
UDT SQL Name	?
UDT Name UEscape	?
Without Overlaps Unique	?
Storage Format	?

Column Name	o_orderstatus
Type	CF
Nullable	Y
Format	X(1)
Max Length	1
Decimal Total Digits	?
Decimal Fractional Digits	?
Range Low	?
Range High	?
UpperCase	C

Table/View?	T
Indexed?	Y
Unique?	N
Primary?	S
Title	?
Column Constraint	?
Char Type	1
IdCol Type	?
UDT Name	?
Temporal Column	N
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	?
Derived_UDT	?
Derived_UDTFieldID	?
Column Dictionary Name	o_orderstatus
Column SQL Name	o_orderstatus
Column Name UEscape	?
Dictionary Title	?
SQL Title	?
Title UEscape	?
UDT Database Dictionary Name	?
UDT Database SQL Name	?
UDT Database Name UEscape	?

UDT Dictionary Name	?
UDT SQL Name	?
UDT Name UEscape	?
Without Overlaps Unique	?
Storage Format	?

Column Name	o_totalprice
Type	D
Nullable	N
Format	-----,99
Max Length	8
Decimal Total Digits	13
Decimal Fractional Digits	2
Range Low	?
Range High	?
UpperCase	N
Table/View?	T
Indexed?	N
Unique?	?
Primary?	?
Title	?
Column Constraint	?
Char Type	?
IdCol Type	?
UDT Name	?
Temporal Column	N
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?

Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	N
Derived_UDT	?
Derived_UDTFieldID	?
Column Dictionary Name	o_totalprice
Column SQL Name	o_totalprice
Column Name UEscape	?
Dictionary Title	?
SQL Title	?
Title UEscape	?
UDT Database Dictionary Name	?
UDT Database SQL Name	?
UDT Database Name UEscape	?
UDT Dictionary Name	?
UDT SQL Name	?
UDT Name UEscape	?
Without Overlaps Unique	?
Storage Format	?

Column Name	o_orderdate
Type	DA
Nullable	N
Format	yyyy-mm-dd
Max Length	4
Decimal Total Digits	?
Decimal Fractional Digits	?
Range Low	?

Range High	?
UpperCase	N
Table/View?	T
Indexed?	Y
Unique?	N
Primary?	S
Title	?
Column Constraint	?
Char Type	?
IdCol Type	?
UDT Name	?
Temporal Column	N
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	?
Derived_UDT	?
Derived_UDTFieldID	?
Column Dictionary Name	o_orderdate
Column SQL Name	o_orderdate
Column Name UEscape	?
Dictionary Title	?
SQL Title	?
Title UEscape	?
UDT Database Dictionary Name	?

UDT Database SQL Name	?
UDT Database Name UEscape	?
UDT Dictionary Name	?
UDT SQL Name	?
UDT Name UEscape	?
Without Overlaps Unique	?
Storage Format	?

Column Name	o_orderpriority
Type	CF
Nullable	Y
Format	X(21)
Max Length	21
Decimal Total Digits	?
Decimal Fractional Digits	?
Range Low	?
Range High	?
UpperCase	N
Table/View?	T
Indexed?	N
Unique?	?
Primary?	?
Title	?
Column Constraint	?
Char Type	1
IdCol Type	?
UDT Name	?
Temporal Column	N
Current ValidTime Unique	?
Sequenced ValidTime Unique	?

NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	?
Derived_UDT	?
Derived_UDTFieldID	?
Column Dictionary Name	o_orderpriority
Column SQL Name	o_orderpriority
Column Name UEscape	?
Dictionary Title	?
SQL Title	?
Title UEscape	?
UDT Database Dictionary Name	?
UDT Database SQL Name	?
UDT Database Name UEscape	?
UDT Dictionary Name	?
UDT SQL Name	?
UDT Name UEscape	?
Without Overlaps Unique	?
Storage Format	?

Column Name	o_clerk
Type	CF
Nullable	Y
Format	X(16)
Max Length	16
Decimal Total Digits	?

Decimal Fractional Digits	?
Range Low	?
Range High	?
UpperCase	N
Table/View?	T
Indexed?	N
Unique?	?
Primary?	?
Title	?
Column Constraint	?
Char Type	1
IdCol Type	?
UDT Name	?
Temporal Column	N
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	?
Derived_UDT	?
Derived_UDTFieldID	?
Column Dictionary Name	o_clerk
Column SQL Name	o_clerk
Column Name UEscape	?
Dictionary Title	?
SQL Title	?

Title UEscape	?
UDT Database Dictionary Name	?
UDT Database SQL Name	?
UDT Database Name UEscape	?
UDT Dictionary Name	?
UDT SQL Name	?
UDT Name UEscape	?
Without Overlaps Unique	?
Storage Format	?

Column Name	o_shippriority
Type	I
Nullable	Y
Format	-(10)9
Max Length	4
Decimal Total Digits	?
Decimal Fractional Digits	?
Range Low	?
Range High	?
UpperCase	N
Table/View?	T
Indexed?	N
Unique?	?
Primary?	?
Title	?
Column Constraint	?
Char Type	?
IdCol Type	?
UDT Name	?
Temporal Column	N

Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	N
Derived_UDT	?
Derived_UDTFieldID	?
Column Dictionary Name	o_shippriority
Column SQL Name	o_shippriority
Column Name UEscape	?
Dictionary Title	?
SQL Title	?
Title UEscape	?
UDT Database Dictionary Name	?
UDT Database SQL Name	?
UDT Database Name UEscape	?
UDT Dictionary Name	?
UDT SQL Name	?
UDT Name UEscape	?
Without Overlaps Unique	?
Storage Format	?

Column Name	o_comment
Type	CV
Nullable	Y
Format	X(79)

Max Length	79
Decimal Total Digits	?
Decimal Fractional Digits	?
Range Low	?
Range High	?
UpperCase	N
Table/View?	T
Indexed?	N
Unique?	?
Primary?	?
Title	?
Column Constraint	?
Char Type	1
IdCol Type	?
UDT Name	?
Temporal Column	N
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	Y
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	N
Derived_UDT	?
Derived_UDTFieldID	?
Column Dictionary Name	o_comment
Column SQL Name	o_comment
Column Name UEscape	?

Dictionary Title	?
SQL Title	?
Title UEscape	?
UDT Database Dictionary Name	?
UDT Database SQL Name	?
UDT Database Name UEscape	?
UDT Dictionary Name	?
UDT SQL Name	?
UDT Name UEscape	?
Without Overlaps Unique	?
Storage Format	?

Example: HELP COLUMN for a Distinct UDT Column

The following example shows possible output for a HELP COLUMN request for a column with a distinct UDT type.

```
HELP COLUMN t1.distinct_column;
```

Column Name	distinct_column
Type	UDT
Nullable	N
Format	?
Max Length	4
Decimal Total Digits	?
Decimal Fractional Digits	?
Range Low	?
Range High	?
UpperCase	N
Table/View?	T
Indexed?	N
Unique?	?

Primary?	?
Title?	?
Column Constraint?	?
Char Type?	?
UDT Name	SYSUDTLIB.distinct_type_name
Temporal Column	N
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	?
Derived_UDT	?
Derived_UDTFieldID	?
Column Dictionary Name	distinct_column
Column SQL Name	distinct_column
Column Name UEscape	?
Dictionary Title	?
SQL Title	?
Title UEscape	?
UDT Database Dictionary Name	?
UDT Database SQL Name	?
UDT Database Name UEscape	?
UDT Dictionary Name	?
UDT SQL Name	?
UDT Name UEscape	?
Without Overlaps Unique	?

Storage Format	?
----------------	---

Example: HELP COLUMN with a Derived Period Column

This is an example of HELP COLUMN output for the following table definition, which includes a derived period column.

```
CREATE TABLE employee (
    eid INTEGER NOT NULL,
    name VARCHAR(100) NOT NULL,
    deptno INTEGER NOT NULL,
    jobstart DATE NOT NULL,
    jobend DATE NOT NULL,
    PERIOD FOR jobduration (jobstart, jobend)
    PRIMARY INDEX(eid);
```

Below is the HELP COLUMN output.

```
HELP COLUMN employee.jobduration;
```

Column Name	jobduration
Type	PD
Nullable	?
Format	?
Max Length	0
Decimal Total Digits	?
Decimal Fractional Digits	?
Range Low	?
Range High	?
UpperCase	?
Table/View?	T
Indexed?	?
Unique?	?
Primary?	?
Title	?

Column Constraint	?
Char Type	?
IdCol Type	?
UDT Name	?
Temporal Column	N
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NC
Security Constraint	N
Derived_UDT	PP
Derived_UDT_FieldID	?
Column Dictionary Name	jobduration
Column SQL Name	jobduration
Column Name UEscape	?
Dictionary Title	?
SQL Title	?
Title UEscape	?
UDT Database Dictionary Name	?
UDT Database SQL Name	?
UDT Database Name UEscape	?
UDT Dictionary Name	?
UDT SQL Name	?
UDT Name UEscape	?
Without Overlaps Unique	?
Storage Format	?

Example: HELP COLUMN for Temporal Columns

This example shows the report returned by a HELP COLUMN request on temporal columns.

Assume the following table definition.

```
CREATE MULTISET TABLE department (
  deptname      VARCHAR(10),
  deptno        INTEGER NOT NULL UNIQUE CHECK(dept_no > 0),
  avgsalary      DECIMAL(9,2),
  deptduration   PERIOD(DATE) NOT NULL AS VALIDTIME,
  depttran       PERIOD(TIMESTAMP(6) WITH TIME ZONE) AS TRANSACTIONTIME)
PRIMARY INDEX (deptno);
```

The following report is an example of a HELP COLUMN request on the *dept_tran* column for this table.

```
HELP COLUMN department.depttran;
HELP COLUMN department.dept_tran;
*** Help information returned. 5 rows.
```

Column Name	dept_tran
Type	PM
Nullable	N
Format	YYYY-MM-DDBHH:MI:SS.S(6)Z
Max Length	24
Decimal Total Digits	?
Decimal Fractional Digits	6
Range Low	?
Range High	?
UpperCase	N
Table/View?	T
Indexed?	N
Unique?	?
Primary?	?
Title	?
Column Constraint	?

Char Type	?
IdCol Type	?
UDT Name	?
Temporal Column	T
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
NonSequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N
Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	N
Derived_UDT	?
Derived_UDTFieldID	?
Column Dictionary Name	depttran
Column SQL Name	depttran
Column Name UEscape	?
Dictionary Title	?
SQL Title	?
Title UEscape	?
UDT Database Dictionary Name	?
UDT Database SQL Name	?
UDT Database Name UEscape	?
UDT Dictionary Name	?
UDT SQL Name	?
UDT Name UEscape	?
Without Overlaps Unique	?
Storage Format	?

The following report is an example of a HELP COLUMN request on the *dept_no* column in this table.

```
HELP COLUMN department.dept_no;
HELP COLUMN department.dept_no;
*** Help information returned. One row.
*** Total elapsed time was 1 second.
```

Column Name	dept_no
Type	I
Nullable	N
Format	-(10)9
Max Length	4
Decimal Total Digits	?
Decimal Fractional Digits	?
Range Low	?
Range High	?
UpperCase	N
Table/View?	T
Indexed?	Y
Unique?	Y
Primary?	S
Title	?
Column Constraint	CURRENT VALIDTIME AND CURRENT TRANSACTIONTIME CHECK(deptno>0)
Char Type	?
IdCol Type	?
UDT Name	?
Temporal Column	?
Current ValidTime Unique	?
Sequenced ValidTime Unique	?
Nonsequenced ValidTime Unique	?
Current TransactionTime Unique	?
Partitioning Column	N

Column Partition Number	0
Column Partition Format	NA
Column Partition AC	NA
Security Constraint	N
Derived_UDT	?
Derived_UDTFieldID	?
Column Dictionary Name	dept_no
Column SQL Name	dept_no
Column Name UEscape	?
Dictionary Title	?
SQL Title	?
Title UEscape	?
UDT Database Dictionary Name	?
UDT Database SQL Name	?
UDT Database Name UEscape	?
UDT Dictionary Name	?
UDT SQL Name	?
UDT Name UEscape	?
Without Overlaps Unique	?
Storage Format	?

Example: HELP COLUMN for a JSON Column with the Auto Column Option

Here is the table definition of MyTable for this example, where the JSON column c is defined with the AUTO COLUMN option.

```
CREATE TABLE TEST.MyTable (
  a INTEGER,
  b JSON(16776192) INLINE LENGTH 4096 CHARACTER SET LATIN,
  c JSON(16776192) INLINE LENGTH 4096 CHARACTER SET LATIN AUTO COLUMN)
PRIMARY INDEX ( a );
```

This statement displays information for column c of the table MyTable.

```
HELP COLUMN MyTable.c;
```

Below is the abbreviated HELP COLUMN output for selected attributes.

Column Name	c
Type	JN
Nullable	Y
Format	X(64000)
Max Length	16,776,192
Table/View?	T
Storage Format	TEXT
Inline Length	4,096
Transform Length	16,776,192
Auto Column	Y

If the AUTO COLUMN option is not set, the value is NULL, or question mark (?) in BTEQ output.

Related Information

- HELP COLUMN in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.
- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Temporal Table Support*, B035-1182

HELP CONSTRAINT

Displays the attributes for a specific named constraint on a table. Use the SHOW TABLE statement to obtain unnamed constraint information. See “SHOW object.”

The HELP CONSTRAINT statement does not report information for row-level security constraints. Specify the CONSTRAINT option with a HELP SESSION request to report the constraint values associated with a session.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must own the table in which the constraint is defined or have at least one privilege on that table.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against a specified constraint.

HELP CONSTRAINT Syntax

```
HELP CONSTRAINT [ database_name. | user_name. ]
                { table_name. | view_name. } constraint_name [;]
```

HELP CONSTRAINT Syntax Elements

Name of the constraint for which to display attribute information.

database_name

user_name

Name of the containing database or user for *table_name* or *view_name* if different from the current database or user.

table_name.constraint_name

Table-qualified name of a constraint for which attribute information is required.

view_name.constraint_name

View-qualified name of a constraint for which attribute information is required.

HELP CONSTRAINT Examples

Example: CHECK Constraint

This example shows the report returned for a table with a CHECK constraint.

```
HELP CONSTRAINT table_1.check_1;
      Name check_1
      Type CHECK
      Constraint CHECK(column_1 > 0 AND column_2 > 0)
```

Example: Temporal CHECK Constraints

Assume the following table definition.

```
CREATE TABLE temporal.department (
  dept_name      VARCHAR(10),
  dept_no        INTEGER NOT NULL UNIQUE,
  dept_duration   PERIOD(DATE) AS VALIDTIME,
```

```
CONSTRAINT vt_check_1 NONSEQUENCED VALIDTIME CHECK(dept_no < 100))
PRIMARY INDEX (dept_name);
```

The following is the output of a HELP CONSTRAINT on this table using BTEQ.

```
HELP CONSTRAINT temporal.department.vtcheck1;
*** Help information returned. One row.
*** Total elapsed time was 1 second.
Name vt_check_1
Type CHECK
Constraint CONSTRAINT vt_check_1 NONSEQUENCED VALIDTIME CHECK
      (dept_no<100)
```

Example: REFERENTIAL Constraint

The following example shows the report returned for a table with REFERENTIAL constraints.

```
HELP CONSTRAINT table_1.reference_1;
      Name reference_1
      Type REFERENCES
      State VALID
      Index Id 4
      FK Columns column_1, column_2
Parent Key Database Name dev
      Parent Table Name table_2
      Parent Index ID 5
      Parent Key Columns column_3, column_4
```

Example: Temporal UNIQUE Constraints

This example uses the following table definition.

```
CREATE TABLE temporal.tab_1 (
  col_1  VARCHAR(10),
  col_2  INTEGER NOT NULL,
  col_3  INTEGER NOT NULL,
  vt_col PERIOD(DATE) AS VALIDTIME,
  CONSTRAINT vt_uniq_1 NONSEQUENCED VALIDTIME UNIQUE(col_2),
  CONSTRAINT vt_uniq_2 UNIQUE (col_3))
PRIMARY INDEX (col_1);
```

The following is the output of a HELP CONSTRAINT on this table using BTEQ.


```

HELP CONSTRAINT temporal.tab1.vtuniq1;
*** Help information returned. One row.
*** Total elapsed time was 1 second.
           Name vt_uniq_1
           Type SEQUENCED VALIDTIME UNIQUE
           Unique? Y
           Index Id 4
           Column Names col_2

```

Example: Temporal UNIQUE Constraints

This example uses the following table definition.

```

CREATE TABLE temporal.tab_1 (
  col_1  VARCHAR(10),
  col_2  INTEGER NOT NULL,
  col_3  INTEGER NOT NULL,
  vt_col PERIOD(DATE) AS VALIDTIME,
  CONSTRAINT vt_uniq_1 NONSEQUENCED VALIDTIME UNIQUE(col_2),
  CONSTRAINT vt_uniq_2 UNIQUE (col_3))
PRIMARY INDEX (col_1);

```

The following is the output of a HELP CONSTRAINT on this table using BTEQ.

```

HELP CONSTRAINT temporal.tab1.vtuniq1;
*** Help information returned. One row.
*** Total elapsed time was 1 second.
           Name vt_uniq_1
           Type SEQUENCED VALIDTIME UNIQUE
           Unique? Y
           Index Id 4
           Column Names col_2

```

Related Information

- CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- HELP CONSTRAINT in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- [SHOW object](#)
- *Teradata Vantage™ - Database Design*, B035-1094
- *Teradata Vantage™ - Temporal Table Support*, B035-1182

HELP ERROR TABLE

Displays the attributes for a specified error table.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

To execute this statement, you must have either of the following privileges.

- Ownership of the specified data tables.
- At least one privilege on the specified object.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against a specified error table.

HELP ERROR TABLE Syntax

Error Table for Data Table

```
HELP ERROR TABLE FOR [ database_name. | user_name. ] data_table_name [;]
```

Error Table

```
HELP TABLE [ database_name. | user_name. ] error_table_name [;]
```

HELP ERROR TABLE Syntax Elements

database_name

Containing database for the specified data table or error table if it is defined in a database other than the current database.

user_name

Containing user for the specified data table or error table if it is defined in a database other than the current user.

data_table_name

Name of the data table for which the error table is defined. This syntax is particularly useful if you did not define a name for the error table and you do not know its system-assigned default name.

error_table_name

Name of the error table for which help information is requested.

Usage Notes

The reports generated by `HELP ERROR TABLE` are identical to those produced by `HELP TABLE` except that they always include information about the additional error table-specific columns listed in the `HELP ERROR TABLE` “System-Defined Attribute Characteristics for the Error Table-Specific Columns” topic in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

The attributes for the two reports are identical (see `HELP MACRO` in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184).

See [Example: HELP TABLE](#).

Related Information

- `INSERT` and `INSERT ... SELECT` in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146
- `MERGE` in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146
- [HELP MACRO](#)
- [SHOW object](#)

HELP TABLE

Displays the attributes for a specified base data table.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

To execute this statement, you must have either of the following privileges.

- Ownership of the specified base table or queue table.
- At least one privilege on the requested object.

Use the `SHOW` privilege to enable a user to perform `HELP` or `SHOW` requests only against a specified table.

HELP TABLE Syntax

```
HELP TABLE [ database_name. | user_name. ] { table_name | error_table_name } [;]
```

HELP TABLE Syntax Elements

database_name

user_name

The containing database or user for *table_name* if something other than the current database or user.

table_name

Table or queue table for which help is required.

error_table_name

Error table for which help is required. See [HELP ERROR TABLE](#) for details.

Usage Notes

Table Columns

For some larger tables, a request to display all or most of the columns returns an error message. If this occurs, you can display the table definition with the SHOW TABLE statement, and may be able to display column attributes by querying the Data Dictionary view named DBC.ColumnsV.

Examples

Example: HELP TABLE

The definition of the department table is the following.

```
CREATE TABLE personnel.department (
  dept_no    SMALLINT NOT NULL,
  dept_name  VARCHAR(12) CHARACTER SET UNICODE,
  loc        CHARACTER(8) CHARACTER SET UNICODE,
  mgr_no     INTEGER NOT NULL)
UNIQUE PRIMARY INDEX (dept_no);
```

The following request returns information about the department table.

```
HELP TABLE personnel.department;
      Column Name dept_no
```

```

        Type I2
        Comment Department number
        Nullable N
        Format -(10)9
        Title ?
        Max Length 4
        Decimal Total Digits ?
        Decimal Fractional Digits ?
        Range Low ?
        Range High ?
        Uppercase N
        Table/View? T
        Default value ?
        Character Type ?
        IDCol Type ?
        UDT Name ?
        Temporal N
        Column Dictionary Name dept_no
        Column SQL Name dept_no
        Column Name UEscape
        Dictionary Title
        SQL Title
        Title UEscape
        Column Name dept_name
        Type CV
        Comment Department name
        Nullable N
        Format X(12)
        Title ?
        Max Length 8
        Decimal Total Digits ?
        Decimal Fractional Digits ?
        Range Low ?
        Range High ?
        Uppercase N
        Table/View? T
        Default Value ?
        Character Type 2
        IDCol Type ?
        UDT Name ?
        Temporal N
        Column Dictionary Name dept_name
        Column SQL Name dept_name
        Column Name UEscape

```

```

Dictionary Title
  SQL Title
  Title UEscape
  Column Name loc
  Type CF
  Comment Department location
  Nullable N
  Format X(18)
  Title ?
  Max Length 18
Decimal Total Digits ?
Decimal Fractional Digits ?
  Range Low ?
  Range High ?
  Uppercase N
  Table/View? T
  Default Value ?
Character Type 2
  IDCol Type ?
  UDT Name ?
  Temporal N
Column Dictionary Name loc
  Column SQL Name loc
  Column Name UEscape
  Dictionary Title
  SQL Title
  Title UEscape
  Column Name mgr_no
  Type I
  Comment Employee number of department manager
  Nullable N
  Format -(10)9
  Title ?
  Max Length ?
  Decimal Total Digits ?
Decimal Fractional Digits ?
  Range Low ?
  Range High ?
  Uppercase ?
  Table/View? T
  Indexed? Y
  Unique? Y
  Default Value ?
Character Type ?

```

```

IDCol Type ?
UDT Name ?
Temporal N
Column Dictionary Name mgr_no
Column SQL Name mgr_no
Column Name UEscape
Dictionary Title
SQL Title
Title UEscape

```

Example: HELP TABLE with a UDT Column

This example reports information about table *t1*, which is defined as follows.

```

CREATE TABLE t1 (
  id    INTEGER NOT NULL,
  myint udt_int)
UNIQUE PRIMARY INDEX (id);

```

The following example shows the HELP TABLE report for a table named *t1* that contains a UDT column named *myint* with the UDT type *udt_int*.

```

HELP TABLE t1;

      Column Name id
            Type I
            Comment ?
            Nullable N
            Format -(10)9
            Title ?
      Max Length 4
    Decimal Total Digits ?
  Decimal Fractional Digits ?
        Range Low ?
        Range High ?
        UpperCase N
    Table/View? T
    Default Value ?
        Char Type ?
        IdCol Type ?
        UDT Name ?
        Temporal ?
    Column Dictionary Name id
      Column SQL Name id

```

```

Column Name UEscape
Dictionary Title
SQL Title
Title UEscape
Column Dictionary Name
Column Name myint
Type UT
Comment ?
Nullable Y
Format -(10)9
Title ?
Max Length 4
Decimal Total Digits ?
Decimal Fractional Digits ?
Range Low ?
Range High ?
UpperCase N
Table/View? T
Default value ?
Char Type ?
IdCol Type ?
UDT Name udt_int
Temporal ?
Column Dictionary Name myint
Column SQL Name myint
Column Name UEscape
Dictionary Title
SQL Title
Title UEscape

```

Example: HELP TABLE with Temporal Columns

Assume the following table definition.

```

CREATE MULTISET TABLE department (
  deptname    VARCHAR(10) CHARACTER SET UNICODE,
  deptno      INTEGER NOT NULL UNIQUE,
  avg_salary   INTEGER,
  dept_duration PERIOD(DATE) AS VALIDTIME,
  dept_tran    PERIOD(TIMESTAMP(6) WITH TIME ZONE)
              TRANSACTIONTIME)
PRIMARY INDEX (deptno);

```


The following example shows the HELP TABLE report for the *department* table using BTEQ.

```

HELP TABLE department;
*** Help information returned. 5 rows.
Column Name deptname
                Type CV
                Comment ?
                Nullable Y
                Format X(10)
                Title ?
                Max Length      10
                Decimal Total Digits ?
                Decimal Fractional Digits ?
                Range Low      ?
                Range High     ?
                UpperCase N
                Table/View? T
                Default value ?
                Char Type  2
                IdCol Type ?
                UDT Name ?
                Temporal N
                Column Dictionary Name deptname
                Column SQL Name deptname
                Column Name UEscape
                Dictionary Title
                SQL Title
                Title UEscape
                Column Name deptno
                Type I
                Comment ?
                Nullable N
                Format -(10)9
                Title ?
                Max Length      4
                Decimal Total Digits ?
                Decimal Fractional Digits ?
                Range Low      ?
                Range High     ?
                UpperCase N
                Table/View? T
                Default value ?
                Char Type  ?
                IdCol Type ?

```

```

        UDT Name ?
        Temporal N
Column Dictionary Name deptno
    Column SQL Name deptno
    Column Name UEscape
    Dictionary Title
    SQL Title
    Title UEscape
    Column Name avg_salary
        Type I
        Comment ?
        Nullable Y
        Format -(10)9
        Title ?
        Max Length          4
    Decimal Total Digits ?
    Decimal Fractional Digits ?
        Range Low          ?
        Range High         ?
        UpperCase N
    Table/View? T
    Default value ?
        Char Type ?
        IdCol Type ?
        UDT Name ?
    Column Constraint ?
        Char Type ?
        IdCol Type ?
        UDT Name ?
        Temporal ?
Column Dictionary Name avg_salary
    Column SQL Name avg_salary
    Column Name UEscape
    Dictionary Title
    SQL Title
    Title UEscape
    Column Name dept_duration
        Type PD
        Comment ?
        Nullable N
        Format YY/MM/DD
        Title ?
        Max Length          8
    Decimal Total Digits ?

```

```

Decimal Fractional Digits ?
    Range Low ?
    Range High ?
    UpperCase N
    Table/View? T
    Default value N
    Char Type ?
    IdCol Type ?
    UDT Name ?
    Temporal Y
Column Dictionary Name dept_duration
Column SQL Name dept_duration
Column Name UEscape
Dictionary Title
SQL Title
Title UEscape
Column Name dept_tran
    Type PM
    Comment ?
    Nullable N
    Format YYYY-MM-DDBHH:MI:SS.S(6)Z
    Title ?
    Max Length 24
Decimal Total Digits ?
Decimal Fractional Digits 6
    Range Low ?
    Range High ?
    UpperCase N
    Table/View? T
    Default value ?
    Char Type ?
    IdCol Type ?
    UDT Name ?
    Temporal Y
Column Dictionary Name dept_tran
Column SQL Name dept_tran
Column Name UEscape
Dictionary Title
SQL Title
Title UEscape

```

Example: HELP TABLE with a JSON Auto Column

Here is the table definition of MyTable for this example, where the JSON column c is defined with the AUTO COLUMN option.

```
CREATE TABLE TEST.MyTable (
  a INTEGER,
  b JSON(16776192) INLINE LENGTH 4096 CHARACTER SET LATIN,
  c JSON(16776192) INLINE LENGTH 4096 CHARACTER SET LATIN AUTO COLUMN)
PRIMARY INDEX ( a );
```

This statement displays information for the table MyTable.

```
HELP TABLE MyTable;
```

Below is the abbreviated HELP TABLE output for selected attributes of column c.

Column Name	c
Type	JN
Nullable	Y
Format	X(64000)
Max Length	16,776,192
Table/View?	T
Inline Length	4,096
Auto Column	Y

HELP VOLATILE TABLE

Displays the attributes for the requested volatile table.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must either own the volatile table or have at least one privilege on it.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against the specified volatile table.

HELP VOLATILE TABLE Syntax

```
HELP VOLATILE TABLE [ volatile_table_name ] [;]
```

HELP VOLATILE TABLE Syntax Elements

volatile_table_name

Name of the volatile table for which you want help information.

If you do not specify the name of a volatile table, the systems returns a list of all volatile tables in the current session.

Usage Notes

Volatile Table Attributes

Because the Session ID and Creator Name are identical for all volatile tables created within a given session, these attributes are not provided in the HELP VOLATILE TABLE report. For a description of HELP VOLATILE TABLE attributes, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184. The SHOW TABLE statement (see [SHOW object](#)) returns the following attributes.

- Protection
- Commit Option
- Transaction Log Option

Example: HELP VOLATILE TABLE

Suppose you need to know the Table ID of the volatile table *sales_temp*. To determine this information, you enter the following request.

```
HELP VOLATILE TABLE sales_temp;
```

The result shows that the Table ID for this table is 40C00E000000.

```
Table Name Sales_temp
Table Id 40C00E000000
Table Dictionary Name Sales_temp
Table SQL Name Sales_temp
Table Uescape ?
```

Related Information

- CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- SHOW object in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184

SHOW TABLE

Displays the most recent SQL create text.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

To use SHOW TABLE, you must have one of the following privileges:

- Any privilege on the table, or any privilege on the database containing the table.
- At least one privilege on the DBC.TVM table.

SHOW TABLE Syntax

```
SHOW [IN XML] [TEMPORARY] TABLE [ database_name. | user_name. ] table_name [;]
```

SHOW TABLE Syntax Elements

IN XML

Generates an XML document of the information required to recreate the table. See [Example: SHOW TABLE in XML Format](#).

TEMPORARY

The table for which the most recent SQL create text is to be reported is a materialized global temporary table.

If you specify TEMPORARY, *table_name* must specify a global temporary table.

If you do not specify TEMPORARY, and the table is a global temporary table, the base global temporary table is shown.

table_name

Name of the table for which to display the most recent SQL create text.

Usage Notes

Exceptions to the SHOW TABLE Table Definition

SHOW TABLE displays a standardized CREATE TABLE statement that would create the named table with the following exceptions.

Because a standard display form is used, the result might not be identical to the text used to create the table. The SHOW TABLE display is such that if the table were dropped and created using the SHOW TABLE output, the result would be a table with structure identical to the one shown.

Table Modifications, Indexes, and SHOW TABLE

If the table has been modified using the ALTER TABLE or CREATE INDEX statements, modifications are reflected in the CREATE TABLE request displayed.

SHOW TABLE displays all index and constraint information for the table. There is an upper limit of 31,744 characters that SHOW TABLE can display.

Column Grouping, Partitioning, and SHOW TABLE

If a column grouping is output as part of the COLUMN clause for any of the following cases, columns and multicolumn groups are separated by commas, with a space following a comma, and ordered by the field ID of the first column of each column partition and, within a column partition, by the field IDs of the columns in the column partition. An AUTO COMPRESS or NO AUTO COMPRESS option is preceded by a space. A COLUMN or ROW format keyword is not followed by a space for a column group. ALL BUT is followed by a space.

For a column-partitioned table, the SQL text returned by SHOW TABLE includes a PARTITION BY clause with a COLUMN clause. Grouping, if any, is included in the COLUMN clause, not in the column list.

Column Autocompression and SHOW TABLE

If all of the column partitions are single-column partitions with system-determined format and NO AUTO COMPRESS, Vantage reports COLUMN NO AUTO COMPRESS for the COLUMN clause.

If all of the column partitions are single-column partitions except for one that is multicolumn, and all column partitions have system-determined format and AUTO COMPRESS, Vantage reports the shorter in length between COLUMN AUTO COMPRESS ALL BUT ((*list_of_columns_in_the_multicolumn_partition*)) or COLUMN AUTO COMPRESS (*list_of_the_columns_in_the_single-column_partitions*) for the COLUMN clause.

If all the column partitions are single-column partitions except for one that is multicolumn and all column partitions have system-determined format and NO AUTO COMPRESS, Vantage reports the shorter in length of COLUMN NO AUTO COMPRESS ALL BUT ((*list_of_columns_in_the_multicolumn_partition*)) or COLUMN NO AUTO COMPRESS (*list_of_the_columns_in_the_single-column_partitions*) for the COLUMN clause.

If none of the preceding cases apply and at least one, but not all, of the column partitions are single-column partitions with system-determined format and AUTO COMPRESS, Vantage reports COLUMN ALL BUT ((*list_of_single_columns*) or (*list_of_multiple_columns*) or both with any applicable options) that specifies all the column partitions except for the single-column partitions with system-determined format and AUTO COMPRESS for the COLUMN clause.

If none of the preceding cases apply and at least one, but not all, of the column partitions are single-column partitions with system-determined format, Vantage reports the shorter in length of the following for the COLUMN clause:

- COLUMN AUTO COMPRESS ALL BUT ((*list_of_single_columns*) or (*list_of_multiple_columns*) with any applicable options), where the column grouping specifies all the column partitions except for the single-column partitions with system-determined format and AUTO COMPRESS.
- COLUMN NO AUTO COMPRESS ALL BUT ((*list_of_single_columns*) or (*list_of_multiple_columns*) with any applicable options), where the column grouping specifies all the column partitions except for the single-column partitions with system-determined format and NO AUTO COMPRESS.

If none of the preceding cases apply, Vantage reports COLUMN ((*list_of_single_columns*) or (*list_of_multiple_columns*) with any applicable options) that specifies all the column partitions for the COLUMN clause.

If none of the preceding cases apply, Vantage reports the shorter in length of the following for the COLUMN clause, where the grouping clause specifies all the column partitions:

- COLUMN AUTO COMPRESS (*list_of_single_columns*) or (*list_of_multiple_columns*) or both with any applicable options).
- COLUMN NO AUTO COMPRESS (*list_of_single_columns*) or (*list_of_multiple_columns*) or both with any applicable options).

Derived Period Columns and SHOW TABLE

Derived period columns are output as PERIOD FOR (*begin_column*, *end_column*). For a temporal table with a derived period VALIDTIME or TRANSACTIONTIME column, AS VALIDTIME or AS TRANSACTIONTIME follows the derived period column output, as appropriate. See *Teradata Vantage™ - Temporal Table Support*, B035-1182 and *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186.

Examples

Example: SHOW TABLE

If a SHOW TABLE request is entered for a table that has been modified, Vantage returns the original CREATE TABLE SQL text, including all current modifications.

For example, this CREATE TABLE statement defines the table.


```
CREATE TABLE personnel.emp_bonus (
  emp_no    SMALLINT FORMAT '9(5)'
            CHECK (emp_no BETWEEN 10001 AND 32001) NOT NULL,
  bonus_no  SMALLINT FORMAT 'Z9'
            CHECK (bonus_no BETWEEN 0 and 99) NOT NULL,
  bonus_amt DECIMAL (6,2)
            CHECK (bonus_amt BETWEEN 1.00 AND 5000.00))
UNIQUE PRIMARY INDEX (emp_no);
```

Then, this ALTER TABLE statement subsequently modifies the emp_bonus table.

```
ALTER TABLE emp_bonus
  ADD dept_no SMALLINT;
```

When you submit a SHOW TABLE request for the emp_bonus table, the system returns the following CREATE TABLE SQL text. The DeptNo column is included in the CREATE TABLE SQL text. The report also displays default specifications, such as FALLBACK.

```
SHOW TABLE personnel.emp_bonus;
CREATE SET TABLE personnel.emp_bonus, FALLBACK (
  emp_no SMALLINT FORMAT '9(5)' CHECK (emp_no BETWEEN 10001
                                     AND      32001)
                                     NOT NULL,
  bonus_no SMALLINT FORMAT 'Z9' CHECK (bonus_no BETWEEN 0
                                     AND      99)
                                     NOT NULL,
  bonus_amt DECIMAL (6,2) CHECK (bonus_amt BETWEEN 1.00
                                     AND      5000.00),
  dept_no SMALLINT)
UNIQUE PRIMARY INDEX ( emp_no );
```

Example: SHOW TABLE with Column-Level Named Constraints

The request in this example names constraints at the column level.

```
CREATE TABLE good_1 (
  column_1 INTEGER NOT NULL CONSTRAINT primary_1 PRIMARY KEY,
  column_2 INTEGER NOT NULL CONSTRAINT unique_1 UNIQUE,
  column_3 INTEGER CONSTRAINT check_1 CHECK (column_3 > 0),
  column_4 INTEGER CONSTRAINT reference_1 REFERENCES parent_1);
```

Now perform the following SHOW TABLE request on good_1.

```
SHOW TABLE good_1;
```

The request returns the following CREATE TABLE SQL text.

```
CREATE SET TABLE personnel.good_1, NO FALLBACK,
  NO BEFORE JOURNAL, NO AFTER JOURNAL (
    column_1 INTEGER NOT NULL,
    column_2 INTEGER NOT NULL,
    column_3 INTEGER,
    column_4 INTEGER,
    CONSTRAINT check_1 CHECK ( column_3 > 0 ),
    CONSTRAINT reference_1 FOREIGN KEY ( column_4 )
      REFERENCES personnel.parent_1 ( f2 ))
  UNIQUE PRIMARY INDEX primary_1 ( column_1 )
  UNIQUE INDEX unique_1 ( column_2 );
```

Example: SHOW TABLE with REFERENCES Constraint

Table t1 is created with a referential constraint which specifies that FOREIGN KEY columns (f1) reference the unique primary index columns in table good_1.

```
CREATE TABLE t1 (
  f1 INTEGER NOT NULL
  f2 INTEGER)
PRIMARY INDEX (f1)
FOREIGN KEY (f1) REFERENCES good_1;
```

SHOW TABLE on t1 returns the following CREATE TABLE SQL text.

```
SHOW TABLE t1;
CREATE SET TABLE t1, NO FALLBACK,
  NO BEFORE JOURNAL, NO AFTER JOURNAL (
  f1 INTEGER NOT NULL,
  f2 INTEGER)
  PRIMARY INDEX ( f1 )
  FOREIGN KEY ( f1 ) REFERENCES personnel.good_1( column_1 );
```

Example: SHOW TABLE for a Table Created without Specifying a MERGEBLOCKRATIO Option

If you do not specify the MERGEBLOCKRATIO option when you create a table, Vantage returns the SQL create text for the table as if you had specified the DEFAULT MERGEBLOCKRATIO option.

For example, suppose you create a table named emp_table, but do not specify the MERGEBLOCKRATIO option. Then, you submit a SHOW TABLE request for emp_table.

```
SHOW TABLE emp_table;
```

The SHOW TABLE output is as follows.

```
CREATE SET TABLE emp_table, NO FALLBACK, NO BEFORE JOURNAL,
      NO AFTER JOURNAL,CHECKSUM = DEFAULT, DEFAULTMERGEBLOCKRATIO,
      MAP = TD_MAP1
      ( emp_no INTEGER )
      PRIMARY INDEX ( emp_no );
```

Example: SHOW TABLE for a User with a Default Character Type of Latin

This CREATE TABLE request creates the following table for a user whose default character type is Latin.

```
CREATE TABLE kanji_user.table_1,
      NO FALLBACK,
      NO BEFORE JOURNAL,
      NO AFTER JOURNAL
      (
        clatin      CHARACTER(5),
        ckanji1     CHARACTER(5) CHARACTER SET UNICODE,
        cgraphic    CHARACTER(5) CHARACTER SET GRAPHIC,
        ckanjisjis  CHARACTER(5) CHARACTER SET KANJISJIS,
        cunicode    CHARACTER(5))
      PRIMARY INDEX (clatin);
```

A SHOW TABLE request for this table returns the following CREATE TABLE SQL text, as expected.

```
CREATE TABLE kanji_user.table_1,
      NO FALLBACK,
      NO BEFORE JOURNAL,
      NO AFTER JOURNAL (
        clatin CHARACTER(5) CHARACTER SET LATIN,
```

```

    ckanji1 CHARACTER(5) CHARACTER SET UNICODE,
    cgraphic CHARACTER(5) CHARACTER SET GRAPHIC,
    ckanjisjis CHARACTER(5) CHARACTER SET KANJISJIS,
    cunicode CHARACTER(5) CHARACTER SET LATIN)
PRIMARY INDEX ( clatin );

```

Example: SHOW TABLE for Algorithmic Compression

Assume that you have defined algorithmic compression on a set of table columns. The SHOW TABLE output for such a table returns the names of the UDFs that contain the compression and decompression algorithms.

Assume table t1 is created with the following definition.

```

CREATE TABLE t1 (
  col_1 INTEGER,
  col_2 CHARACTER(10) COMPRESS ALGCOMPRESS scsu_comp
                        ALGDECOMPRESS scsu_decomp ('abc', 'efg'));

```

You then submit a SHOW TABLE request on t1. Vantage returns the following SQL create text for this request.

```

CREATE SET TABLE  user_name.t1 ,NO FALLBACK ,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT
(
col_1 INTEGER,
col_2 CHARACTER(10)
CHARACTER SET LATIN NOT CASESPECIFIC
COMPRESS
ALGCOMPRESS scsu_comp
ALGDECOMPRESS scsu_decomp ('abc','efg'))
PRIMARY INDEX ( col_1 );

```

Assume you create table t2 with the following definition.

```

CREATE TABLE t2 (
  col_1 INTEGER,
  col_2 CHARACTER(10));

```

You then submit the following ALTER TABLE request on t2 to modify its definition to include algorithmic compression on col_2.

```
ALTER TABLE t2
ADD col_2 CHARACTER(10) COMPRESS ALGCOMPRESS scsu_comp
                        ALGDECOMPRESS scsu_decomp ('abc', 'efg');
```

A SHOW TABLE request for table t2 returns the following SQL create text.

```
CREATE SET TABLE  user_name.t2, NO FALLBACK,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT
(
    col_1 INTEGER,
    col_2 CHARACTER(10)
        CHARACTER SET LATIN NOT CASESPECIFIC
        COMPRESS
        ALGCOMPRESS scsu_comp
        ALGDECOMPRESS scsu_decomp ('abc      ', 'efg      '))
PRIMARY INDEX ( col_1 );
```

Example: SHOW TABLE for a Column-Partitioned Table or Join Index

The following rules apply to the report returned by a SHOW TABLE request for a column-partitioned table or a SHOW JOIN INDEX request made on a column-partitioned join index.

For a partitioned table or join index, Vantage includes an ADD option for a partitioning level if the level has column partitioning or if the number of defined partitions for a row partitioning level is less than the maximum number of partitions for the level and the level is not the first level that has row partitioning. The output is the same as for a row-partitioned table or join index if the object only has row partitioning and an ADD option was not specified for the level and the number of row partitions currently defined is the same as the maximum number of row partitions for the level.

The rules for the ADD option are as follows.

- ADD 0 is not reported for any row partitioning level.
- ADD is not reported for the first row partitioning level.
- ADD is reported for a row partitioning level if the number of defined partitions is less than the maximum number of partitions and it is not the first row partitioning level.
- ADD is reported for a column partitioning level, even if it specifies ADD 0.

If ADD is reported, the value is equal to the maximum number of partitions minus the currently defined number of partitions for that partitioning level.

For a column-partitioned table or join index, the output is the same except the PARTITION BY clause includes a COLUMN clause. Grouping is included in the COLUMN clause, not in the column definition

list for a table or the select list for a join index. The column grouping in the COLUMN clause is reported as follows.

If all the column partitions are single-column partitions with system-determined COLUMN or ROW format and without a NO AUTO COMPRESS specification, Vantage does not report a column grouping following the COLUMN clause.

If all the column partitions are single-column partitions except for 1 that is multicolumn, and all column partitions have system-determined COLUMN or ROW format without a NO AUTO COMPRESS specification, Vantage reports the shorter in terms of characters of these bullet items following the COLUMN clause.

- COLUMN ALL BUT ((*multicolumn_partition_column_list_separated_by_commas_and_ordered_by_field_ID*))
or
- COLUMN (*columns_in_single-column_partitions_separated_by_commas_and_ordered_by_field_ID*)

For example,

```
COLUMN ALL BUT ((d, p, z))
COLUMN (a, b, c, g)
```

If the partitioning is not that of the previous bullet, and at least one, but not all, of the column partitions are single column partitions with system-determined COLUMN or ROW format without the NO AUTO COMPRESS option that specifies all the column partitions except for the single-column partitions with system-determined COLUMN or ROW format without the NO AUTO COMPRESS option is included following the COLUMN clause.

Vantage reports the different forms with entries separated by commas and ordered by field ID.

- COLUMN ALL BUT (*single_columns_list_with_options*)
or
- COLUMN ALL BUT (*multiple_columns_list_with_options*)
or
- COLUMN ALL BUT (*single_columns_list_with_options* , (*multiple_columns_list_with_options*))

For example,

```
COLUMN ALL BUT (ROW d, (i, t), k NO AUTO COMPRESS,
                COLUMN(m, s, u, v))
COLUMN ALL BUT (COLUMN(j, m, o))
COLUMN ALL BUT (ROW(e, j, z) NO AUTO COMPRESS)
```

Otherwise, if none of the preceding cases, Vantage reports one of the following forms with entries separated by commas and ordered by the field ID of the first column of each column partition. Within a column partition, Vantage orders the entries by the field IDs of the partition columns.

- COLUMN (*single_columns_list_with_options*)
- or
- COLUMN (*multiple_columns_list_with_options*)
- or
- COLUMN (*single_columns_list_with_options*, (*multiple_columns_list_with_options*))

For example,

```
COLUMN (ROW d, (i, t), k NO AUTO COMPRESS, COLUMN(m, s, u, v))
```

This example creates column-partitioned table t1 with the following definition.

```
CREATE TABLE t1 (
  a INTEGER,
  b INTEGER,
  c INTEGER,
  d INTEGER,
  e INTEGER,
  f INTEGER,
  g INTEGER,
  h INTEGER,
  i INTEGER,
  j INTEGER,
  k INTEGER,
  l INTEGER,
  m INTEGER,
  n INTEGER,
  o INTEGER,
  p INTEGER,
  q INTEGER,
  r INTEGER,
  s INTEGER,
  t INTEGER,
  u INTEGER,
  v INTEGER,
  w INTEGER,
  x INTEGER,
  y INTEGER,
  z INTEGER)
```

```

NO PRIMARY INDEX
PARTITION BY COLUMN ALL BUT (a, b, (g, d), ROW(s, t, j),
                             h NO AUTO COMPRESS, x) ADD 65509;

```

Assume that the value of AutoCompressDefault has changed to 2. A SHOW TABLE request for t1 returns the following information.

```

SHOW TABLE t1;
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.

-----
CREATE MULTiset TABLE  user_name.t1 ,NO FALLBACK ,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT,
DEFAULT MERGEBLOCKRATIO,
MAP = TD_MAP1
(
a INTEGER,
b INTEGER,
c INTEGER,
d INTEGER,
e INTEGER,
f INTEGER,
g INTEGER,
h INTEGER,
i INTEGER,
j INTEGER,
k INTEGER,
l INTEGER,
m INTEGER,
n INTEGER,
o INTEGER,
p INTEGER,
q INTEGER,
r INTEGER,
s INTEGER,
t INTEGER,
u INTEGER,
v INTEGER,
w INTEGER,
x INTEGER,
y INTEGER,
z INTEGER)

```



```
NO PRIMARY INDEX
PARTITION BY COLUMN AUTO COMPRESS
ALL BUT ((d, g), h NO AUTO COMPRESS, ROW(j, s, t)) ADD 65509;
```

Columns a, b, and x are not included in the grouping clause for the COLUMN clause because they are single-column partitions by default with system-determined format and autocompression. The column partitions and columns for a column partition are in the same order as in the definition list.

Example: SHOW TABLE in XML Format

The table is defined as follows:

```
CREATE MULTiset TABLE YourDB.orders ,NO FALLBACK ,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT,
DEFAULT MERGEBLOCKRATIO,
MAP = TD_MAP1
(
  o_orderkey INTEGER,
  o_date DATE FORMAT 'yyyy-mm-dd',
  o_status CHAR(1) CHARACTER SET LATIN CASESPECIFIC,
  o_custkey INTEGER,
  o_totalprice DECIMAL(13,2),
  o_orderpriority CHAR(21) CHARACTER SET LATIN CASESPECIFIC,
  o_clerk CHAR(16) CHARACTER SET LATIN CASESPECIFIC,
  o_shippriority INTEGER,
  o_comment VARCHAR(79) CHARACTER SET LATIN CASESPECIFIC)
UNIQUE PRIMARY INDEX ( o_orderkey );
```

The XML document generated by a SHOW IN XML TABLE request reports all of the information required to recreate the table:

```
<?xml version="1.0" encoding="UTF-8"
standalone="no" ?><TeradataDBObjectSet version="1.0" xmlns="http://
schemas.teradata.com/dbobject" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://schemas.teradata.com/dbobject http://
schemas.teradata.com/dbobject/DBObject.xsd"><Table afterJournal="No"
baseClass="Table" beforeJournal="No" checkSumLevel="Default"
dbName="YourDB" fallback="false" kind="Multiset" map="TD_MAP1"
map_kind="contiguous" mergeBlockRatio="Default" name="orders" objId="0:3650"
objVer="4" systemVersioned="false">
<ColumnList><Column name="o_orderkey" nullable="true"
```

```

order="1"><DataType><Integer/></DataType></Column>
<Column format="yyyy-mm-dd" name="o_date" nullable="true"
order="2"><DataType><Date/></DataType></Column>
<Column name="o_status" nullable="true" order="3"><DataType><Char
casespecific="true" charset="LATIN" length="1" uppercase="false"
varying="false"/></DataType></Column>
<Column name="o_custkey" nullable="true" order="4"><DataType><Integer/></DataType></Column>
<Column name="o_totalprice" nullable="true" order="5"><DataType><Decimal
precision="13" scale="2"/></DataType></Column>
<Column name="o_orderpriority" nullable="true" order="6"><DataType><Char
casespecific="true" charset="LATIN" length="21" uppercase="false"
varying="false"/></DataType></Column><Column name="o_clerk" nullable="true"
order="7"><DataType><Char casespecific="true" charset="LATIN" length="16"
uppercase="false" varying="false"/></DataType></Column>
<Column name="o_shippriority" nullable="true" order="8"><DataType><Integer/></
DataType></Column>
<Column name="o_comment" nullable="true" order="9"><DataType><Char
casespecific="true" charset="LATIN" length="79" uppercase="false"
varying="true"/></DataType></Column></ColumnList>
<Indexes><PrimaryIndex unique="true"><ColumnList><Column name="o_orderkey"
order="1"/></ColumnList></PrimaryIndex></Indexes>
<SQLText><![CDATA[CREATE MULTISET TABLE YourDB.orders ,NO FALLBACK ,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT,
DEFAULT MERGEBLOCKRATIO,
MAP = TD_MAP1
(
o_orderkey INTEGER,
o_date DATE FORMAT 'yyyy-mm-dd',
o_status CHAR(1) CHARACTER SET LATIN CASESPECIFIC,
o_custkey INTEGER,
o_totalprice DECIMAL(13,2),
o_orderpriority CHAR(21) CHARACTER SET LATIN CASESPECIFIC,
o_clerk CHAR(16) CHARACTER SET LATIN CASESPECIFIC,
o_shippriority INTEGER,
o_comment VARCHAR(79) CHARACTER SET LATIN CASESPECIFIC)
UNIQUE PRIMARY INDEX ( o_orderkey )]]></SQLText></Table>
<Environment><Server dbRelease="16.10" dbVersion="16.10" hostName="localhost"/
><User userId="00000704" userName="User"/><Session charset="UTF8"
dateTime="2017-03-24T16:19:32"/></Environment>
</TeradataDBObjectSet>

```

Example: SHOW TABLE with a Derived Period Column

This is an example of SHOW TABLE output for the following table definition which includes a derived period column.

```
CREATE TABLE employee (
    eid INTEGER NOT NULL,
    name VARCHAR(100) NOT NULL,
    deptno INTEGER NOT NULL,
    jobstart DATE NOT NULL,
    jobend DATE NOT NULL,
    PERIOD FOR jobduration (jobstart, jobend)
) PRIMARY INDEX(eid);
```

Below is the SHOW TABLE output.

```
CREATE SET TABLE TEST.employee ,NO FALLBACK ,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT,
DEFAULT MERGEBLOCKRATIO
MAP = TD_MAP1
(
eid INTEGER NOT NULL,
name VARCHAR(100) NOT NULL,
deptno INTEGER NOT NULL,
jobstart DATE NOT NULL,
jobend DATE NOT NULL,
PERIOD FOR jobduration (jobstart, jobend)
) PRIMARY INDEX(eid);
```

Example: SHOW TABLE with an Auto Column

Following is the table definition for this example, which specifies the AUTO COLUMN option for column c.

```
CREATE TABLE MyTable (
    a INTEGER,
    b JSON(16776192) INLINE LENGTH 4096 CHARACTER SET LATIN,
    c JSON(16776192) INLINE LENGTH 4096 CHARACTER SET LATIN AUTO COLUMN)
PRIMARY INDEX ( a );
```

This statement displays information for the table MyTable.

```
SHOW TABLE MyTable;
```

Below is the SHOW TABLE output.

```
CREATE SET TABLE TEST.MyTable ,FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO,
    MAP = TD_MAP1
(
    a INTEGER,
    b JSON(16776192) INLINE LENGTH 4096 CHARACTER SET LATIN,
    c JSON(16776192) INLINE LENGTH 4096 CHARACTER SET LATIN AUTO COLUMN)
PRIMARY INDEX ( a );
```

This statement displays the table information in XML format.

```
SHOW IN XML TABLE MyTable;
```

Below is the SHOW TABLE output in XML format.

```
<?xml version="1.0" encoding="UTF-16" standalone="no" ?
><TeradataDBObjectSet version="1.0"
xmlns="http://schemas.teradata.com/dbobject"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://schemas.teradata.com/dbobject http://
schemas.teradata.com/dbobject/DBObject.xsd">
<Table afterJournal="No"
baseClass="Table"
beforeJournal="No"
checksumLevel="Default"
dbName="TEST"
fallback="false"
kind="Set"
map="TD_MAP1" map_kind="contiguous"
mergeBlockRatio="Default"
name="MyTable" objId="0:3654" objVer="1" systemVersioned="false">

<ColumnList>
<Column name="a" nullable="true" order="1">
```

```

<DataType><Integer/></DataType>
</Column>

<Column name="b" nullable="true" order="2">
<DataType><JSON charset="LATIN" inlinelength="4096" size="16776192"/>
</DataType>
</Column>

<Column autocolumn="true" name="c" nullable="true" order="3">
<DataType><JSON charset="LATIN" inlinelength="4096" size="16776192"/>
</DataType>
</Column>
</ColumnList>

<Indexes><PrimaryIndex unique="false">
<ColumnList>
<Column name="a" order="1"/></ColumnList>
</PrimaryIndex>
</Indexes>
<SQLText><![CDATA[CREATE SET TABLE TEST.MyTable ,NO
FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO,
    MAP = TD_MAP1
(
    a INTEGER,
    b JSON(16776192) INLINE LENGTH 4096 CHARACTER SET LATIN,
    c JSON(16776192) INLINE LENGTH 4096 CHARACTER SET LATIN AUTO COLUMN)
PRIMARY INDEX ( a )]]>
</SQLText>
</Table>

<Environment><Server dbRelease="16.50d.00.07"
dbVersion="16.50d.00.07dr183550" hostName="localhost"/>
<User userId="00000100" userName="DBC"/>
<Session charset="ASCII"
dateTime="2017-09-21T16:41:44"/>
</Environment>
</TeradataDBObjectSet>

```

View Statements

CREATE VIEW and REPLACE VIEW

Creates or replaces a view on a set of tables or views or both.

REPLACE VIEW redefines an existing view or, if the specified view does not exist, creates a new view with the specified name. Privileges that were granted directly on the original view are retained for the replace view definition.

For information about the temporal forms of CREATE VIEW and REPLACE VIEW, see *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186 and *Teradata Vantage™ - Temporal Table Support*, B035-1182.

ANSI Compliance

CREATE VIEW is ANSI SQL:2011-compliant.

REPLACE VIEW is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges: CREATE VIEW

To create a view, you must have the CREATE VIEW privilege.

The creator receives the DROP VIEW, INSERT, UPDATE, DELETE, and SELECT privileges on the newly created view WITH GRANT OPTION.

If a user other than the owner accesses a view, then all of the relevant privileges needed by the immediate owner of the view to access underlying tables and views must also be held by the immediate owner of the view WITH GRANT OPTION.

You must have the following privileges to update a table through a view.

User	Privileges Required
Immediate owner of view	<ul style="list-style-type: none"> • UPDATE on the view. • UPDATE on the table.
Other users	<ul style="list-style-type: none"> • UPDATE on the view. • UPDATE WITH GRANT OPTION on the table.

You must have the following privileges to delete rows from a table through a view:

User	Privileges Required
Immediate owner of view	<ul style="list-style-type: none"> • DELETE on the view. • DELETE on the table.

User	Privileges Required
Other users	<ul style="list-style-type: none"> • DELETE on the view. • DELETE WITH GRANT OPTION on the table.

You must have the following privileges to update or delete rows from a table using a search condition through a view:

User	Privileges Required
Immediate owner of view	<ul style="list-style-type: none"> • SELECT ACCESS on the view. • SELECT ACCESS on the table.
Other users	<ul style="list-style-type: none"> • SELECT ACCESS on the view. • SELECT ACCESS WITH GRANT OPTION on the objects specified in the search condition list.

Required Privileges: REPLACE VIEW

You must have the relevant privileges listed above, as well as the following additional privileges to replace a view:

- DROP VIEW privilege.
- CREATE VIEW privilege if the specified view does not exist.

Privileges Granted Automatically

When you create a new view, the following privileges are automatically granted to the view:

- DELETE
- DROP VIEW
- GRANT
- INSERT
- SELECT
- UPDATE

CREATE VIEW and REPLACE VIEW Syntax

```
{ CREATE VIEW | CV | REPLACE VIEW } [ database_name. | user_name. ] view_name
[ ( column_name [,...] ) ] AS
{ view_specification | ( view_specification ) } [;]
```

view_specification

```

[ locking_clause ... ]
[ as_of_clause ]
[ WITH { nonrecursive_with_modifier | recursive_with_modifier } [,...] ]
SELECT selection
FROM source [,...]
[ WHERE search_condition ]
[ GROUP BY group_specification [,...] ]
[ HAVING having_condition ]
[ QUALIFY qualify_condition ]
[ WITH CHECK OPTION ]
[ ORDER BY order_by_specification [,...] ]

```

locking_clause

```

LOCKING item_to_lock [ FOR | IN ] lock_type [ MODE ] [ NOWAIT ]

```

as_of_clause

```

AS OF calendar_function (
    { DATE | TIMESTAMP [ WITH TIME ZONE ] } expression [,
    system_calendar_name ] )

```

nonrecursive_with_modifier

```

query_name [ ( column_name [,...] ) ] AS ( select_expression )

```

recursive_with_modifier

```

RECURSIVE query_name [ ( column_name [,...] ) ] AS (
    seed union_specification [union_specification ...]
)

```

selection

```

[ DISTINCT | ALL ] [ TOP { n | m PERCENT } ] [ WITH TIES ]
{ * | sub_selection [,...] }

```


source

```
{ [ database_name. | user_name. ] { table_name | view_name } [ [AS]
correlation_name ] |

joined_table_source |

derived_table_source
}
```

group_specification

```
{ ordinary_grouping_set |
empty_grouping_set |
rollup_list |
cube_list |
grouping_sets_specification
}
```

order_by_specification

```
{ expression |

{ [ [ database_name. | user_name. ] table_name. ] column_name |
column_name_alias |
column_position
} [ ASC | DESC ]
}
```

item_to_lock

```
{ [ DATABASE ] { database_name | user_name } |
[ TABLE ] [ database_name. | user_name. ] table_name |
[ VIEW ] [ database_name. | user_name. ] view_name |
ROW
}
```

lock_type

```
{ ACCESS |
{ EXCLUSIVE | EXCL } |
```

```

    SHARE |
    READ [ OVERRIDE ] |
    WRITE |
    LOAD COMMITTED
  }

```

seed

```

{ SELECT | SEL } [ DISTINCT | ALL ] { * | seed_selection [,...] }
  FROM seed_source
  WHERE where_search_condition
  [ GROUP BY group_specification [,...] ]
  [ { HAVING | QUALIFY } having_qualify_search_condition ]
  [ ORDER BY order_by_specification [,...] ]

```

union_specification

```

UNION ALL { seed | recursive }

```

sub_selection

```

{ expression [ [AS] expression_alias ] |
  [ database_name. | user_name. ] table_name.*
}

```

joined_table_source

```

joined_table { [ INNER | { LEFT | RIGHT | FULL } [ OUTER ] ]
                JOIN joined_table ON search_condition |

                CROSS JOIN single_table
              }

```

derived_table_source

```

( subquery ) [AS] derived_table_name [ ( column_name [,...] ) ]

```

seed_selection

```
{ expression [ [AS] expression_alias ] |
  table_name.*
}
```

seed_source

```
{ table_name [ [AS] correlation_name ] |
  joined_table_source |
  derived_table_source
}
```

recursive

```
{ SELECT | SEL } { * | seed_selection [,...] }
FROM { implicit_join [,...] | explicit_join }
WHERE where_search_condition
```

implicit_join

```
{ query_name_specifier [...] | table_name } [ [AS] correlation_name_1 ]
```

explicit_join

```
{ { query_name | join_table_name } LEFT [ OUTER ] JOIN joined_table |
  join_table_name RIGHT [ OUTER ] JOIN { query_name | joined_table } |
  query_name INNER JOIN joined_table |
  join_table_name INNER JOIN query_name
} ON search_condition
```

query_name_specifier

```
query_name [ [AS] correlation_name_2 ]
```

CREATE VIEW and REPLACE VIEW Syntax Elements

database_name

user_name

Name of the database or user to contain *view_name* if something other than the current database or user.

view_name

Name of the new view.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

If *view_name* is not fully qualified, the default database is used.

column_name

Name of a view column. If more than one column is specified, list their names in the order in which each column is to be displayed for the view.

Note:

Views that reference a row-level security table can include columns based on row-level security constraints, but it is not required. However, the view enforces any security constraints contained in the base table whether or not they are included in the view definition.

A view can reference both row-level security tables and non-row-level security tables, but all referenced row-level security tables must contain the same security constraint columns or subsequent requests to access the view fail with an error.

locking_clause

For more information about the LOCKING clause, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

LOCKING

The type of lock to be placed on a database, table, view, or row hash. This setting will override any default lock placed on that object by the system. See *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184, and *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

item_to_lock**DATABASE**

A lock is to be placed at the database level for this view definition.

database_name

user_name

Name of the database or user that is to be locked for this view definition.

TABLE

Specifies that a lock is to be placed at the table level for this view definition.

database_name

user_name

Name of the containing database or user for *table_name* if other than the current database or user.

table_name

Name of a user base table to be locked for this view definition.

VIEW

Specifies that a lock is to be placed at the view level for this view definition.

database_name

user_name

Name of the containing database or user for *view_name* if something other than the current database or user.

view_name

Name of a view to be locked for this view definition.

ROW

Specifies that a lock is to be placed at the row hash level for this view definition.

lock_type

The type of lock to be placed when accessing base tables through this view.

MODE

An optional keyword.

NOWAIT

Specifies that if the indicated lock cannot be obtained, the request using this view should be aborted.

This option can prevent a situation where a request is waiting for resources, possibly tying up other resources in the process of waiting for a lock to become available.

as_of_clause

Specifies that the view is defined beginning at a specific point in time.

For information about temporal qualifiers, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

calendar_function

One of the embedded services system calendar functions.

For information about the embedded services calendar functions, see *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211.

DATE *expression***DATE AT TIME ZONE *expression***

A DATE or DATE AT TIME ZONE expression argument for the specified embedded services system calendar function.

You can specify a SELECT CURRENT_DATE or SELECT CURRENT_DATE AT TIME ZONE argument instead of an explicit DATE or DATE AT TIME ZONE expression argument.

TIMESTAMP *expression***TIMESTAMP WITH TIME ZONE *expression***

A TIMESTAMP or TIMESTAMP WITH TIME ZONE expression argument for the specified embedded services system calendar function.

You can specify a SELECT CURRENT_TIMESTAMP or SELECT CURRENT_TIMESTAMP AT TIME ZONE argument instead of an explicit TIMESTAMP or TIMESTAMP WITH TIME ZONE expression argument.

system_calendar_name

An optional system calendar name used to define a different calendar than the one assigned to the current session.

For the definitions of the system calendars, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

You do not need to specify a system calendar name for week-related specific times because, for example, the previous Monday for a specified date is the same for all of the calendars.

nonrecursive_with_modifier

query_name

Name of the query.

column_name

Name of a column in the named query definition.

select_expression

Nonrecursive SELECT statement that retrieves the row data to store in the named query.

You can specify an expression that returns a UDT in a column list only if the transform group has a fromsql routine. The system automatically converts the expression from its UDT value to the external type using the fromsql routine before returning the result to a client application.

You can specify a row-level security constraint column in the select list of a SELECT request. However, the column cannot be specified as part of an arithmetic expression in the select list. The value returned for the column is the coded value for the row-level security constraint from the row.

recursive_with_modifier

A RECURSIVE named query that can refer to itself in the query definition. The named query list consists of at least one nonrecursive, or seed, statement and at least one recursive statement.

query_name

Recursive query name.

column_name

Name of a column in the named query definition.

seed

The seed statement is a nonrecursive SELECT statement that retrieves row data from other tables to store in the named query.

DISTINCT

Only one row is returned from any set of duplicates that result from a given expression list.

Two rows are considered duplicates only if each value in one is equal to the corresponding value in the other.

ALL

Return all rows, including duplicates, in the results of the expression list. This is the default value.

Return all columns of all tables referenced in the FROM clause of the seed statement.

When qualified by *table_name*, return all columns of *table_name* only.

seed_selection***expression***

Any valid SQL expression, including scalar UDFs.

expression_alias_name

Alias for the expression.

table_name

Name of a table, derived table, or view.

table_name.***** in the select list can define the table from which rows are to be returned when two or more tables are referenced in the FROM clause.

FROM

Introduction to the names of one or more tables, views, or derived tables from which *expression* is to be derived.

The FROM clause in a seed statement cannot specify the TABLE option.

seed_source***table_name***

Name of a single table, derived table, table UDF, or view referred to in the FROM clause.

correlation_name

Alias for the table referenced in the FROM clause.

where_search_condition

Conditional search expression that must be satisfied by the row or rows returned by the seed statement.

If you specify the value for a row-level security constraint in a search condition, that value must be expressed in encoded form.

group_specification***ordinary_group_set***

Column expression by which the rows returned by the seed statement are grouped.

The expression cannot group result rows that have a LOB, ARRAY, or VARRAY type.

ordinary_grouping_set falls into three general categories:

- *column_name*
- *column_position*
- *column_expression*

empty_grouping_set

Contiguous LEFT PARENTHESIS, RIGHT PARENTHESIS pair with no argument. You use this syntax to request a grand total of the computed group totals.

rollup_list

ROLLUP expression that reports result rows in a single dimension with one or more levels of detail.

The expression cannot group result rows that have a LOB, ARRAY, or VARRAY type.

cube_list

CUBE expression that reports result rows in multiple dimensions with one or more levels of detail.

The expression cannot group result rows that have a LOB, ARRAY, or VARRAY type.

grouping_sets_specification

GROUPING SETS expression that reports result rows in one of two ways:

- As a single dimension, but without a full ROLLUP.
- As multiple dimensions, but without a full CUBE.

having_qualify_search_condition

Conditional expression that must be satisfied by the result rows.

If you specify the value for a row-level security constraint in this search condition, that value must be expressed in encoded form.

order_by_specification

Order in which result rows are to be sorted.

expression

Expression in the SELECT expression list of the seed statement, either by name or by means of a constant that specifies the numeric position of the expression in the expression list.

column_name

Names of columns used in the ORDER BY clause in the SELECT statement. These can be ascending or descending.

column_name_alias

Column name alias specified in the select expression list of the query for the column on which the result rows are to be sorted.

If you specify a *column_name_alias* to sort by, then that alias cannot match the name of any column that is defined in the table definition for any table referenced in the FROM clause of the query whether that column is specified in the select list or not. The system always references the underlying physical column having the name rather than the column that you attempt to reference using that same name as its alias.

You can specify the sort column by *column_position* value within the select list for the query.

column_position

Numeric position of the columns specified by the ORDER BY clause. These can be ascending or descending.

ASC

Results are to be ordered in ascending sort order.

If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session.

The default order is ASC.

DESC

Results are to be ordered in descending sort order.

If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.

union_specification

UNION ALL

Operator that adds results of iterative operations to the named query.

recursive

The recursive statement is a SELECT statement that retrieves row data from a join of the named query and other tables.

You cannot include NORMALIZE in a recursive statement of a recursive query.

*

All columns of all tables referenced in the FROM clause of the recursive statement be returned.

When qualified by *table_name*, specifies that all columns of *table_name* only are to be returned.

seed_selection

Any valid SQL expression, with these exceptions:

- Aggregate functions
- Ordered analytical functions

AS

Optional introduction to *expression_alias_name*.

expression_alias_name

Alias for the *expression*.

table_name

Name of the named query or the name of a table or view.

table_name.* in the select list can define the table from which rows are to be returned when two or more tables are referenced in the FROM clause.

FROM

Introduction to the named query and one or more tables or views from which *expression* is to be derived.

The FROM clause in a recursive statement cannot specify the TABLE option.

where_search_condition

A conditional search expression that must be satisfied by the row or rows returned by the recursive statement.

A value you specify for a row-level security constraint in a search condition must be in encoded form.

implicit_join

This option enables the FROM clause of the recursive statement to specify the name of the RECURSIVE query and one or more single table references, creating an implicit inner join.

query_name_specifier***query_name***

Named query referred to in the FROM clause.

correlation_name_2

Alias for the query name referenced in the FROM clause.

table_name

Name of a single table or view referred to in the FROM clause.

correlation_name_1

Alias for the table name referenced in the FROM clause.

explicit_join

Options for joined tables enable the FROM clause of the seed statement to specify that multiple tables be joined in explicit ways, described as follows.

query_name

Named query referred to in the FROM clause.

join_table_name

Name of a joined table.

INNER JOIN

Join in which qualifying rows from one table are combined with qualifying rows from another table according to a join condition.

Inner join is the default join type.

LEFT OUTER

Outer join with the table that was listed first in the FROM clause.

In a LEFT OUTER JOIN, the rows from the left table that are not returned in the result of the inner join of the two tables are returned in the outer join result and extended with nulls.

RIGHT OUTER

Outer join with the table that was listed second in the FROM clause.

In a RIGHT OUTER JOIN, the rows from the right table that are not returned in the result of the inner join of the two tables are returned in the outer join result and extended with nulls.

JOIN

Introduction to the name of the second table to participate in the join.

joined_table

Name of the joined table.

ON *search_condition*

One or more conditional expressions that must be satisfied by the result rows.

A value you specify for a row-level security constraint in a search condition must be in encoded form.

An ON condition clause is required if the FROM clause specifies an outer join.

selection

You can delimit the SELECT clause with parentheses. If you do, you must specify both opening and closing parentheses or the system aborts the request.

You can invoke an SQL UDF at any point of the SELECT clause of a view definition. The rules are the same as those for a SELECT request.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for complete documentation of the SELECT clause.

DISTINCT

Only one row is to be returned from any set of duplicates that might result from a given *expression list*.

Two rows are considered duplicates only if each value in one is equal to each corresponding value in the other.

You cannot specify a DISTINCT operator if the view definition also includes a TOP *n* or TOP *m* PERCENT specification.

ALL

All rows, including duplicates, are to be returned in the results of the expression list.

This is the default value.

TOP

Returns the top-ranked rows to from the SELECT operation.

You cannot specify a TOP *n* operator if the view definition also specifies any of the following:

- DISTINCT operator
- SAMPLE clause
- QUALIFY clause
- Subquery, that is, a subquery within a view definition cannot contain a TOP operator.

Views that specify a TOP operator are not updatable.

You can also specify an ORDER BY clause with the TOP operator. Otherwise, ORDER BY clauses are not valid within view definitions.

The sort expression for the ORDER BY clause cannot contain BLOBs, CLOBs, UDTs, columns with Period or Geospatial types, or ordered analytic functions.

TOP performs as well or better as the ordered analytic functions QUALIFY RANK() or QUALIFY ROW_NUMBER() that can be used in a similar fashion.

n

A DECIMAL or an INTEGER value for the number of rows to return from the SELECT operation.

Note that this can be more limiting than the TOP *n* clause you can specify with DML requests, depending on whether you specify an ORDER BY clause or not. If you do not specify an ORDER BY clause, then a TOP *n* clause only specifies that *any n* base table rows be returned, not the TOP *n*.

***m* PERCENT**

Returns only *m* percent of rows from the SELECT operation, where *m* can be either a DECIMAL or an INTEGER value.

If (*m* *COUNT(*)/100.0) does not evaluate to an integer value, then the system rounds it up to the next highest integer value.

For example, if (*m* *COUNT(*)/100.0) evaluates to 0.1, then it is rounded up to 1.

If (*m* *COUNT(*)/100.0) evaluates to 9.2, then it is rounded up to 10.

Vantage only applies this rounding up to the first 15 digits following the decimal point. For example, if (*m* *COUNT(*)/100.0) evaluates to 0.0000000000000001, then the system does not round it up to 1.

WITH TIES

All qualified rows having the same value for the ORDER BY fields are to be included in the results set.

This option only works if you also specify an ORDER BY clause; otherwise, the system ignores it.

If you do not specify the WITH TIES option, then the system returns only the first *n* or *m* PERCENT qualified rows.

For example, suppose you specify TOP 2 WITH TIES and ORDER BY OrderValue. If the top customer for this field has an OrderValue of 300,000 USD, but there are two additional customers with the same next highest OrderValue of 295,000 USD, then all 3 are returned in the result set.

If you did not specify WITH TIES, then only the first of the two customers encountered in the result list having an OrderValue of 295,000 USD would be returned.

*

All columns from all tables referenced in the FROM clause are to be returned.

When qualified by *table_name*, * specifies that all columns of only the user base table or view specified by *table_name* are to be returned.

View columns are explicitly enumerated when views are defined; therefore, if a table is changed after a view is defined, those changes will not appear if the SELECT * construct is used.

sub_selection

A valid SQL expression.

You can specify both aggregate and arithmetic operators in a view definition expression.

You cannot specify a `SAMPLE` clause if the view definition also includes a `TOP n` or `TOP m PERCENT` specification.

The presence of aggregates in a view definition renders that view non-updatable.

expression_alias

A temporary name for the expression.

Alias names are used to name expressions and must be always be specified for a self-join operation on the table.

source

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for complete documentation of the `FROM` clause.

FROM

The names of one or more tables or views from which *expression* is to be derived.

You can create views that reference global temporary tables and volatile tables.

A view can reference a global temporary trace table, but this view is not updatable. See [CREATE GLOBAL TEMPORARY TRACE TABLE](#).

You cannot create a view that references a queue table . See [CREATE TABLE \(Queue Table Form\)](#).

For information about the `FROM` clause, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

database_name

user_name

Name of the containing database or user for *table_name* or *view_name* if something other than the current database or user.

table_name

Name of a data table from which columns for this view are to be projected.

view_name

Name of a view from which columns for this view are to be projected.

joined_table_source

joined_table

Name of a joined user base table or view.

You can create views that reference global temporary tables and volatile tables.

A view can reference a global temporary trace table but such a view is not updatable. See [CREATE GLOBAL TEMPORARY TRACE TABLE](#).

You cannot create a view on a queue table. See [CREATE TABLE \(Queue Table Form\)](#).

INNER

A join in which qualifying rows from one table are combined with qualifying rows from another table according to a specified join condition.

INNER is the form specified by the ANSI SQL-2011 standard. Teradata also supports an extension that allows you to separate join relations using COMMA characters.

Inner join is the default join type for view definitions.

OUTER

A join in which qualifying rows from one table or view that do not have matches in the other table or view, are included in the join result. The rows from the outer table or view are extended with nulls.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for more information about outer joins.

LEFT OUTER

An outer join on the table or view that was listed first in the FROM clause.

In a LEFT OUTER JOIN, the rows from the left table or view that are not returned in the result of the inner join of the two tables/views are returned in the outer join result and extended with nulls.

RIGHT OUTER

An outer join on the table or view that was listed second in the FROM clause.

In a RIGHT OUTER JOIN, the rows from the right table or view that are not returned in the result of the inner join of the two tables/views are returned in the outer join result and extended with nulls.

FULL OUTER

A join that returns rows, including non-qualifying rows, from both tables or views.

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

JOIN

An introduction to the name of the second table to participate in the join.

joined_table

The name of the joined user base table, view, or derived table.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

You cannot create or replace a view on a queue table (see [CREATE TABLE \(Queue Table Form\)](#)).

ON *search_condition*

One or more conditional expressions that must be satisfied by the result rows. An ON condition clause is required if the FROM clause specifies outer join syntax.

You cannot specify a SAMPLE clause within a subquery predicate within an ON clause.

CROSS JOIN

A CROSS JOIN is an unconstrained, or Cartesian join. The Cartesian product of two tables/views returns a concatenated product of all rows from all tables or views specified in the FROM clause.

single_table

Name of a user base table or view participating in the join.

You cannot create or replace a view on a queue table (see [CREATE TABLE \(Queue Table Form\)](#)).

derived_table_source

A derived table is constructed by evaluating a table expression over the columns and values of the base table set. The semantics of derived tables correspond to those of views. A view is similar to a named derived table.

Derived tables allow you to specify a spool file composed of selected data from the base table set supporting the table expression in the FROM list of a query.

For information about derived tables, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

subquery

A SELECT request that defines the derived table.

You cannot specify the TOP *n* or TOP *m* PERCENT options (see “TOP *n*” and “TOP *m* PERCENT” later in this table, and *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146) in a subquery.

You cannot specify an ORDER BY clause in the subquery specification of a CREATE VIEW request.

derived_table_name

The name of a derived table.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

search_condition

A conditional search expression that must be satisfied by the row set returned by the specified SELECT request.

You cannot specify a SAMPLE clause within a subquery predicate within a WHERE clause.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for complete documentation of the WHERE clause.

group_specification

GROUP BY

An introduction to an optional reference to one or more expressions in the select expression list used to group rows in the results set.

ordinary_grouping_set

A column expression by which the rows returned by the request are grouped.

You cannot include BLOB, CLOB, UDT, Period, or Geospatial columns in the grouping expression.

You can group by:

- `column_name`

A set of column names drawn from the list of tables specified in the FROM clause of the SELECT request that is used in the GROUP BY clause to specify the columns by which data is to be grouped.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

- `column_position`

The sequential numeric position of columns within the *column_list* clause of the SELECT statement that is used in the GROUP BY clause to specify the order by which data is to be grouped.

The sequential numeric position of columns within the *column_list* clause of the SELECT statement that is used in the GROUP BY clause to specify the order by which data is to be grouped.

This must be a positive integer.

Use of *column_position* is a Teradata extension to the ANSI SQL-2011 standard.

- *expression*

Any list of valid SQL expressions specified for the GROUP BY clause.

You can specify *column_name*, *column_position*, and *expression* either as individual entries or as a list.

Use of *expression* is a Teradata extension to the ANSI SQL-2011 standard.

For details about ordinary grouping sets, see SQL Data Manipulation Language.

empty_grouping_set

A contiguous LEFT PARENTHESIS, RIGHT PARENTHESIS pair with no argument. In general, this syntax is used to request a grand total.

rollup_list

A ROLLUP expression that reports result rows in a single dimension with one or more levels of detail. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for further information.

cube_list

A CUBE expression that reports result rows in multiple dimensions with one or more levels of detail. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for further information.

grouping_sets_specification

A GROUPING SETS expression that reports result rows in one of two ways:

- As a single dimension, but without a full ROLLUP.
- As multiple dimensions, but without a full CUBE.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for further information.

having_condition

One or more conditional Boolean expressions that must be satisfied by the results groups.

You can use aggregate operators in a HAVING condition.

You cannot specify BLOB, CLOB, UDT, or Period columns in the HAVING condition.

You cannot specify a SAMPLE clause within a subquery predicate within a HAVING clause.

HAVING *condition* filters rows from a single group defined in the select expression list that has only aggregate results, or it selects rows from the group or groups defined in a GROUP BY clause.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for complete documentation of the HAVING clause.

qualify_condition

An introduction to a conditional ordered analytical function filtering clause in the SELECT statement.

A QUALIFY clause is valid when coded as part of a seed statement.

QUALIFY is never valid when coded as part of a recursive statement.

One or more conditional Boolean expressions that must be satisfied by the results groups.

You can use aggregate operators in a QUALIFY condition.

You cannot specify BLOB, CLOB, UDT, or Period columns in the QUALIFY condition.

You cannot specify a SAMPLE clause within a subquery predicate within a QUALIFY clause.

QUALIFY *condition* filters rows from a previously computed ordered analytical function.

For more information about the QUALIFY clause, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

WITH CHECK OPTION

Restricts the rows that can be updated in the table by an INSERT or UPDATE request.

The following rules apply to updatable views and WITH CHECK OPTION.

- If WITH CHECK OPTION is specified, the view should be updatable. If a view is not updatable, then specifying WITH CHECK OPTION has no effect.

In addition, any INSERT or UPDATE operations on an underlying base table through the view does not create a row for which the WHERE clause condition evaluates to FALSE.

- If WITH CHECK OPTION is not specified in an updatable view, then any WHERE clause contained in the query defining the view is ignored for any INSERT or UPDATE request made through the view.

- If you create nested views, you can define them to reference only a single base table, which might allow those views to be updatable. In this case, the presence of a WITH CHECK OPTION in a view definition causes the WHERE clause for that view, and WHERE clauses of any underlying views, to be checked in the constraint for INSERT or UPDATE requests.

order_by_specification

Order in which result rows are to be sorted.

You can only specify an ORDER BY clause for a view definition if you also specify either the TOP *n* or the TOP *m* PERCENT operators.

For more information about the ORDER BY clause, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

expression

An expression in the subselect expression list, either by name, or by means of a constant that specifies the numeric position of the expression in the expression list.

If the sort field is a character string, the *expression* used in the ORDER BY phrase can include a type modifier to force the sort to be either CASESPECIFIC or NOT CASESPECIFIC.

You cannot specify BLOB, CLOB, UDT, Period, or Geospatial columns or ordered analytic functions in the ORDER BY expression list.

column_name

Names of columns used in the ORDER BY clause in the subselect expression. These can be ascending or descending.

You cannot specify BLOB, CLOB, UDT, Period, or Geospatial columns in the ORDER BY column list.

column_name_alias

A column name alias specified in the select expression list of the view definition for the column on which the result rows are to be sorted.

If you specify a *column_name_alias* to sort by, then that alias cannot match the name of any column that is defined in the view definition for any table or view referenced in the FROM clause of the view definition whether that column is specified in the select list or not. This does not work because the system always references the underlying physical column having the name rather than the column that you attempt to reference using that same name as its alias.

If you specify an improper *column_name_alias*, the system returns an error to the requestor.

The workaround for this is to specify the sort column by its *column_position* value within the select list for the view definition. See ORDER BY in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

column_position

Numeric position of the columns specified by the ORDER BY clause. These can be ascending or descending.

ASC

Ascending sort order.

The default order is ASC.

If a sort option is not specified, result values are sorted in ascending order according to the client system collating sequence.

If ORDER BY is not specified, rows are returned unsorted.

DESC

Descending sort order.

If ORDER BY is not specified, rows are returned unsorted.

CREATE VIEW and REPLACE VIEW Examples

Example: Creating a View with Column Titles

The following request creates a view of the *department* table. Each column in the view is defined with a title. Therefore, the view data is displayed with column titles that differ from those defined for the *department* table.

```
CREATE VIEW dept AS
  SELECT  deptno(TITLE 'Department Number'),
          deptname(TITLE 'Department Name'),
          loc (TITLE 'Department Location'),
          mgrno(TITLE 'Manager Number')
  FROM department;
```

Example: Using a View in an Update, Insert, or Delete Request

This example shows how use of a view name in an UPDATE, INSERT, or DELETE request allows you to add, change, or remove data in the table set on which the view is based. For example, updating data via a view changes data in the underlying table. Inserting or deleting rows via a view adds or removes rows from the underlying table.

Consider the following *staff_info* view, which gives a personnel clerk retrieval access to employee numbers, names, job titles, department numbers, sex, and dates of birth for all employees except vice presidents and managers:

```
CREATE VIEW staff_info (number, name, position, department,
sex, dob) AS
  SELECT employee.empno, name, jobtitle, deptno, sex, dob
  FROM employee
  WHERE jobtitle NOT IN ('Vice Pres', 'Manager')
  WITH CHECK OPTION;
```

If the owner of *staff_info* has the insert privilege on the *employee* table, and if the clerk has the insert privilege on *staff_info*, then the clerk can use this view to add new rows to *employee*. For example, performing the following INSERT request inserts a row in the underlying *employee* table that contains the specified information:

```
INSERT INTO staff_info (number, name, position, department, sex,
                        dob)
VALUES (10024, 'Crowell N', 'Secretary', 200, 'F', 'Jun 03 1960');
```

The constraint on *staff_info* illustrated by the following WHERE clause applies to any insert using this view that includes the WITH CHECK OPTION phrase.

```
...
WHERE jobtitle NOT IN ('Vice Pres', 'Manager')
...
```

Therefore, the preceding INSERT request would fail if the Position entered for Crowell was Vice Pres or Manager.

If this view were defined to not include the WITH CHECK OPTION, and a user had UPDATE privilege, that user could update a job title to *Vice Pres* or *Manager*. The user would be unable to access the changed row through the view.

The following request changes the department number (from 200 to 300) entered for Crowell in the preceding INSERT request:

```
UPDATE staff_info
SET department = 300
WHERE number = 10024;
```

Performing the following DELETE request removes the row for employee Crowell from the *staff_info* table:


```
DELETE FROM staff_info
WHERE number = 10024;
```

A view is a useful method for allowing users access to table data. However, as the preceding examples suggest, granting another user insert, update, and delete privileges on a view means relinquishing some control over your data. Carefully consider granting such privileges.

The default is not to constrain updated or inserted values unless the view definition explicitly includes WITH CHECK OPTION.

Example: Creating a View With a Dynamic UDT Expression

This example shows how a dynamic UDT can be specified in a view definition.

First, the following request shows the long version in which the column expressions are all specified:

```
SELECT udf_aggregate_mp_struct(NEW VARIANT_TYPE(MultiType.w AS w,
MultiType.x AS x,
      MultiType.y AS y,
      NEW MP_STRUCTURED_INT(MultiType.w, MultiType.x,
      MultiType.y) AS z)) AS m
FROM MultiType;
*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.
      m
-----
      60
```

Now create a view that incorporates the previous SELECT request:

```
CREATE VIEW multitype_v AS
SELECT udf_aggregate_mp_struct(NEW VARIANT_TYPE(MultiType.w AS w,
MultiType.x AS x,
      MultiType.y AS y,
      NEW MP_STRUCTURED_INT(MultiType.w, MultiType.x,
      MultiType.y) AS z)) AS m FROM MultiType;
```

Running the following SELECT request against the newly created view returns the identical result:

```
SELECT *
FROM multitype_v;
*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.
      m
```

60

Example: Replacing a View

To change the department *name* column in the *employee_info* view to a department number column, type:

```
REPLACE VIEW employee_info (number, name, position, department)
AS SELECT employee.empno, name, jobtitle, deptno
FROM emp_info
WHERE jobtitle NOT IN ('vice pres', 'manager');
```

You must have the DROP privilege on a view or its containing database or user to replace it.

If you enter a REPLACE VIEW request for a view that does not exist, the system creates the view following the specifications in the REPLACE statement.

Example: Creating a View on a Table With Row-Level Security Constraints

This example shows how you can create a view on a table that is defined with row-level security constraints. First create the table *emp_record*. The view *emp_record_view* will be defined on this table.

The row-level security constraint in the *emp_record* table is defined as *group_membership*. When Vantage creates this *emp_record*, it implicitly adds a fourth security constraint column named *group_membership* to the table.

```
CREATE TABLE emp_record (
  emp_name  VARCHAR(30),
  emp_number INTEGER,
  salary    INTEGER,
  group_membership CONSTRAINT)
UNIQUE PRIMARY INDEX (emp_name);
```

Now define the view *emp_record_view* on *emp_record*.

```
CREATE VIEW emp_record_view AS
SELECT emp_number, salary, group_membership
FROM emp_record
WHERE emp_name=user;
```

Example: Using a WITH Modifier in CREATE VIEW

A common table expression (CTE) in a WITH modifier can reference either preceding or subsequent CTEs defined in the WITH modifier, provided that the CTE does not indirectly reference itself. That is, circular references are not allowed.

Following is the table definition for this example.

```
CREATE TABLE orders (customer_id INTEGER, total_cost FLOAT);
INSERT INTO orders (43563, 734.12);
INSERT INTO orders (65758, 211.15);
INSERT INTO orders (23235, 1264.98);
INSERT INTO orders (43563, 583.23);
INSERT INTO orders (89786, 278.66);
INSERT INTO orders (13253, 401.97);
INSERT INTO orders (98765, 1042.23);
INSERT INTO orders (23235, 699.23);
INSERT INTO orders (43563, 935.35);
INSERT INTO orders (88354, 375.09);
```

This example uses a named query in the WITH modifier to create the view sales_v. The WITH modifier includes a nonrecursive common table expression (CTE), specified as multiple_order_totals, that references the table multiple_orders, which is previously defined in the WITH clause.

```
CREATE VIEW sales_v AS
  WITH multiple_orders AS (
    SELECT customer_id, COUNT(*) AS order_count
    FROM orders
    GROUP BY customer_id
    HAVING COUNT(*) > 1
  ),
  multiple_order_totals AS (
    SELECT customer_id, SUM(total_cost) AS total_spend
    FROM orders
    WHERE customer_id IN (SELECT customer_id FROM multiple_orders)
    GROUP BY customer_id
  )
  SELECT * FROM multiple_order_totals;
```

Then, you can query the view.

```
SELECT * FROM sales_v
ORDER BY total_spend DESC;
```

The query returns this answer set:

customer_id	total_spend
43563	2.25270000000000E 003
23235	1.96421000000000E 003

This example of a WITH modifier includes a nonrecursive common table expression (CTE), specified as `multiple_order_totals`, that references the table `multiple_orders`, which is subsequently defined in the WITH clause.

```
CREATE VIEW sales_v AS
WITH multiple_order_totals AS (
  SELECT customer_id, SUM(total_cost) AS total_spend
  FROM orders
  WHERE customer_id IN (SELECT customer_id FROM multiple_orders)
  GROUP BY customer_id
),
multiple_orders AS (
  SELECT customer_id, COUNT(*) AS order_count
  FROM orders
  GROUP BY customer_id
  HAVING COUNT(*) > 1
)
SELECT * FROM multiple_order_totals;
```

Then, you can query the view.

```
SELECT * FROM sales_v
ORDER BY total_spend DESC;
```

The query returns this answer set:

customer_id	total_spend
43563	2.25270000000000E 003
23235	1.96421000000000E 003

Example: Using a Recursive Query in a WITH Modifier in CREATE VIEW

This shows the view `t1_view` based on the recursive query `s5`.

```
CREATE VIEW t1_view AS WITH RECURSIVE s5 (MinVersion_view) AS (SELECT a1 FROM
t1 WHERE a1 > 1
UNION ALL
SEL MinVersion_view FROM s5 WHERE MinVersion_view > 3),
RECURSIVE s6 (MinVersion_view2) AS (SELECT a1 FROM t1 WHERE a1 = 2
UNION ALL
SEL MinVersion_view2 FROM S6 WHERE MinVersion_view2 > 2)
SEL * FROM s5,s6;
```

This statement displays the content of the view `t1_view`.

```
SEL * FROM t1_view;
```

MinVersion_view	MinVersion_view2
3	2
2	2

Example: Creating a View that Defines a Self-Join with a Table

The following request creates a view that allows a personnel executive to keep track of employees who have more experience on the job than their supervisors. This request defines a self-join of the *employee* table. The correlation names *workers* and *managers*, created in the FROM clause, see the two temporary tables participating in the self-join.

```
CREATE VIEW emp_info (workername,workeryrsexp,department,
                    managername,manageryrsexp) AS
SELECT workers.name, workers.yrsexp, workers.deptno,
       managers.name, managers.yrsexp
FROM employee AS workers, employee AS managers
WHERE workers.deptno = managers.deptno
AND managers.jobtitle IN ('Manager', 'Vice Pres')
AND workers.yrsexp > managers.yrsexp;
```

Example: Invoking an SQL UDF Within a View Definition

This example invokes the SQL UDF *value_expression* in the select list of the definition of the view named *v1*.

```
CREATE VIEW v1 (a, b, c)
AS SELECT a1, test.value_expression(3,4), c1
FROM t1
WHERE a1 > b1;
```

The next example invokes the SQL UDF *value_expression* in the WHERE clause of the view named *v2*.

```
CREATE VIEW v2 (a, b, c)
AS SELECT a1, b1, c1
FROM t1
WHERE test.value_expression(b1, c1) > 10;
```

Example: Creating a View with Aggregates

The following request illustrates the use of aggregates in a view definition. The result rows are grouped by department number and include only those rows with an average salary of \$35,000 or higher.

```
CREATE VIEW dept_sal (deptno, minsal, maxsal, avgsal) AS
  SELECT deptno, MIN(salary), MAX(salary), AVG(salary)
  FROM employee
  GROUP BY deptno
  HAVING AVG(salary) >= 35000;
```

Now perform the following SELECT request using this view:

```
SELECT *
FROM dept_sal;
```

The query returns the following response set:

DeptNo	MinSal	MaxSal	AvgSal
-----	-----	-----	-----
600	28,600.00	45,000.00	36,650.00
300	23,000.00	65,000.00	47,666.67
700	30,000.00	45,000.00	37,666.67
500	22,000.00	56,000.00	38,285.71

The following SQL request returns the response set that follows:

```
SELECT deptno, minsal, minsal+10000, avgsal
FROM dept_sal
WHERE avgsal > (minsal + 10000);
```

DeptNo	MinSal	(MinSal+10000)	AvgSal
-----	-----	-----	-----
500	22,000.00	32000.00	38,285.71
300	23,000.00	33000.00	47,666.67

Example: Creating a View for Names and Job Titles Only

The following request creates a view of the *employee* table so that it provides access only to the names and job titles of the employees in department 300:

```
CREATE VIEW dept300 (name, jobtitle) AS
  SELECT name, jobtitle
  FROM employee
```

```
WHERE DeptNo = 300
WITH CHECK OPTION;
```

The WITH CHECK OPTION prevents using this view to INSERT a row into *employee* through the view, or to update any row of *employee* for which *DeptNo* <> 300.

Related Information

- [DROP MACRO](#)

CREATE RECURSIVE VIEW and REPLACE RECURSIVE VIEW

Creates or replaces a recursive view definition.

Any recursive view defined on a row-level security-protected table must include the row-level security constraint columns in the view definition.

ANSI Compliance

CREATE RECURSIVE VIEW is compliant with the ANSI SQL:2011 standard.

REPLACE RECURSIVE VIEW is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the CREATE VIEW privilege to create a new recursive view using the CREATE RECURSIVE VIEW syntax or to create a new recursive view using the REPLACE RECURSIVE VIEW syntax. You can only use the REPLACE syntax to create a recursive view if that view does not already exist.

To replace an existing recursive view, you must have the DROP VIEW privilege on the view or its containing database.

The creator receives the DROP VIEW and SELECT privileges on the newly created recursive view WITH GRANT OPTION.

If a user other than the owner needs to access a recursive view, then all of the relevant privileges required by the immediate owner of the recursive view to access underlying tables and views must also be held by the user of the recursive view WITH GRANT OPTION.

You cannot update a base table through a recursive view. You cannot reference a recursive view using any of the following SQL DML statements:

- DELETE
- INSERT
- MERGE
- UPDATE

Because of this, there are no privileges associated with update operations on a recursive view.

Privileges Granted Automatically

When you create a new view, the following privileges are automatically granted to the view:

- DELETE
- DROP VIEW
- GRANT
- INSERT
- SELECT
- UPDATE

CREATE RECURSIVE VIEW and REPLACE RECURSIVE VIEW Syntax

```
{ CREATE | REPLACE } RECURSIVE VIEW [ database_name_1. |
user_name_1. ] view_name_1
  ( column_name [,...] ) AS
  { view_specification | ( view_specification ) } [;]
```

view_specification

```
[ locking_specification [...] ]
[ date_specification ]
seed_statement
UNION ALL [ union_specification [...] ]
recursive_statement
```

locking_specification

```
LOCKING item_to_lock [ FOR | IN ] lock_type [ MODE ] [ NOWAIT ]
```

date_specification

```
AS OF calendar_function (
  { DATE date_expression | TIMESTAMP [ WITH TIME ZONE ]
  timestamp_expression }
  [, calendar_name ]
)
```


seed_statement

```
{ SELECT | SEL } [ DISTINCT | ALL ] { * |
seed_statement_selection [, ...] }
  FROM seed_statement_source [, ...]
  [ WHERE search_condition ]
  [ GROUP BY grouping_specification [, ...] ]
  [ HAVING having_condition ]
  [ QUALIFY qualify_condition ]
```

union_specification

```
{ seed_statement_specification [...] | recursive_statement UNION ALL }
```

recursive_statement

```
{ SELECT | SEL } [ ALL ] { * | recursive_statement_selection [, ...] }
  FROM recursive_statement_source [, ...]
  [ WHERE search_condition ] [;]
```

item_to_lock

```
{ [ DATABASE ] { database_name_2 | user_name_2 } |
  [ TABLE ] [ database_name_3. | user_name_3. ] table_name |
  [ VIEW ] [ database_name_4. | user_name_4. ] view_name_2 |
  ROW
}
```

lock_type

```
{ ACCESS |
  { EXCLUSIVE | EXCL } |
  SHARE |
  READ [ OVERRIDE ] |
  WRITE |
  LOAD COMMITTED
}
```

seed_statement_selection

```
{ expression | [ database_name. | user_name. ] table_name.* }
```

seed_statement_source

```
{ [ database_name. | user_name. ]
  { table_name [ [AS] correlation_name ] |

    joined_table
      { join_on | CROSS JOIN [ database_name. | user_name. ]
single_table }
  } |

  ( subquery ) [AS] derived_table_name [ ( column_name [,...] ) ]
}
```

grouping_specification

```
{ ordinary_grouping_set |
  empty_grouping_set |
  rollup_list |
  cube_list |
  grouping_set_specification
}
```

seed_statement_specification

```
seed_statement UNION ALL
```

recursive_statement_selection

```
{ expression [ [AS] correlation_name ] | table_name.* }
```

recursive_statement_source

```
{ table_name [ [AS] correlation_name ] |

  joined_table
    { [ INNER | { LEFT | RIGHT | FULL } [ OUTER ] ]
```

```

        JOIN joined_table ON search_condition |

    CROSS JOIN single_table
}
}

```

join_on

```

[ INNER | { LEFT | RIGHT | FULL } [ OUTER] ] JOIN
[ database_name. | user_name. ] joined_table ON search_condition

```

CREATE RECURSIVE VIEW and REPLACE RECURSIVE VIEW Syntax Elements

database_name_1***user_name_1***

Containing database or user for *view_name_1* if something other than the current database or user.

view_name_1

Name of the recursive view to be created or replaced.

If *view_name_1* is not fully qualified, the default database is used.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

column_name

Mandatory name of a view column or column set. If more than one column is specified, list their names in the order in which each is to be displayed for the view.

locking_specification**LOCKING**

Type of lock to be placed on a database, table, view, or row hash. This setting will override any default lock placed on that object by the system.

You can specify a LOCKING clause in the seed portion of a recursive view definition, but not in the recursive portion of its definition.

lock_type

Severity of the lock to be placed when accessing base tables through this view as:

- ACCESS
- READ
- WRITE
- EXCLUSIVE
- LOAD COMMITTED

MODE

Optional keyword.

NOWAIT

Specifies that if the indicated lock cannot be obtained, the request using this view should be aborted.

This option is used to avoid a potential deadlock situation, where a request is waiting for resources, possibly tying up other resources in the process of waiting for a lock to become available.

item_to_lock**DATABASE**

Specifies that a lock is to be placed at the database level for this view definition.

database_name_2***user_name_2***

Name of the database or user to be locked.

TABLE

Specifies that a lock is to be placed at the table or view level for this view definition.

table_name

Name of a user base table to be locked for this view definition.

You can create views that reference global temporary tables and volatile tables.

A view can reference a global temporary trace table, but such a view is not updatable. See [CREATE GLOBAL TEMPORARY TRACE TABLE](#).

Note:

A recursive view definition can reference both row-level security-protected (RLS) tables and non-RLS tables, or other views that are based upon RLS tables, but all RLS base tables referenced in the view must have the same RLS constraints. It is not necessary to specify the row-level security constraint columns in the view definition. However when users access the view, the system enforces row-level security constraints for base tables regardless of whether the constraints are part of the view definition.

database_name_3***user_name_3***

Name of the containing database or user for *table_name* if different from the current database or user.

VIEW

Specifies that a lock is to be placed at the view level for this view definition.

view_name_2

Name of a view to be locked for this view definition.

database_name_4***user_name_4***

Name of the containing database or user for *view_name* if different from the current database or user.

ROW

Specifies that a lock is to be placed at the row hash level for this view definition.

date_specification

Specifies that the view is being defined AS OF a specific point in time.

calendar_function

One of the embedded services system calendar functions.

For more information about the embedded services calendar functions, see *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211.

date_expression

A DATE expression argument for the specified embedded services system calendar function.

You can specify a SELECT CURRENT_DATE or SELECT CURRENT_DATE AT TIME ZONE argument instead of an explicit DATE or DATE AT TIME ZONE expression argument.

timestamp_expression [WITH TIME ZONE]

A TIMESTAMP or TIMESTAMP WITH TIME ZONE expression argument for the specified embedded services system calendar function.

You can specify a SELECT CURRENT_TIMESTAMP or SELECT CURRENT_TIMESTAMP AT TIME ZONE argument instead of an explicit TIMESTAMP or TIMESTAMP WITH TIME ZONE expression argument.

calendar_name

Optional system calendar name used to define a different calendar than the one assigned to the current session.

For the definitions of the system calendars, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

You do not need to specify a system calendar name for week-related specific times because, for example, the previous Monday for a specified date is the same for all of the calendars.

seed_statement**SELECT
SEL**

Specifies that the *expression* to be associated with the *column_name* list is to select from one or more existing user base tables or views.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for complete documentation of the SELECT clause.

DISTINCT

Only one row is to be returned from any set of duplicates that might result from a given *expression* list.

Two rows are considered duplicates only if each value in one is equal to each corresponding value in the other.

The DISTINCT operator is only valid when used within a seed statement.

ALL

All rows, including duplicates, are to be returned in the results of the expression list.

This is the default value.

All columns from all tables referenced in the FROM clause are to be returned.

When qualified by *table_name*, * (asterisk) specifies that all columns of only the user base table or view specified by *table_name* are to be returned.

View columns are explicitly enumerated when views are defined. If a table is changed after a view is defined, those changes will not appear if the SELECT * construct is used.

seed_statement_selection

expression

A valid SQL expression.

You can specify both aggregate and arithmetic operators in the seed query of a view definition expression, but not in its recursive query.

The presence of aggregates in a view definition renders that view non-updatable.

database_name

user_name

Containing database or user for *table_name* if something other than the current database or user.

table_name

Name of a user base table or view.

This can itself be a recursive reference in the recursive statement portion of the view definition. You cannot make a reference to a recursive view in the seed statement of the view definition.

Use a *table_name.** specification in the select list to define the specific table from which rows are to be returned when two or more tables are referenced in the FROM clause.

You cannot create or replace a view on a queue table (see [CREATE TABLE and CREATE TABLE AS](#)).

FROM *seed_statement_source*

Names of one or more tables or views from which *expression* is to be derived.

The FROM clause in a seed statement must not reference *view_name*.

Either the FROM clause in a recursive statement or a subquery specified within the recursive statement must reference *view_name*.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for information about the FROM clause.

database_name**user_name**

Containing database or user for *table_name* if something other than the current database.

table_name

Table from which to derive *expression*.

correlation_name

An alias name for the user base table that is referenced by *table_name*.

Correlation names are used to name table expressions and must always be specified for a self-join operation on the table.

Correlation names are also referred to as range variables.

database_name**user_name**

Containing database or user for *joined_table* if something other than the current database or user.

joined_table

Name of a joined user base table or view.

You cannot create or replace a view on a queue table (see [CREATE TABLE \(Queue Table Form\)](#)).

join_on**INNER**

A join in which qualifying rows from one table are combined with qualifying rows from another table according to a specified join condition.

INNER is the form specified by the ANSI SQL-2011 standard. Teradata also supports an extension that allows you to separate join relations using COMMA characters.

Inner join is the default join type for view definitions.

OUTER

A join in which qualifying rows from one table or view that do not have matches in the other table or view, are included in the join result. The rows from the outer table or view are extended with nulls.

Outer joins are only valid selectively in recursive view definitions. See the individual outer join types for specific information.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for more information about outer joins.

LEFT OUTER

An outer join on the table or view that was listed first in the FROM clause.

In a LEFT OUTER JOIN, the rows from the left table or view that are not returned in the result of the inner join of the two tables/views are returned in the outer join result and extended with nulls.

A left outer join is valid when coded within a seed statement.

Left outer joins are only valid within a recursive statement when the recursive query reference is the inner, or left, table in the outer join definition.

RIGHT OUTER

An outer join on the table or view that was listed second in the FROM clause.

In a RIGHT OUTER JOIN, the rows from the right table or view that are not returned in the result of the inner join of the two tables/views are returned in the outer join result and extended with nulls.

A right outer join is valid when coded within a seed statement.

Right outer joins are only valid within a recursive statement when the recursive query reference is the outer, or right, table in the outer join definition.

FULL OUTER

A join that returns rows, including non-qualifying rows, from both tables or views.

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

Full outer joins are only valid when coded within a seed statement.

One or more conditional expressions that must be satisfied by the result rows. An ON condition clause is required if the FROM clause specifies outer join syntax.

You cannot specify a `SAMPLE` clause within a subquery predicate within an `ON` clause.

CROSS JOIN

A `CROSS JOIN` is an unconstrained, or Cartesian join. The Cartesian product of two tables/views returns a concatenated product of all rows from all tables or views specified in the `FROM` clause.

database_name

user_name

Containing database or user for *single_table* if something other than the current database or user.

single_table

Name of a user base table or view participating in the join.

You cannot create or replace a view on a queue table (see [CREATE TABLE and CREATE TABLE AS](#)).

subquery

A `SELECT` statement that defines the derived table.

derived_table_name

Name of a derived table.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

column_name

A column name for a member of the set of all column names specified in the subquery select list.

Specify only an unqualified column name here; do not use fully qualified forms such as the following:

- *databaseName.tableName.columnName*
- *tableName.columnName*

WHERE *search_condition*

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for complete documentation of the `WHERE` clause.

search_condition

A conditional search expression that must be satisfied by the row set returned by the specified SELECT statement.

You cannot specify a SAMPLE clause within a subquery predicate within a WHERE clause.

GROUP BY *grouping_specification*

An introduction to an optional reference to one or more expressions in the select expression list used to group rows in the result set.

A GROUP BY clause is valid when coded as part of a seed statement.

GROUP BY is never valid when coded as part of a recursive statement.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for more information.

ordinary_grouping_set

A column expression by which the rows returned by the request are grouped.

ordinary_grouping_set includes:

- *column_name* specifies a set of column names drawn from the list of tables specified in the FROM clause of the SELECT statement that is used in the GROUP BY clause to specify the columns by which data is to be grouped.

You cannot include LOB columns in the grouping expression.

- *column_position* specifies the sequential numeric position of columns within the *column_list* clause of the SELECT statement that is used in the GROUP BY clause to specify the order in which data is to be grouped.

This must be a positive integer.

You cannot include LOB columns in the grouping expression.

Use of *column_position* is a Teradata extension to the ANSI SQL-2011 standard.

- *expression* specifies any list of valid SQL expressions specified for the GROUP BY clause.

You can specify *column_name*, *column_position*, and *expression* either as individual entries or as a list.

You cannot include LOB columns in the ordinary grouping set.

Use of *expression* is a Teradata extension to the ANSI SQL-2011 standard.

empty_grouping_set

A contiguous LEFT PARENTHESIS, RIGHT PARENTHESIS pair with no argument. In general, this syntax is used to request a grand total.

rollup_list

A ROLLUP expression that reports result rows in a single dimension with one or more levels of detail. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for further information.

cube_list

A CUBE expression that reports result rows in multiple dimensions with one or more levels of detail. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for further information.

grouping_sets_specification

A GROUPING SETS expression that reports result rows in one of two ways:

- As a single dimension, but without a full ROLLUP.
- As multiple dimensions, but without a full CUBE.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for further information.

HAVING *having_condition*

An introduction to a conditional grouping clause in the SELECT statement.

A HAVING clause is valid when coded as part of a seed statement.

HAVING is never valid when coded as part of a recursive statement.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for complete documentation of the HAVING clause.

having_condition

one or more conditional Boolean expressions that must be satisfied by the results groups.

You can use aggregate operators in a HAVING condition.

You cannot specify a SAMPLE clause within a subquery predicate within a HAVING clause.

You cannot specify BLOB, CLOB, UDT, or Period columns in the HAVING condition.

HAVING *condition* filters rows from a single group defined in the select expression list that has only aggregate results, or it selects rows from the group or groups defined in a GROUP BY clause.

QUALIFY *qualify_condition*

An introduction to a conditional ordered analytical function filtering clause in the SELECT statement.

A QUALIFY clause is valid when coded as part of a seed statement.

QUALIFY is never valid when coded as part of a recursive statement.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for complete documentation of the QUALIFY clause.

qualify_condition

One or more conditional Boolean expressions that must be satisfied by the results groups.

You can use aggregate operators in a QUALIFY condition.

You cannot specify a SAMPLE clause within a subquery predicate within a QUALIFY clause.

You cannot specify BLOB, CLOB, UDT, or Period columns in the QUALIFY condition.

QUALIFY *condition* filters rows from a previously computed ordered analytical function.

Examples**Example: A Simple Recursive View**

This example uses the following base table as its base to build a simple recursive review definition with a counter to control the possibility of infinite recursion:

partlist		
Part	Subpart	Quantity
00	01	5
00	05	3
01	02	2
01	03	3
01	04	4
02	05	7
02	06	6

This view is designed to answer questions such as the following: Which parts are required to build part 01?

Notice that as written, this query does not really answer the question because there might be more than 100 levels in the data. Strictly speaking, the question this request asks is this: Which parts, up to a maximum of 100 levels, are required to build part 01?

The recursive view definition is as follows:

```
CREATE RECURSIVE VIEW rpl (part, subpart, quantity, depth) AS (
  SELECT root.part, root.subpart, root.quantity, 0 AS depth
  FROM partlist AS root
  WHERE root.part = '01'
  UNION ALL
  SELECT child.part, child.subpart, child.quantity, parent.depth + 1
  FROM rpl AS parent, partlist AS child
  WHERE parent.subpart = child.part
  AND   parent.depth <= 100);
```

The query to answer the question of which parts are required to build part 01 is the following SELECT request:

```
SELECT part, subpart, quantity
FROM rpl
ORDER BY part, subpart, quantity;
```

The result set for this query is the following.

Part	Subpart	Quantity
01	02	2
01	03	3
01	04	4
02	05	7
02	06	6

Example: Simple REPLACE RECURSIVE VIEW

The following example replaces the existing definition for the recursive view named rec with the specified SQL code:

```
REPLACE RECURSIVE VIEW rec(p, mycount) AS (
  SELECT n, 0 AS mycount
  FROM t
  WHERE n = 1
```

```

UNION ALL
  SELECT rec.p, rec.mycount + 1
  FROM t, rec
  WHERE rec.p = t.n
  AND   rec.mycount <= 20

```

Example: Controlling Infinite Recursion

The following example uses the *flights* table to indicate a method for limiting the possibility of infinitely recursive processing of cyclic data:

```

CREATE RECURSIVE VIEW reachable_from (destination, cost, depth) AS (
  SELECT root.destination, root.cost, 0 AS depth
  FROM flights AS root
  WHERE root.source = 'Paris'
  UNION ALL
  SELECT out.destination, in.cost + out.cost, in.depth + 1 AS depth
  FROM reachable_from AS in, flights AS out
  WHERE in.destination = out.source
  AND   in.depth <= 20);

```

This recursive view is written to be queried by the following SELECT request.

```

SELECT *
FROM reachable_from;

```

In this example, the variable *depth* is used as a counter, initialized to 0 within the seed query for the recursive view definition and incremented by 1 within the recursive query for the definition.

The AND condition of the WHERE clause then tests the counter to ensure that it never exceeds a value of 20. Because the depth counter was initialized to 0, this condition limits the recursion to 21 cycles.

Example: Aggregate and Ordered Analytic Function Usage

The first recursive view definition demonstrates the valid use of an ordered analytic function, which is highlighted in **bold** typeface. The usage is valid because it is within the seed statement rather than the recursive statement of the definition.

```

CREATE RECURSIVE VIEW reachable_from (source, destination, cost,
                                     depth) AS (
  SELECT source, destination, MSUM(flights.cost, 25,
flights.destination), 0 AS depth
  FROM flights

```

```

UNION ALL
  SELECT in1.source, out1.destination, in1.cost + out1.cost,
         in1.depth + 1
  FROM reachable_from in1, flights AS out1
  WHERE in1.destination = out1.source
  AND   in1.depth <= 100);

```

The second recursive view definition demonstrates a non-valid use of the same ordered analytic function, which is again highlighted in **bold** typeface. The usage is not valid because it is within the recursive statement rather than the seed statement of the definition.

```

CREATE RECURSIVE VIEW oaf_problem (source, destination, mcost,
                                   depth) AS (
  SELECT source, destination, MSUM(cost, 25, destination),
         0 AS depth
  FROM flights
  UNION ALL
  SELECT in1.source, out1.destination,
         MSUM(in1.mcost + out1.cost, 25, out1.destination),
         in1.depth + 1
  FROM oaf_problem AS in1, flights AS out1
  WHERE in1.destination = out1.source
  AND   in1.depth <= 100;

```

Example: Left Outer Join Usage

The first recursive view definition demonstrates a correct use of a left outer join, which is highlighted in **bold**. The usage is valid because the recursive relation in the recursive statement of the view definition is used as the outer relation in the left outer join.

```

CREATE RECURSIVE VIEW rec (f1, mycount) AS (
  SELECT a1, 0 AS mycount
  FROM nonrec
  UNION ALL
  SELECT a2, mycount + 1
  FROM rec LEFT OUTER JOIN nonrec ON nonrec.a1 = rec.f1
  WHERE rec.mycount <= 100);

```

The second recursive view definition demonstrates a non-valid use of a left outer join, which is highlighted in **bold**. The usage is not valid because the recursive relation in the recursive statement of the view definition is used as the inner relation in the left outer join.


```
CREATE RECURSIVE VIEW rec (f1, mycount) AS (
  SELECT a1, 0 AS mycount
  FROM nonrec
  UNION ALL
  SELECT a2, mycount + 1
  FROM nonrec LEFT OUTER JOIN rec ON nonrec.a1 = rec.f1
  WHERE rec.mycount <= 100);
```

You can use left outer joins without restriction in the seed statement of a recursive view definition.

Example: Right Outer Join Usage

The first recursive view definition demonstrates a correct use of a right outer join, which is highlighted in **bold** typeface. The usage is valid because the recursive relation in the recursive statement of the view definition is used as the outer relation in the right outer join.

```
CREATE RECURSIVE VIEW rec (f1, mycount) AS (
  SELECT a1, 0 AS mycount
  FROM nonrec
  UNION ALL
  SELECT a2, mycount + 1
  FROM nonrec RIGHT OUTER JOIN rec ON nonrec.a1 = rec.f1
  WHERE rec.mycount <= 100);
```

The second recursive view definition demonstrates a non-valid use of a right outer join, which is highlighted in **bold** typeface. The usage is not valid because the recursive relation in the recursive statement of the view definition is used as the inner relation in the right outer join.

```
CREATE RECURSIVE VIEW rec (f1, mycount) AS (
  SELECT a1, 0 AS mycount
  FROM nonrec
  UNION ALL
  SELECT a2, mycount + 1
  FROM rec RIGHT OUTER JOIN nonrec ON nonrec.a1 = rec.f1
  WHERE rec.mycount <= 100);
```

Right outer joins can be used without restriction in the seed statement of a recursive view definition.

Example: Full Outer Join Usage

You cannot code a full outer join within the recursive statement of a recursive view definition.

Full outer joins can be used without restriction in the seed statement of a recursive view definition.

The first recursive view definition demonstrates a correct use of a full outer join, which is highlighted in **bold** typeface. The usage is valid because the full outer join is coded within the seed statement of the view definition.

```
CREATE RECURSIVE VIEW rec (f1, mycount) AS (
  SELECT a1, 0 AS mycount
  FROM nonrec1 FULL OUTER JOIN nonrec2 ON nonrec1.a1 = nonrec2.a2
  UNION ALL
  SELECT a2, mycount + 1
  FROM nonrec, rec
  WHERE rec.mycount <= 100);
```

The second recursive view definition demonstrates a non-valid use of a full outer join, which is highlighted in **bold** typeface. The usage is not valid because the recursive relation in the recursive statement of the view definition is used as the inner relation in the full outer join.

```
CREATE RECURSIVE VIEW rec (f1, mycount) AS (
  SELECT a1, 0 AS mycount
  FROM nonrec
  UNION ALL
  SELECT a2, mycount + 1
  FROM nonrec FULL OUTER JOIN rec ON nonrec.a1 = rec.f1
  WHERE rec.mycount <= 100);
```

Example: Mutual Recursion

This example demonstrates mutual recursion, which is not supported for recursive view definitions.

Mutual recursion occurs when both of the following are true:

- Recursive view A invokes, either directly or indirectly, recursive view B.
- Recursive view B invokes, either directly or indirectly, recursive view A.

In the two mutually recursive view definitions provided here, the view named `odd` references the view named `even`, and vice versa.

```
CREATE RECURSIVE VIEW even (n) AS (
  SELECT *
  FROM (SELECT 0) AS a(i)
  UNION ALL
  SELECT m + 1
  FROM odd);
CREATE RECURSIVE VIEW odd (m) AS (
  SELECT *
```

```

FROM (SELECT 1) AS a(i)
UNION ALL
SELECT n + 1
FROM even);

```

Example: RECURSIVE Specified for a Non-Recursive View

The view definition in the following example does not contain a reference to the recursive relation, `rec`, inside its own definition. Because of that omission, even though the statement specifies the keyword `RECURSIVE`, it does not define a recursive view. The result is that the view definition specifies a normal, non-recursive, view.

```

REPLACE RECURSIVE VIEW rec (p) AS (
  SELECT n
  FROM t
  WHERE n = 1
UNION ALL
  SELECT t.n
  FROM t
  WHERE t.n = 0;

```

The `RECURSIVE` keyword means only that a view definition is potentially recursive. This is analogous to the situation where a query written with an outer join specification only potentially makes an outer join, and, depending on how the conditions are specified, might make only an inner join.

Conversely, if you do not specify the `RECURSIVE` keyword, then the view cannot be recursive.

Example: GROUP BY Clause Usage

Similarly to aggregates and ordered analytic functions, a `GROUP BY` clause is valid when coded as part of the seed statement in a recursive view definition, but is not valid when coded as part of the recursive statement.

The first example demonstrates correct usage of a `GROUP BY` clause. In this example, the `GROUP BY` is coded as part of the seed statement.

```

CREATE RECURSIVE VIEW aggregation (source,destination,mycount) AS (
  SELECT source, destination, 0 AS mycount
  FROM flights
  GROUP BY source, destination
UNION ALL
  SELECT in1.source, out1.destination, in1.mycount + 1
  FROM aggregation AS in1, flights AS out1

```

```
WHERE in1.destination = out1.source
AND   in1.mycount <=100);
```

The second example demonstrates non-valid usage of a GROUP BY clause. In this example, the GROUP BY clause is coded as part of the recursive statement.

```
CREATE RECURSIVE VIEW aggregation (source,destination,mycount) AS (
  SELECT source, destination, 0 AS mycount
  FROM flights
  UNION ALL
  SELECT in1.source, out1.destination, in1.mycount + 1
  FROM aggregation AS in1, flights AS out1
  WHERE in1.destination = out1.source
  AND   in1.mycount <=100
  GROUP BY in1.source, out1.destination);
```

Related Information

- [DROP MACRO](#)
- [RENAME MACRO](#)
- [HELP MACRO](#)
- [SHOW object](#)

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for more information about recursion and recursive queries.

RENAME VIEW

Renames an existing view.

ANSI Compliance

RENAME VIEW is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have DROP privileges on the view to be renamed and the appropriate CREATE privileges on its containing database or user.

RENAME VIEW Syntax

```
RENAME VIEW [ database_name. | user_name. ] old_view_name
[ TO | AS ] [ database_name. | user_name. ] new_view_name [;]
```

RENAME VIEW Syntax Elements

database_name

user_name

Optional name of the containing database or user for the view to be renamed or renamed view if other than the current database or user.

You cannot use this statement to change the database or user qualifier for the view.

old_view_name

Existing name for the view.

new_view_name

New name for the view.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

RENAME VIEW Example

The following request renames the `staff_info` view as `staff_information`.

```
RENAME VIEW staff_info TO staff_information;
```

DROP VIEW

Drops the definition of a specified view from the Data Dictionary.

When you drop a view, the system:

- Sets an EXCLUSIVE lock on the view.
- Frees the disk space used by the dropped view and any fallback copy.
- Removes any explicit access privileges on the object.
- Removes the metadata for the dropped object from the data dictionary. This includes dropping any statistics collected on the view from `DBC.StatsTbl`.

ANSI Compliance

DROP VIEW is ANSI SQL:2011-compliant.

Required Privileges

You must have the appropriate DROP privilege on the specified view.

DROP VIEW Syntax

```
DROP VIEW [ database_name. | user_name ] view_name [;]
```

DROP VIEW Syntax Elements

database_name

user_name

Name of the containing database or user for the view to be dropped.

This specification is required only if the view to be dropped is contained in a different database than the current database.

view_name

Name of the view to drop.

Example: Dropping a View

This request drops a view named *dept300*.

```
DROP VIEW dept300;
```

Related Information

- CREATE RECURSIVE VIEW and REPLACE RECURSIVE VIEW in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- CREATE VIEW and REPLACE VIEW in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184

HELP VIEW

Displays the attributes for the specified view or recursive view.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

To execute this statement, you must have either of the following privileges.

- Ownership of the view.
- At least one privilege on the requested object.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against a specified view.

HELP VIEW Syntax

```
HELP VIEW [ database_name. | user_name ] view_name [;]
```

HELP VIEW Syntax Elements

database_name

user_name

Containing database or user for *view_name* if something other than the current database or user.

view_name

View or recursive view for which help is required.

Example: HELP VIEW for a UDT View

The following example shows the HELP VIEW report for a view containing a UDT named `udt_int`.

```
HELP VIEW udt_view;
      Column Name id
            Type I
            Comment ?
            Nullable Y
            Format -(10)9
            Title ?
            Max Length 4
      Decimal Total Digits ?
      Decimal Fractional Digits ?
            Range Low ?
            Range High ?
            UpperCase N
      Table/View? V
      Default value ?
            Char Type ?
            IdCol Type ?
            UDT Name ?
            Temporal ?
      Column Name my_int
            Type UT
            Comment ?
```

```

        Nullable Y
        Format -(10)9
        Title ?
        Max Length 4
        Decimal Total Digits ?
        Decimal Fractional Digits ?
        Range Low ?
        Range High ?
        UpperCase N
        Table/View? V
        Default value ?
        Char Type ?
        IdCol Type ?
        UDT Name ud_int
        Temporal ?

```


Index Statements

CREATE INDEX

Creates a secondary index on an existing data table or join index.

Teradata Vantage™ - SQL Fundamentals, B035-1141 provides an overview of secondary indexes, while *Teradata Vantage™ - Database Design*, B035-1094 provides more detail.

ANSI Compliance

CREATE INDEX is a Teradata extension to the ANSI SQL:2011 standard. The ANSI SQL standard does not define DDL for creating indexes.

The CREATE UNIQUE INDEX syntax is functionally equivalent to adding a UNIQUE NOT NULL or PRIMARY KEY NOT NULL constraint set to a column in ANSI SQL:2011 if all of the columns in the column set are NOT NULL.

A CREATE UNIQUE INDEX statement on a column set where one or more columns are nullable is similar.

Required Privileges

You must have the INDEX or DROP TABLE privilege on the table or join index.

Privileges Granted Automatically

None.

CREATE INDEX Syntax

```
CREATE index_specification [,...] ON { table_specification | join_index_specification } [;]
```

index_specification

```
[UNIQUE] INDEX [index_name] [ALL] (index_column_name [,...] )
    [ ordering_clause ]
    [ loading_clause ]
```

table_specification

```
[TEMPORARY] [ database_name. | user_name. ] table_name
```

join_index_specification

```
[ database_name. | user_name. ] join_index_name
```

ordering_clause

```
ORDER BY [ VALUES | HASH ] [ (order_column_name) ]
```

loading_clause

```
WITH [NO] LOAD IDENTITY
```

CREATE INDEX Syntax Elements***index_specification*****UNIQUE**

Any two rows in the table cannot have the same value or combination of values in the indexed columns.

You cannot specify UNIQUE for a secondary index:

- if you also specify ALL.
- for a secondary index defined on a join index.
- if you also specify an ORDER BY clause.
- for a column that has a LOB UDT or XML data type.

index_name

An optional name for the secondary index.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

ALL

A NUSI should maintain row ID pointers for each logical row of a join index, not just the compressed physical rows.

ALL enables a NUSI to cover a join index, enhancing performance by eliminating the need to access the join index itself when all columns included in a query are contained in the NUSI. Use ALL when a NUSI is being defined for a join index and you want to make it eligible for the Optimizer to select when covering reduces access plan cost.

The ALL option ignores the case specification for a column so that a NUSI defined with the ALL option can:

- Include case-specific values.
- Cover a table or join index on a NOT CASESPECIFIC column set.

You cannot specify the ALL option for primary or unique secondary indexes.

Specifying ALL may require additional index storage space.

You cannot specify multiple NUSIs that differ only by the presence or absence of the ALL option.

ALL is not applicable to geospatial indexes.

index_column_name

The names of one or more columns whose values are to be indexed.

You can define a nonunique secondary index (NUSI) on the same columns included in a primary AMP index.

You can define a NUSI, but not a USI, on a LOB UDT column.

Both a NUSI and a USI can contain row level security constraint columns, but they are not required.

You can specify a user-defined column named *partition* or *partition#L n*, where *n* ranges from 1 through 62. You cannot specify the system-derived columns *PARTITION* or *PARTITION#L n* in the column name list.

You can specify up to 64 columns for the new index. The index is based on the combined values of each column. Unless you specify the *ORDER BY* clause, the index is hash-ordered on all its columns.

If you specify multiple columns, the index is created on the combined values of each column named. A combined maximum of 32 secondary, hash, and join indexes can be created for one table. This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints for nontemporal tables and the single-table join indexes used to implement PRIMARY KEY and UNIQUE constraints for temporal tables. For details, see *Teradata Vantage™ - Temporal Table Support*, B035-1182.

Each multicolumn NUSI defined with an *ORDER BY* clause counts as two consecutive indexes in this calculation.

See *CREATE INDEX* in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

An index on a geospatial column can only contain that column.

The number of system-defined single-table join indexes contributed by PRIMARY KEY and UNIQUE constraints on temporal table columns is included in the combined limit of 32 secondary, hash, and join indexes per base data table.

Multiple indexes can be defined on the same columns as long as each index differs in its ordering option (VALUES versus HASH).

You cannot include columns with the JSON data type in an index.

ordering_clause

Row ordering on each AMP by a single NUSI column: either value-ordered or hash-ordered.

If you specify HASH, the rows are hash-ordered on the AMP. Otherwise, the rows are value-ordered.

Each multicolumn NUSI created with an ORDER BY clause counts as two consecutive indexes against the maximum of any mix of 32 secondary, hash, and join indexes that can be defined per table. This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints for nontemporal tables and the single-table join indexes used to implement PRIMARY KEY and UNIQUE constraints for temporal tables. For details, see *Teradata Vantage™ - Temporal Table Support*, B035-1182. See the CREATE INDEX topic “Why Consecutive Indexes Are Important For Value-Ordered NUSIs” *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

If you do not specify an ORDER BY clause, the system automatically hash orders index rows on their columns.

You cannot order by the system-derived PARTITION or PARTITION#L *n* columns. However, you can specify a user-defined column named partition.

You can order by HASH on a UDT column, but not by VALUE.

ORDER BY is not applicable to geospatial indexes.

VALUES

Value-ordering for the ORDER BY column.

Select VALUES to optimize queries that return a contiguous range of values, especially for a covered index or a nested join.

You cannot specify ORDER BY VALUES on a UDT column.

VALUES is not applicable to geospatial indexes.

HASH

Hash-ordering for the ORDER BY column.

Select HASH to limit hash-ordering to one column.

Hash-ordering a multicolumn NUSI on one of its columns allows the NUSI to participate in a nested join where join conditions involve only that ordering column.

You can specify ORDER BY HASH on a UDT column.

HASH is not applicable to geospatial indexes.

order_column_name

Column in the *index_column_name* list that specifies the sort order to be used.

An order column in a secondary index cannot have a Period, BLOB, CLOB, XML, LOB UDT, Geospatial, ARRAY/VARRAY, or VARIANT_TYPE data type.

You cannot include the system-derived columns PARTITION or PARTITION#L *n* in the *order_column_name* list.

However, you can specify a user-defined column named PARTITION or PARTITION#L *n*, where *n* ranges from 1 through 62.

Supported data types for a value-ordered, four-bytes-or-less *order_column_name* are the following:

- BYTEINT
- DATE
- DECIMAL
- INTEGER
- SMALLINT

If you specify ORDER BY without also specifying an explicit *order_column_name*, then the system uses the first column specified in the *index_column_name* list to order the rows in the index.

loading_clause

You can specify this option for indexes created on load isolated tables.

WITH LOAD IDENTITY

Records the RowLoadID for the index created on a load isolated table.

NO

Does not record RowLoadID with the ROWIDs in a NUSI. The base row, using the qualified ROWID from the index row must be used to determine the committed property. This index may be simpler to maintain but read operations are more expensive than the regular NUSI based option, limiting the usability of index.

*table_specification***TEMPORARY**

The table is a global temporary table and the index is being created on its instance in the current session.

If you do not specify TEMPORARY, the index is created on the base global temporary table definition. In this case, if there are materialized instances of the table you cannot create the index on the base global temporary table.

Temporary tables cannot be specified for geospatial indexes.

If you specify TEMPORARY against a permanent table, the system returns an error.

You cannot create indexes of any kind on a volatile table.

database_name

Name of the containing database for *table_name*, if other than the current database.

user_name

Name of the containing user for *table_name*, if other than the current user.

table_name

Table for which an index is to be created.

You cannot specify any of the following:

- Derived table
- Hash index
- Journal table
- Ordinary view
- Recursive view
- Volatile table

*join_index_specification***database_name**

Name of the containing database for *join_index_name*, if other than the current database.

user_name

Name of the containing user for *join_index_name*, if other than the current user.

join_index_name

Name of the join index on which this index is to be defined.

You cannot define a USI on a join index. NUSIs are allowed.

Examples

Example: Creating Named Indexes

This example demonstrates a named unique secondary index.

```
CREATE UNIQUE INDEX ident (name, ssn) ON employee;
```

This definition allows the named index to be dropped by using its name:

```
DROP INDEX ident ON employee;
```

Example: Creating a Named Unique Secondary Index

This example creates a named unique secondary index *index_1*. Index *index_1* must not be an existing name in table *table_1*.

```
CREATE UNIQUE INDEX index_1 (column_1, column_2) ON table_1;
```

Example: Creating a Named Nonunique Secondary Index

This example creates a named nonunique secondary index *index_2*. Index *index_2* must not be an existing name in table *table_2*.

```
CREATE INDEX index_2 (column_1) ON table_2;
```

Example: Creating an Unnamed Unique Secondary Index

This example creates an unnamed unique secondary index.

```
CREATE UNIQUE INDEX (column_1, column_2) ON table_3;
```

Example: Creating an Unnamed Nonunique Secondary Index

The first example creates a nonunique secondary index on the *dept_no* column of the *employee* table:

```
CREATE INDEX (dept_no) ON employee;
```

The UNIQUE option is not used in this request because multiple rows contain the same department number. The index should be useful, however, because department numbers are often used for conditional retrievals and for ordering employee listings.

The second example creates an unnamed nonunique secondary index on the *column_1* column of a table named *table_4*.

```
CREATE INDEX (column_1) ON table_4;
```

Example: Creating a Secondary Index on a Join Index

This example creates a secondary index on a join index named *order_join_line*:

```
CREATE INDEX shipidx (l_shipdate) ON order_join_line;
```

A query against the base tables uses both indexes:

```
SELECT o_orderdate, o_custkey, l_partkey, l_quantity,
       l_extendedprice
FROM lineitem, order
WHERE l_orderkey = o_orderkey
AND   l_shipdate = '1997-09-18';
```

An EXPLAIN shows that the RETRIEVE step reads from the *order_join_line* join index table by way of the *shipidx* index:

```
"order_join_line.l_shipdate = 970918"
```

Example: Defining Secondary Indexes on UDT Columns

These examples show how UDT columns can be used to define unique and nonunique secondary indexes for a table or join index.

First define the UDTs that are used for the example columns.

The type *tbl_integer* is a distinct UDT based on the INTEGER data type.


```
CREATE TYPE tbl_integer AS INTEGER FINAL;
```

The type *tbl_char50* is also a distinct UDT and is based on the CHARACTER data type.

```
CREATE TYPE tbl_char50 AS CHARACTER(50) FINAL;
```

Assume you define the following tables, where *tbl_integer* and *tbl_char50* are both UDT data types. Note that the UPI for *table_1* and the NUPI for *table_2* are also defined on UDT columns.

```
CREATE TABLE table_1 (
  id      tbl_integer,
  emp_name tbl_char50)
UNIQUE PRIMARY INDEX(id);
CREATE TABLE table_2 (
  id      tbl_integer,
  emp_name tbl_char50,
  start_date DATE)
INDEX(emp_name);
```

The following USI is defined on the UDT column *emp_name* of *table_1*.

```
CREATE UNIQUE INDEX idx_1(emp_name) ON table_1;
```

The following NUSI is defined on the UDT columns *emp_name* and *start_date* of *table_2*.

```
CREATE INDEX idx_2(emp_name, start_date) ON table_2;
```

Examples: Creating Indexes on Load Isolated Tables

This example creates a named secondary index on a load isolated table that records the RowLoadID.

```
CREATE INDEX(dept_no) WITH LOAD IDENTITY ON employee;
```

This example creates a named secondary index on a load isolated table that does not use the RowLoadID:

```
CREATE INDEX(dept_no) WITH NO LOAD IDENTITY ON employee;
```

CREATE JOIN INDEX

Creates a join index for one or multiple tables, optionally with aggregation.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

The ANSI SQL standard does not define DDL for creating indexes.

Other SQL dialects support similar non-ANSI standard statements with names such as the following:

- CREATE MATERIALIZED VIEW

Required Privileges

You must have the following sets of privileges to create a join index:

- CREATE TABLE on the database in which the join index is to be created.
- DROP TABLE or INDEX on each of the covered tables or their containing databases.

Privileges Granted Automatically

Owners have implicit privileges on the join index. The following default privileges are granted automatically to the creator of a join index:

- DROP TABLE
- DUMP
- INDEX
- RESTORE
- SELECT

CREATE JOIN INDEX Syntax

```
CREATE JOIN INDEX [ database_name. | user_name. ] join_index_name
  [ table_option [,...] ]
  select_clause
  FROM source [,...]
  [ WHERE search_condition ]
  GROUP BY grouping_or_ordering_specification [,...]
  ORDER BY grouping_or_ordering_specification [,...]
  [ index [[,]...] ] [;]
```

table_option

```
{ MAP = map_name [ COLOCATE USING colocation_name ] |
  [NO] FALLBACK [ PROTECTION ] |
  CHECKSUM = integrity_checking_level |
  BLOCKCOMPRESSION = block_compression_option
}
```

select_clause

```

AS SELECT
  { selection [,...] |
    ( selection [,...] ) , ( selection [,...] ) |
    [ COLUMN | ROW ] ( selection [,...] ) [ [NO] AUTO COMPRESS ]
  }

```

source

```

{ [ database_name. | user_name. ] table_name [ [AS] corrolation_name ] |
  joined_table
}

```

grouping_or_ordering_specification

```

{ column_name | column_position | column_alias | expression_alias }

```

index

```

{ [ UNIQUE ] PRIMARY INDEX [ index_name ] ( primary_index_column
[,...] ) |
  NO PRIMARY INDEX |
  PRIMARY AMP [ INDEX ] [ index_name ] ( index_column_name [,...] ) |
  PARTITION BY { partitioning_level | ( partitioning_level [,...] ) } |
  INDEX [ index_name ] [ ALL ] ( index_column_name [,...] )
  ORDER BY [ VALUES | HASH ] [ ( order_column_name ) ]
}

```

selection

```

[ [ database_name. | user_name. ] table_name ] { column_name |
ROWID } | aggregation_clause }

```

joined_table

```

{ ( joined_table ) |

  joined_table [ INNER | { LEFT | RIGHT } [ OUTER ] ]
  JOIN joined_table ON search_condition |

```

```

    table_name [ [AS] correlation_name ]
}

```

partitioning_level

```

{ partitioning_expression |
  COLUMN [ [NO] AUTO COMPRESS ] [ [ ALL BUT ] column_partition ]
} [ ADD constant ]

```

aggregation_clause

```

{ expression |
  SUM ( numeric_expression ) |
  { COUNT | MIN | MAX } ( value_expression ) |
  EXTRACT ( { YEAR | MONTH } FROM date_expression )
} [ [AS] expression_alias ]

```

CREATE JOIN INDEX Syntax Elements

database_name

An optional database name if the join index is to reside in a database other than the current database. A join index can be defined in a database other than the database containing the base tables represented in the join index definition.

user_name

An optional user name if the join index is to reside in a user other than the current user. A join index can be defined in a database other than the database containing the base tables represented in the join index definition.

join_index_name

The name of the join index created by this request. For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141. Join index names conform to the rules for naming tables, including explicit qualification with a database or user name.

table_option

Table options provide basic information about the join index being created: the containing database, the name of the join index, whether fallback-protection is enabled, whether the data blocks are compressed, and whether disk I/O integrity checking is enabled.

MAP

You can specify an existing contiguous or sparse map for the join index.

Optionally, you can specify a collocation name. A join index can have different map or collocation name than the base tables.

map_name

Name of an existing contiguous or sparse map.

You cannot specify TD_DataDictionaryMap or TD_GlobalMap.

COLOCATE USING *colocation_name*

Name for collocating the join index on the same AMPs with other tables, join indexes, or hash indexes.

You can only specify this option for a sparse map. For a contiguous map, the *colocation_name* is not needed for collocation and is set to NULL.

If you do not specify a collocation name, the name defaults to *database_index*, where *database* is the name of the database or user followed by an underscore (_) and *index* is the name of the join index. If *database* exceeds 63 characters, *database* is truncated to 63 characters. If *index* exceeds 64 characters, *index* is truncated to 64 characters.

FALLBACK

The join index uses fallback protection.

FALLBACK is the default.

When a hardware read error occurs, the file system reads the fallback copy of the data and reconstructs the rows in memory on their home AMP. Support for Read From Fallback is limited to the following cases:

- Requests that do not attempt to modify data in the bad data block
- Primary subtable data blocks
- Reading the fallback data in place of the primary data. In some cases, Active Fallback can repair the damage to the primary data dynamically. In situations where the bad data block cannot be repaired, Read From Fallback substitutes an error-free fallback copy of the corrupt rows each time the read error occurs.

PROTECTION

An optional keyword.

NO

If a join index is not fallback protected, an AMP failure prevents the following events from occurring:

- Processing queries that use the join index.
- Updating the base table on which the join index is defined.

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

CHECKSUM

A table-specific disk I/O integrity checksum level. The checksum setting applies to primary data rows, fallback data rows, and all secondary index rows for the join index.

integrity_checking_level**ON**

Calculate checksums using the entire disk block. Sample 100% of the disk blocks to generate a checksum.

OFF

Disables checksum disk I/O integrity checks.

DEFAULT

The default setting is the current DBS Control checksum setting specified for this table type.

Example: Specifying a Disk I/O Integrity Checksum for a Join Index

The following example defines a join index with a disk I/O integrity checksum specification of ON. If secondary indexes are defined on `ord_cust_idx`, then the specified checksum value also applies to them.

```
CREATE JOIN INDEX ord_cust_idx, CHECKSUM=ON AS
SELECT c_nationkey, SUM(o_totalprice(FLOAT)) AS price, o_orderdate
FROM orders, customer
WHERE o_custkey = c_custkey
```

```
GROUP BY c_nationkey, o_orderdate
ORDER BY o_orderdate;
```

BLOCKCOMPRESSION

Block-compress the data in the join index, including compression that occurs based on the temperature of the cylinders on which the data is stored. The definitions of the various thresholds are determined by the DBS Control setting TempBLCThresh. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102. For details, see CREATE JOIN INDEX in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

block_compression_option

AUTOTEMP

The file system determines the block-level compression setting for the join index based on its Teradata Virtual Storage temperature. If temperature-based block-level compression is disabled but block-level compression is enabled, Vantage treats AUTOTEMP join indexes the same as MANUAL join indexes. You can still issue query band options or Ferret commands, but if the compressed state of the data does not match its temperature, such changes might be undone by the system over time.

DEFAULT

The join index uses the compression option setting (MANUAL, AUTOTEMP, or NEVER) of the DBS Control parameter DefaultTableMode. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102. Note that value of DefaultTableMode is not saved in the join index definition as part of a CREATE JOIN INDEX request, so a join index set to DEFAULT is affected by any future change to the DefaultTableMode parameter.

MANUAL

Block level compression is applied based on the default for the join index at the time the join index is created. Join indexes can be compressed or uncompressed at any time after loading by using the Ferret COMPRESS and UNCOMPRESS commands. Data inserted into the existing join index inherits the current compression status of the join index at the time the data is inserted.

NEVER

The join index is not compressed even if the DBS Control block compression settings indicate otherwise. Vantage does not allow Ferret commands to manually compress the join index, but Ferret commands to decompress the index are valid.

select_clause

The select list of a join index specification defines the list of columns and expressions that make up the definition for the index.

You can also use the select list of your join index definition to create an aggregate join index by using the SUM, MIN, MAX, and COUNT aggregate functions.

The SELECT clause projects the columns from one or more base tables that are to be included in the join index definition.

You cannot specify the following options in this SELECT clause:

- TOP n, FULL OUTER JOIN, CROSS JOIN, HAVING, QUALIFY, and SAMPLE
- Any function except RANDOM
- Statistical functions
- Aggregate functions other than SUM and COUNT
- Set operators UNION, EXCEPT/MINUS, INTERSECT, and their derivatives
- Subqueries

You cannot specify an EXPAND ON clause in the definition of a join index.

You cannot invoke an SQL UDF anywhere in the definition of a join index.

Certain optional clauses do not become part of the join index definition, for example, output FORMAT and TITLE phrases. You can determine if an option you specify does not become part of the stored join index definition by performing a SHOW JOIN INDEX request to view the DDL for the CREATE JOIN INDEX statement.

For a complete description of the SELECT statement, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

COLUMN

Column format for storing a column partition.

As a general rule, Vantage assigns COLUMN format to narrow column partitions and ROW format to wide column partitions.

COLUMN preceding a column grouping in the select list of the join index definition stores the column or column group in containers using COLUMN format.

If you do not specify COLUMN or ROW, the system determines whether to use COLUMN or ROW format based on the width of the column value and other factors.

ROW

Row format for storing a column partition.

ROW preceding a column in the select list of the join index definition stores the column in subrows using ROW format.

AUTO COMPRESS

NO AUTO COMPRESS

If you do not specify AUTO COMPRESS or NO AUTO COMPRESS, Vantage uses the autocompression value specified after COLUMN keyword that begins the column partitioning specification. If you do not specify an autocompression value there, the system default is AUTO COMPRESS.

For more information about autocompression, see *Teradata Vantage™ - Database Design*, B035-1094.

selection

database_name.table_name

user_name.table_name

The fully qualified path to *column_name*, if required to uniquely identify the *column_name* or ROWID.

Note:

A join index can reference a maximum of 1 row level security table. If you create a join index on a row -level security table, you must include all security constraint columns for the table in the index definition.

column_name

The name of a base table column or an expression to be included in the join index.

You can include columns in the join index with a data type of non-LOB XML, non-LOB ST_GEOMETRY, non-LOB JSON, and non-LOB DATASET. However, you cannot include these data types in the primary index of a join index.

A column in a join index cannot have a data type of BLOB, CLOB, BLOB-based UDT, CLOB-based UDT, VARIANT_TYPE, ARRAY, VARRAY, LOB XML, LOB ST_GEOMETRY, LOB JSON, or LOB DATASET.

You can specify a user-defined column named partition or partition#L-*n*, where *n* ranges from 1 through 62.

However, you cannot include the system-derived columns PARTITION or PARTITION#L *n* in the *column_name* list.

A join index defined with an expression in its select list has more restricted coverage than a join index defined using a base table column.

However, if you define a complex expression in the definition of a single-table join index, you should always collect statistics on it because the Optimizer can use those statistics directly to estimate the selectivity of those expressions for base table expressions specified in a query predicate. For more information, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

You can base an expression on a UDT column or an expression that references at least one column with the following exceptions:

- Expressions that involve aggregate or OLAP functions
- UDF expressions
- Built-in functions or keywords that are explicitly prohibited, such as DEFAULT or PARTITION

For information about the DEFAULT built-in function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145. For information about the PARTITION keyword, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146. These expressions must be aliased. Otherwise, the system returns an error to the requestor.

For two columns with the same name, you must alias both names using a column name alias. See *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

If you also define a repeating group *column_name* list, the system stores each distinct value of the *column_name* list only once.

- If you define a repeating group *column_name* list, the limit on the total number of columns you can specify for the fixed group *column_name* list is 64.
- If you do not define a repeating group *column_name* list, there is no defined limit on the total number of columns you can specify for the fixed group *column_name* list.

The maximum is restricted only by the number of columns that can be defined within the row size limit.

This limit is not related to the restriction of 64 columns that can be defined per referenced base table per join index.

You can specify a maximum of 64 columns per referenced base table per join index.

ROWID

Provides the internal row identifier associated with a row of a base table. The data type of the ROWID column is BYTE(10) except for 8-byte partitioning, where it is BYTE(16). Specify the keyword ROWID as one of the values for either the fixed *column_name* list or the repeating *column_name* list to enable the Optimizer to join a partial covering index to its base table to access any non-covered columns.

If the join index has a primary index, whether partitioned or nonpartitioned, you can optionally specify an alias for the *column_name* and the system-derived ROWID column.

If the join index is a column-partitioned join index, the following rules apply.

- You must specify the system-derived ROWID column and you must also specify an alias for it.
- You can group columns together in the select list of the definition by delimiting them with parentheses. Vantage stores the specified columns together in the same column partition.

You can also group columns in a COLUMN specification of a PARTITION BY clause for the join index, but you cannot group columns in the select list and the COLUMN clause for a column-partitioned join index.

If you reference multiple tables in the join index definition, you must fully qualify each ROWID specification.

A GROUP BY clause that references a ROWID must have a column name alias. The literal ROWID is not valid in the column list argument of a GROUP BY specification.

If you reference a ROWID column name alias in the select list of a join index definition, you can also reference that column name alias in a CREATE INDEX request that creates a secondary index on the join index. However, you cannot directly reference the ROWID keyword in a CREATE INDEX request.

You can only specify ROWID in the outermost select list of a CREATE JOIN INDEX statement.

aggregation_clause

The SUM, COUNT, MIN, MAX, and EXTRACT functions are not valid in the definition of column-partitioned join indexes.

expression

Expression to select columns to use for join index.

SUM(*numeric_expression*)

The SUM aggregate function to compute the sum of the column or column expression with a numeric data type, specified by *numeric_expression*.

You cannot specify a SUM function in the definition of a column-partitioned join index.

The data type for any column defined by a SUM function must be typed as FLOAT to avoid numeric overflow.

If you do not assign a data type to a SUM column, Vantage types it as FLOAT automatically. If you assign a type other than FLOAT, the system returns an error to the requestor.

SUM DISTINCT is not permitted.

Each aggregate join index must be created with a COUNT(*) or a COUNT(value_expression). If you do not specify this expression, Vantage adds a COUNT(*) to the definition to support index maintenance.

For more information about the SUM aggregate function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

COUNT(value_expression)

Use the COUNT aggregate function to count the number of values in the column or column expression with any data type, specified by *value_expression*.

You cannot specify a COUNT function in the definition of a column-partitioned join index.

For a column defined by the COUNT function, you can specify the data type as FLOAT, DECIMAL (38,0), NUMBER, or BIGINT. If you do not assign a data type to a COUNT column, the data type defaults to FLOAT.

COUNT DISTINCT is not permitted.

Each aggregate join index must be created with a COUNT(*) or a COUNT(*value_expression*).

If you do not specify this expression, the system adds a COUNT(*) to the definition for index maintenance, with the default name of CountStar. Do not use the name CountStar if you create an aggregate join index because Vantage reserves that name for any expression being summed or counted in the index definition.

For more information about the COUNT aggregate function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

MIN (value_expression)

Use the MIN aggregate function to compute the minimum value in the column or column expression, specified by *value_expression*. MIN is valid for character data as well as numeric data. For a character expression, MIN returns the lowest sort order.

Each aggregate join index must be created with a COUNT(*) or COUNT(*value_expression*). If you do not specify such an expression, Vantage adds a COUNT(*) by default.

You cannot specify a MIN function in the definition of a column-partitioned join index.

For more information about the MIN aggregate function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

MAX(value_expression)

Use the MAX aggregate function to compute the maximum value in the column or column expression, specified by *value_expression*. MAX is valid for character data as well as numeric data. For a character expression, MAX returns the highest sort order.

Each aggregate join index must be created with a COUNT(*) or COUNT(*value_expression*). If you do not specify such an expression, Vantage adds a COUNT(*) by default.

You cannot specify a MIN function in the definition of a column-partitioned join index.

For more information about the MAX aggregate function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

EXTRACT FROM

Use the EXTRACT date function. You cannot specify an EXTRACT function in the definition of a column-partitioned join index.

See *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211 for more information about the EXTRACT function.

Note that join index definitions support only the extraction of years and months from a date column.

YEAR

Extract a year from a date column.

MONTH

Extract a month from a date column.

date_expression

Date column or date expression from which to derive the year or month.

AS *expression_alias*

A mandatory unique alias for a SUM, COUNT, MIN, MAX, or extracted year or month expression.

The keyword AS preceding *expression_alias* is optional.

All expressions specified in the select list must be aliased. Otherwise, Vantage returns an error to the requestor.

This name permits further use of the result as the grouping key for a GROUP BY clause or the ordering key for an ORDER BY clause.

Row Compression

You can compress a set of join index column values that repeat often across rows. This is referred to as row compression. Group the columns with repeating values. Then, group the columns with values that do not repeat frequently. This non-repeating column set is called the fixed column set.

You can configure row compression for a join index. In the join index definition, group the set of repeating columns in parenthesis, then group the fixed set of columns in parenthesis, separating the two groups with a COMMA character.

The join index *t3_jindx* in [Example: Creating Join Indexes With a UDT Column as a Primary or Secondary Index](#) is a simple example of how to create a row-compressed join index.

For a detailed explanation of join index row compression, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

You cannot specify row compression for a partitioned join index.

Repeating Column Set for Join Index Row Compression

column_name

One of a list of multiple columns to be included for row compression in the join index. This column set represents the repeated portion of the join index row. Together with the second set of parenthetical fixed columns, these columns form the compressed portion of a compressed join index definition.

For more information about row compression for join indexes, see CREATE JOIN INDEX in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

You cannot define a partitioned primary index for a join index that is also compressed.

A column-partitioned join index cannot be row-compressed.

Columns in a join index cannot have any of the following data types: BLOB, CLOB, LOB UDT, VARIANT-TYPE, ARRAY/VARRAY, Geospatial, JSON, or DATASET.

If two specified columns have the same name, both names must be aliased using a column name alias. See *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

You cannot specify the system-derived columns PARTITION or PARTITION#*Ln*, where *n* is an integer from 1 through 62, as part of the *column_name* list.

You can, however, specify a user-defined column named PARTITION or PARTITION#*L n*.

A join index defined with an expression in its select list has more restricted coverage than a join index defined using a base table column. However, if you define a complex expression in the definition of a single-table join index, always collect statistics on it because the Optimizer can use those statistics directly to estimate the selectivity of those expressions for base table expressions specified in a query predicate. For details, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

You can base an expression on a UDT column or an expression that references at least one column with the exception of the following: expressions that involve aggregate or OLAP functions, UDF expressions, and built-in functions or keywords that are explicitly prohibited such as DEFAULT or PARTITION. For information about the DEFAULT built-in function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145. For information about the PARTITION keyword, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146. All such expressions must be aliased. Otherwise Vantage returns an error to the requestor. You can alias the *column_name* with a column name alias.

ROWID

You can specify the keyword ROWID as one of the values for *column_name* to enable the Optimizer to join a partial covering index to its base table to access any non-covered columns.

You cannot specify ROWID in both the fixed and repeating *column_name* group columns. If you reference a ROWID in a GROUP BY clause, you must define a column name alias for it. The literal ROWID is not valid in the column list argument of a GROUP BY specification.

To reference multiple tables in the join index definition, you must fully qualify each ROWID specification. If you reference a ROWID column name alias in the select list of a join index definition, then you can also reference that column name alias in a CREATE INDEX request that creates a secondary index on the join index. You can only specify ROWID in the outermost select list of a CREATE JOIN INDEX statement. Do not specify row compression for aggregate join indexes.

You can specify up to 64 columns for the repeating column set. If you specify a repeating column set, you cannot specify more than 64 columns for the set because the total column limit for a compressed join index is 128 columns. Any number of repeating columns can be stored for a repeating column component of a join index. However, there is a physical limit in that the total size of any one fixed column set-repeated column set pair cannot exceed the Vantage row size limit. You can alias the ROWID with a column name alias.

Fixed Column Set for Join Index Row Compression

column_name

One of a list of multiple columns to be included for row compression in the join index. This column set represents the fixed portion of the join index row. Together with the first set of parenthetical repeating columns, these columns form the compressed portion of a compressed join index definition.

You can alias *column_name* with a column name alias.

You cannot specify the system-derived columns PARTITION or PARTITION#*Ln* as part of the *column_name* list.

However, you can specify a user-defined column named PARTITION or PARTITION#*L n*.

A join index defined with an expression in its select list has more restricted coverage than a join index defined using only base table columns. However, if you define a complex expression in the definition of a single-table join index, always collect statistics on it because the Optimizer can use those statistics directly to estimate the selectivity of those expressions for base table expressions specified in a query predicate. See *Teradata Vantage™ - SQL Request and Transaction Processing* for details.

You can base an expression on a UDT column or on an expression that references at least one column with the exception of the following.

- Expressions based on aggregate or OLAP functions
- Expressions based on a UDF

- Expressions based on explicitly prohibited built-in functions such as DEFAULT. For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

All such expressions must be aliased. Otherwise, Vantage returns an error to the requestor.

If 2 of the specified columns have the same name, you must alias both names using a column name alias.

You can specify up to 64 columns for the fixed column set, because the total column limit for a compressed join index is 128 columns. Any number of fixed columns can be stored for a fixed column component of a join index. However, there is a physical limit in that the total size of any one fixed portion-repeated portion pair cannot exceed the Vantage row size limit.

ROWID

Specify the keyword ROWID as one of the values for the *column_list* to enable the Optimizer to join a partial covering index to its base table to access the non-covered columns. You cannot specify ROWID in both the fixed and repeating column_name group columns.

You can only specify ROWID in the outermost select list of a CREATE JOIN INDEX request.

You can alias ROWID with a column name alias.

If you refer to multiple tables in the join index definition, you must fully qualify each ROWID specification.

If you specify ROWID in a GROUP BY clause, you must define a column name alias for it. The literal ROWID is not valid in the column list argument of a GROUP BY specification.

If you refer to a ROWID column name alias in the select list of your join index definition, you can also refer to that column name alias in a CREATE INDEX request that creates a secondary index on the join index. However, you cannot refer directly to the ROWID keyword in a CREATE INDEX request.

Grouping Column-Partitioned Join Index Column Data

You can group column-partitioned join index column data either in the column specification list or in the partitioning for the index, but not both. The following syntax describes grouping column-partitioned data in the column list for the index.

(column_name)

A list of columns to be grouped together into a column partition. You can only specify this option for a column-partitioned join index.

- If you precede the column grouping with the COLUMN keyword, Vantage stores column partitions using COLUMN format.
- If you precede the column grouping with the ROW keyword, Vantage stores the grouped columns using ROW format.
- If you specify neither COLUMN nor ROW, Vantage makes the determination of whether to store the grouped columns using COLUMN or ROW format.

database_name.table_name**user_name.table_name**

The fully qualified path to the *column_name* or the repeating *column_name* list for a compressed join index definition, if required for unique identification of *column_name* or ROWID.

COLUMN

The columns grouped in the following (*column_name*) specification are to be stored using COLUMN format.

If you do not specify either COLUMN or ROW, Vantage determines which storage format to use.

You can also specify this option in the PARTITION BY clause, but you cannot specify the COLUMN or ROW options in both the select list and in the PARTITION BY clause of the same join index.

ROW

The columns grouped in the following (*column_name*) specification are to be stored using ROW format.

You can also specify this option in the PARTITION BY clause, but you cannot specify the COLUMN or ROW options in both the select list and in the PARTITION BY clause of the same join index.

If you do not specify either ROW or COLUMN, Vantage determines which storage format to use.

column_alias**AS column_alias**

An alias for *column_name*. This is mandatory for an expression.

- If the join index has a primary index, an alias is optional for the system-derived ROWID column.

- If the join index is column-partitioned, an alias is mandatory for the system-derived ROWID column.

All expressions specified in the select list must be aliased.

The AS keyword preceding *column_name_alias* is optional.

Do not use the name CountStar if you create an aggregate join index because the system reserves that name for any expression being summed, counted, or extracted in the aggregate definition.

AUTO COMPRESS

Enable autocompression for a column partition of a column-partitioned join index. Vantage automatically determines and applies the best available compression method, if it can reduce the size of physical rows. AUTO COMPRESS is the default.

Vantage applies any user-specified compression specified for the index columns in their base table and, for column partitions with COLUMN format also applies row header compression.

For more information about autocompression, see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184, and *Teradata Vantage™ - Database Design*, B035-1094.

NO AUTO COMPRESS

Disable autocompression for a column partition of a column-partitioned join index. You cannot specify this option for a primary-indexed join index.

FROM *source*

For more information about the FROM clause, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

database_name

An optional database name if *table_name* is contained in a database other than the current database.

user_name

An optional user name if *table_name* is contained in a user other than the current or user.

table_name

The name of a table containing the column or columns to include in the join index or the name of a table to join with at least one other table to form a join index. You cannot define a join index on any of the following database objects:

- Global temporary tables
- Hash indexes
- Join indexes
- Journal tables
- Ordinary views
- Queue tables
- Recursive views
- Volatile tables

Each table in the FROM clause should have at least one column in either the fixed *column_name* list or in the repeating *column_name* list, if you are using row-compression.

In a single-table join index, the *column_name* list must contain all columns in the FROM clause table.

You can only define column partitioning for a single-table join index.

Each table name can be qualified by a database name if necessary and can have a maximum of 64 different columns referenced in the entire join index definition.

Every table must be connected to at least one other table with a join condition. The cross join and full outer join conditions are not supported.

The maximum number of table_name references allowed is the same as the system limit for SELECT requests.

A FROM clause can have any number of simple tables, but is limited to a single joined non-simple table expression. For example, you can use a simple table list or a single non-simple table expression:

```
FROM table_1, table_2, table_3, table_4
FROM (table_1 INNER JOIN table_2 ON x1 = x2)
INNER JOIN (table_3 ON x1 = x3)
```

The following example is not allowed because it has more than one non-simple table expression:

```
FROM (table_1 INNER JOIN table_2 ON table_1 x1 = x2), (table_3
INNER JOIN table_4 ON x3 = x4)
```

correlation_name

An alias for *table_name*.

The keyword AS preceding *correlation_name* is optional.

joined_table

A recursive use of the syntax of a joined table within the definition of a joined table. The joined table clause is used only for multitable join indexes and cannot be specified in the definition of a column-partitioned join index.

INNER JOIN

Keywords defining the join as an ordinary join.

LEFT OUTER JOIN**RIGHT OUTER JOIN**

Keywords defining the join as either a left or a right outer join. Full outer join is not supported for join indexes. For most non-aggregate applications, this is the better choice for defining a join index. You cannot specify outer joins for a join index that also contains aggregate operators in its definition. When you specify an outer join, the following rules apply:

- The outer table joining column for each condition must be contained in either the fixed *column_name* list or in the repeating *column_name* list, if using row-compression.
- The inner table of each join condition must have at least one non-nullable column in either the fixed *column_name* list or in the repeating *column_name* list, if using row-compression.
- The select list cannot specify a CASE or COALESCE column expression that is defined on a column set from the inner table of the outer join.

search_condition

A conditional expression for eliminating any rows from a query that do not evaluate to TRUE.

You can base an expression on a UDT column or an expression that references at least one column, with the following exceptions:

- Expressions that include aggregate or OLAP functions
- UDF expressions
- Built-in functions or keywords that are explicitly prohibited such as DEFAULT or PARTITION. For information about the DEFAULT built-in function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145. For information about the PARTITION keyword, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

All expressions must be aliased. Otherwise, Vantage returns an error to the requestor.

Multiple join conditions must be connected with the AND logical operator.

There must be an equality join between columns from the outer table and the inner table of the outer join. Then, you can specify any other conditions, such as an inequality between the joined tables and a single table condition on either of the joined tables.

Data types for any columns used in a join condition must be drawn from the same domain because neither explicit nor implicit data type conversions are permitted.

You cannot specify the ROWID keyword in the search condition of a join operation.

table_name

Name of a single table if no tables are joined in this clause.

correlation_name

An alias for *table_name*. The keyword AS preceding *correlation_name* is optional.

WHERE *search_condition*

The WHERE clause is useful for creating sparse join indexes.

For more information about the WHERE clause, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

search_condition

A conditional expression to eliminate all rows from the SELECT clause query that do not evaluate to TRUE.

You can base an expression on a UDT column or an expression that references at least one column with the exception of the following: expressions that involve aggregate or OLAP functions, UDF expressions, and built-in functions or keywords that are explicitly prohibited such as DEFAULT or PARTITION. For information about the DEFAULT built-in function, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

For information about the PARTITION keyword, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

You must alias expressions. Otherwise, an error is returned.

Multiple join conditions must be connected using the AND logical operator.

Data types for any columns used in a join condition must be drawn from the same domain because neither explicit nor implicit data type conversions are permitted.

You cannot specify the ROWID keyword in the search condition of a sparse join index definition.

You must drop any join indexes created prior to Vantage V2R6.2 that contain the keyword ROWID in their definition, then recreate them using currently valid syntax.

If you specify a join condition on a UDT column, the specified tables must be joined on an equality condition. Otherwise, Vantage returns an error to the requestor.

The following rules apply to specifying inequality operators in a WHERE clause search condition:

- Inequality conditions between columns from the same table are always valid.
- Inequality conditions between columns from different tables are supported only if they are ANDed to at least one equality condition.
- Inequality conditions can be specified only for columns having the same data type in order to enforce domain integrity.
- You can only specify a join condition using an inequality operator if you specify multiple join conditions and use the AND logical operator to connect the inequality join condition or conditions with an equality join condition.
- You can specify an equality or inequality join condition between two tables based on an expression if you specify multiple join conditions between the two tables and use the AND logical operator to connect the join condition that is specified using an expression or columnar expressions with an equality join condition between columns from the two tables.
- The only valid comparison operators for an inequality condition are the following.

<

<=

>

>=

GROUP BY *grouping_or_ordering_specification*

Group aggregate index rows on the AMPs.

You cannot specify a GROUP BY clause in the definition of a column-partitioned join index.

For a detailed description of the GROUP BY clause, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

column_name

The name of the column or columns within the join index definition select list used to group aggregate index rows on the AMPs.

The GROUP BY column list must include all the non-aggregated columns for an aggregate join index.

column_position

The ordinal position of the column or columns within the join index definition select list used to group aggregate index rows on the AMPs.

If you specify the system-derived ROWID column in the GROUP BY column list, it must have been previously aliased with a column name alias. You cannot specify the literal ROWID as part of the column list for the GROUP BY option of a join index definition.

If you specify a GROUP BY clause, but do not specify a COUNT or SUM operator, the system adds a COUNT(*) expression, named CountStar, to the column list.

column_alias

An alias for the column or columns.

expression_alias

An alias for the expression.

ORDER BY *grouping_or_ordering_specification*

Order index rows on the AMPs. You cannot specify an ORDER BY clause in the definition of a column-partitioned join index. You cannot specify an ORDER BY clause in the definition for a partitioned join index. Sort order is restricted to ascending. If not specified in the ORDER BY clause, the join index rows on an AMP are sorted by the row hash of their primary index. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for complete documentation of the ORDER BY clause.

column_name

Name of the column within the join index definition select column list used to order index rows on the AMPs. The ORDER BY clause of a join index can specify no more than one column name in its column list. You can specify a user-defined column named PARTITION or PARTITION#L *n*, where *n* ranges from 1 through 62. You cannot specify the system-derived columns PARTITION or PARTITION#L *n* as part of the ORDER BY list. Aggregate columns and expressions are not permitted in the ORDER BY clause. All ordering column names must be in the *column_name_1* list. The *column_name* is limited to a maximum of four bytes of the following data types:

- BYTEINT
- DATE
- DECIMAL
- INTEGER
- SMALLINT

column_position

The ordinal position of the column within the join index definition select column list used to order index rows on the AMPs.

column_alias

An alias for the column.

expression_alias

An alias for the expression.

index

You can use the indexes clause to define a primary index, primary AMP index, partitioning, and a set of secondary indexes for the join index.

You can use this clause with multitable and single-table join indexes.

PRIMARY INDEX

A primary index defined on the join index. The primary index for an aggregate join index must be drawn from the columns specified in the GROUP BY clause of the join index definition. You cannot specify any aggregated columns as part of the primary index. If the primary index is not compressed, you can define it as either a single-level or multilevel row partitioned primary index by specifying one or more partitioning expressions. See [PARTITION BY](#). You cannot partition the primary index of a join index if the index is defined with compression.

The following rules apply to specifying a row partitioned primary index for a join index.

- The partitioning columns for an aggregate join index must be drawn from the columns specified in the GROUP BY clause of the join index definition.
- You cannot specify an aggregated column as a partitioning column for the PPI.
- The partitioning level in a join index acts as a constraint on its underlying base tables. If an insert, delete, or update operation on a base table causes a partition violation in the join index by making one or more of the partitioning expressions evaluate to null, the request returns an error message, and neither the base table nor the join index is updated.
- If a join index definition includes an outer join, deleting base table rows might cause inserts into that join index for the unmatched rows.
- For your partitioned join indexes, you must define partitioning levels that do not prevent rows from being inserted into, updated, or deleted from the base tables when required.
- If you do not define an explicit NUPI, the first column defined for the join index is assigned to be the NUPI by default. You can define an uncompressed join index with a partitioned primary index. However, the primary index cannot be partitioned if the join index is compressed.

- Each NUSI counts toward the maximum number of 32 secondary indexes that you can define on a join index. Each multicolumn NUSI defined with an ORDER BY clause counts as two consecutive indexes against the limit of 32 per join index.

You cannot define the primary index for a join index on a column with any of the following data types:

- BLOB-based UDT
- CLOB-based UDT
- XML-based UDT
- VARIANT-TYPE
- ARRAY/VARRAY
- XML
- Geospatial
- Derived Period
- Period

UNIQUE

You can define a unique primary index for an uncompressed single-table join index, but the primary index for any other join index must be nonunique even when it is defined on a unique column.

index_name

The optional name of the primary being defined. For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

(primary_index_column)

Names of the primary index columns. For a composite primary index, *primary_index_column* indicates a comma-separated list of all the index columns in parenthesis. You cannot specify the begin or end columns of a derived period column in a primary index.

You cannot alter a table to have a row-level security constraint column as a component of its primary index.

You cannot define a primary index on a column defined with the JSON data type.

PRIMARY AMP

Add a primary AMP index.

Rows are hash-distributed to AMPs for a column-partitioned table or join index. Column partition values are ordered on each AMP by an internal partition number and a row hash for a column-partitioned table or join index.

If a PRIMARY AMP clause is specified, you must specify a PARTITION BY clause that includes a column-partitioning level, either in the PRIMARY AMP clause or by itself in the index list.

INDEX

Optionally, the INDEX keyword can be specified with AMP for readability.

index_name

Name of the primary AMP index. For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

(index_column_name)

The primary AMP index columns. For a composite primary index, *index_column_name* indicates a comma-separated list of all the index columns in parenthesis. You cannot specify the begin or end columns of a derived period column in a primary index.

You cannot alter a table to have a row-level security constraint column as a component of its primary index.

You cannot define a primary index on a column defined with the JSON data type.

Example: Join Index with a Primary AMP Index and Column Partitioning

This example assumes the following table has been created:

```
CREATE TABLE t1 (a INT, b INT, c INT, d INT) PRIMARY INDEX (a);
```

The following CREATE JOIN INDEX statements are equivalent:

```
CREATE JOIN INDEX jt1_1 AS SELECT ROWID AS rw, a, b FROM t1 WHERE a<10  
PRIMARY AMP (a) PARTITION BY COLUMN (rw NO AUTO COMPRESS, ROW(a,b));
```

```
CREATE JOIN INDEX jt1_1 AS SELECT ROWID AS rw, a, b FROM t1 WHERE a<10  
PRIMARY AMP INDEX (a) PARTITION BY COLUMN (rw NO AUTO COMPRESS, ROW(a,b));
```

Example: Join Index with a Primary Index and Partitioning by Row and Column

This example assumes the following table has been created:

```
CREATE TABLE t1 (a INT, b INT, c INT, d INT) PRIMARY INDEX (a);
```

This example shows a CREATE JOIN INDEX statement with a primary index and partitioned by row and column:

```
CREATE JOIN INDEX jt1_4 AS SELECT ROWID rw, a, b, d
FROM t1
PRIMARY INDEX (a) PARTITION BY (RANGE_N(b BETWEEN 1 AND 1000 EACH
1), COLUMN);
```

NO PRIMARY

The join index is defined with no primary index or primary AMP index.

You cannot specify NO PRIMARY if you specify UNIQUE PRIMARY INDEX or PRIMARY INDEX.

The NO PRIMARY specification is optional for column-partitioned join indexes. The default is no primary index in this case.

If the preceding item in the index list is a partitioning clause that is not part of an index clause, you must specify a COMMA character preceding NO PRIMARY. Otherwise, the comma is optional.

You cannot specify a column name list following a NO PRIMARY specification.

If the preceding item in the index list is a partitioning clause that is not part of an index clause, you must specify a COMMA character preceding NO PRIMARY INDEX. Otherwise, the comma is optional.

See *Teradata Vantage™ - Database Design*, B035-1094 for the complete set of rules that control primary index defaults.

INDEX

Optional keyword.

PARTITION BY

The join index is partitioned by one or more partitioning levels. You can define either a single-level or multilevel partitioning for a join index, but only if the index is not row-compressed.

You cannot specify a PARTITION BY clause in the same join index definition as an ORDER BY CLAUSE.

You cannot collect statistics on the system-derived PARTITION#L *n* columns for a join index.

partitioning expression

Defines partitions for the level. You can also specify the ADD keyword for a partitioning level. For more information about partitioning tables and join indexes, see *Teradata Vantage™ - Database Design*, B035-1094. If the preceding item in the index list is a partitioning clause that is not part of an index clause, you must specify a COMMA character following the PARTITION BY clause. Otherwise, the comma is optional. You can define multilevel partitioning for an uncompressed join index, with as many as 62 partitioning levels. Each level must be defined by a RANGE_N or CASE_N function or by the COLUMN keyword. You cannot specify character partitioning expressions for columns or constants that use the Kanji1 or KanjiSJIS server character sets. The result of a partitioning expression that is not a RANGE_N or CASE_N function is cast to an INTEGER, if it is not

already an INTEGER. The value must be from 1 through 65,535. A RANGE_N or CASE_N function is only allowed for single-level partitioning. The maximum number of partitioning levels that you can specify for a join index with 2-byte partitioning is 15. The maximum number of partitioning levels that you can specify for a join index with 8-byte partitioning is 62. A partitioning expression for a join index must be a deterministic expression and cannot specify external or SQL UDFs or columns having any of the following data types:

- ARRAY
- VARRAY
- BLOB
- CLOB
- Geospatial

While the partitioning expression for a join index cannot be defined on a Geospatial or ARRAY/VARRAY column, the partitioning expression can be defined on a Period column. This is particularly useful for defining updatable current dates and updatable current timestamps in a PPI.

In deciding whether to use a CASE_N or RANGE_N function in your partitioning expression, refer to the following:

- Use a CASE_N function to define a mapping between conditions to INTEGER numbers.

If a partitioning expression is based only on a CASE_N function, the maximum number of partitions you can define is a maximum of approximately 4,000 conditions.

This maximum can be further limited by other limits such as the size of the request text.
- Use a RANGE_N function to define a mapping of ranges of INTEGER, CHARACTER, or DATE values to INTEGER numbers. You should use a RANGE_N function to map BIGINT and TIMESTAMP values to BIGINT numbers.

If a single-level partitioning expression is based only on a RANGE_N function with INTEGER data type, the maximum number of partitions you can define is 65,535 for 2-byte partitioning and 2,147,483,647 for 8-byte partitioning.

If a partitioning expression is based only on a RANGE_N function with BIGINT data type, the maximum number of partitions you can define is 9,223,372,036,854,775,805.

You can define up to 9,223,372,036,854,775,807 ranges if you specify both the NO RANGE and UNKNOWN partitions.

See *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145 for documentation of functions that are particularly useful for use as partitioning levels, particularly RANGE_N and CASE_N.

You can define a wide variety of partitions with a large range in the number of combined partitions. However, you must consider the usefulness of defining a particular partitioning and its impact, both positive and negative, on performance and storage.

You must specify only a single COLUMN partitioning level for a column-partitioned join index.

You can group multiple columns together into a single partition by specifying a list of columns to be grouped together.

If you specify column grouping in a partitioning level of a join index definition, you cannot also specify column grouping in its select list.

If you specify neither COLUMN nor ROW for a column partition, the system determines whether COLUMN or ROW format is used for the column partition (for information about COLUMN and ROW formats, see *Teradata Vantage™ - Database Design*, B035-1094).

COLUMN

Keyword to indicate partitioning level.

AUTO COMPRESS

NO AUTO COMPRESS

To enable or disable autocompression:

- If you specify AUTO COMPRESS for a column partitioning level in a PARTITION BY clause, the system applies autocompression for a column partition unless the you explicitly specify NO AUTO COMPRESS. This is the default.
- If you specify NO AUTO COMPRESS for a column partitioning level in a PARTITION BY clause, the system does not apply autocompression for a column partition unless you explicitly specify AUTO COMPRESS.

The system applies row header autocompression for column partitions with COLUMN format.

ALL BUT (*column_name_list*)

ALL BUT (*column_group_list*)

A multicolumn partition with autocompression and system-determined COLUMN or ROW format that includes all of the columns in the join index definition that are not specified in *column_name_list* or *column_group_list*.

You can only specify this option for column-partitioned join index.

COLUMN

Specifies that the join index has a column-partitioned level.

If you precede a column grouping with the COLUMN keyword, the system stores the grouped columns using COLUMN format.

ROW

Specifies that a column partition has ROW format. A ROW format means that only one column-partition value is stored in a physical row as a subrow.

If you precede a column grouping with the ROW keyword, the system stores the grouped columns using ROW format.

ADD *constant*

The maximum number of partitions for a partitioning level is the number of partitions it defines plus the value of the BIGINT constant value specified by *constant*.

The value for *constant* must be an unsigned BIGINT constant and cannot exceed 9,223,372,036,854,775,807.

You can only specify this option for column-partitioned join indexes.

INDEX

A set of nonunique secondary indexes defined on the join index.

index_name

The optional name of the NUSI being defined. For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

ALL

The defined NUSI is to maintain Row ID pointers for each logical row of the join index, instead of only the compressed physical rows.

ALL also ignores the NOT CASESPECIFIC attribute of data types so a NUSI can include case-specific values.

ALL enables a NUSI to cover a join index, which enhances performance by eliminating the need to access the join index when all values needed by a query are in the secondary index. However, ALL might also require the use of additional index storage space.

Use this keyword only when a secondary index is being defined on top of a join index.

You cannot specify ALL with a PRIMARY index.

You cannot specify multiple indexes that differ only by the presence or absence of the ALL option.

index_column_name

A column set whose values are to be an index on this join index. If you specify more than one column, the new index is based on the combined values of each column.

A maximum of 64 columns can be defined for one index.

Multiple indexes can be defined on the same columns as long as each index differs in its ordering option, for example, VALUES or HASH.

If two specified columns have the same name, both names must be aliased with unique column name aliases. See *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Columns in a secondary index cannot have any of the following data types: .

- Period
- Geospatial
- BLOB
- CLOB
- ARRAY/VARRAY

In the *index_column_name* list, you cannot specify the system-derived columns PARTITION or PARTITION#*Ln*, where *n* is an integer from 1 through 62. However, you can specify a user-defined column named PARTITION or PARTITION#*L n*.

ORDER BY

Row ordering on each AMP by a single NUSI column. The ordering of the NUSI column values can either value-ordered or hash-ordered. Each multicolumn NUSI defined with an ORDER BY clause counts as two consecutive indexes against the limit of 32 per join index. You cannot specify an ORDER BY clause in the same join index definition as a partitioned primary index.

Following are the rules for specifying an ORDER BY clause:

- If you specify ORDER BY VALUES, *column_name* must be numeric with four bytes or less.
ORDER BY VALUES is not supported for UDT columns.
- If you specify ORDER BY without specifying the HASH or VALUES keywords, VALUES is assumed by default.
- The ORDER BY *column_name* must be one of the columns specified in the select list.
- You cannot specify the system-derived PARTITION column as the ORDER BY column.

You can specify a user-defined column named partition.

VALUES

Value-ordering for the ORDER BY column. This is the default specification.

ORDER BY VALUES is not supported for UDT columns.

Select VALUES to optimize queries that return a contiguous range of values, especially for a covered index or a nested join.

HASH

Hash-ordering for the ORDER BY column.

You can specify ORDER BY HASH on a UDT column.

Select HASH to limit hash-ordering to one column, rather than all columns of the primary index, which is the default.

Hash-ordering a multicolumn NUSI on one of its columns allows the index to participate in a nested join where join conditions involve only that ordering column.

order_column_name

A column in the select list that specifies the sort order to be used for NUSI ordering.

Columns in a join index cannot have any of the following data types: Geospatial, BLOB, CLOB, XML, LOB UDT, XML UDT, VARIANT_TYPE, or ARRAY/VARRAY.

You cannot specify the system-derived columns PARTITION or PARTITION#*Ln*, where *n* is an integer ranging from 1 through 62, inclusive, as part of the *column_name_2* list.

However, you can specify a user-defined column named PARTITION or PARTITION#*L n*.

Supported data types for a value-ordered, four-bytes-or-less *column_name_2* are:

- BYTEINT
- DATE
- DECIMAL
- INTEGER
- SMALLINT

Usage Notes**Default Map for the User, Database, or Profile**

The immediate owner of the join index can be a user or a database. See the CREATE USER [DEFAULT MAP](#) option, the MODIFY USER [DEFAULT MAP](#) option, the CREATE DATABASE [DEFAULT MAP](#) option, the MODIFY DATABASE [DEFAULT MAP](#) option, the CREATE PROFILE [DEFAULT MAP](#) option, or the MODIFY PROFILE [DEFAULT MAP](#) option.

User, Database, or Profile DEFAULT MAP OVERRIDE ON ERROR Option

If the map you specify is not valid for any reason, (such as it does not exist, has not been granted to you, or is not in the same secure zone), Vantage either substitutes a default map or returns an error, subject to the values of the DEFAULT MAP settings for your PROFILE, USER, or DATABASE.

Default Map for a CREATE JOIN INDEX Statement

If you do not specify the MAP option when creating a join index, the system determines a default map for the join index in the following order of precedence:

- If the immediate owner is not the creator:
 - Default map, if defined, for the profile of the immediate owner.
 - Default map, if defined, for the immediate owner.
 - System-default map.
- Default map, if defined, for the profile of the creator.
- Default map, if defined, for the creator.
- System-default map.

Secure Zones and Sparse Maps

For a sparse map, you must be in the same secure zone as the sparse map.

Examples

Example: Creating and Using a Simple Join Index

This example set uses the following table definitions.

```
CREATE TABLE customer (
  c_custkey    INTEGER,
  c_name       CHARACTER(26),
  c_address    VARCHAR(41),
  c_nationkey  INTEGER,
  c_phone      CHARACTER(16),
  c_acctbal    DECIMAL(13,2),
  c_mktsegment CHARACTER(21),
  c_comment    VARCHAR(127))
UNIQUE PRIMARY INDEX (c_custkey);

CREATE TABLE orders (
  o_orderkey    INTEGER NOT NULL,
  o_custkey     INTEGER,
  o_orderstatus CHARACTER(1),
  o_totalprice  DECIMAL(13,2) NOT NULL,
  o_orderdate   DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk       CHARACTER(16),
  o_shippriority INTEGER,
```

```

        o_comment      VARCHAR(79))
    UNIQUE PRIMARY INDEX (o_orderkey);

CREATE TABLE lineitem (
    l_orderkey          INTEGER NOT NULL,
    l_partkey           INTEGER NOT NULL,
    l_suppkey           INTEGER,
    l_linenummer        INTEGER,
    l_quantity          INTEGER NOT NULL,
    l_extendedprice     DECIMAL(13,2) NOT NULL,
    l_discount          DECIMAL(13,2),
    l_tax               DECIMAL(13,2),
    l_returnflag        CHARACTER(1),
    l_linestatus        CHARACTER(1),
    l_shipdate          DATE FORMAT 'yyyy-mm-dd',
    l_commitdate        DATE FORMAT 'yyyy-mm-dd',
    l_receiptdate       DATE FORMAT 'yyyy-mm-dd',
    l_shipinstruct      VARCHAR(25),
    l_shipmode          VARCHAR(10),
    l_comment           VARCHAR(44))
    PRIMARY INDEX (l_orderkey);

```

The following statement defines a join index on these tables. Subsequent examples demonstrate the effect of this join index on how the Optimizer processes various queries.

```

CREATE JOIN INDEX order_join_line AS
SELECT (l_orderkey, o_orderdate, o_custkey, o_totalprice),
       (l_partkey, l_quantity, l_extendedprice, l_shipdate)
FROM lineitem
LEFT JOIN orders ON l_orderkey = o_orderkey
ORDER BY o_orderdate
PRIMARY INDEX (l_orderkey);

*** Index has been created.
*** Total elapsed time was 15 seconds.

```

The following query is provided as an example to show how the newly created join index, `order_join_line`, can be used by the Optimizer. An `EXPLAIN` of the `SELECT` statement includes a `RETRIEVE` step from join index table `order_join_line` by way of an all-rows scan with a condition of ("NOT (order_join_line.o_orderdate IS NULL)").

```

EXPLAIN SELECT o_orderdate, o_custkey, l_partkey, l_quantity,
l_extendedprice

```

```
FROM lineitem, orders
WHERE l_orderkey = o_orderkey;
```

Explanation

- 1) First, we lock ORDER_JOIN_LINE for read on a reserved RowHash to prevent global deadlock.
 - 2) Next, we lock ORDER_JOIN_LINE for read.
 - 3) We do an all-AMPs RETRIEVE step from tom.ORDER_JOIN_LINE by way of an all-rows scan with a condition of ("NOT (ORDER_JOIN_LINE.o_orderdate IS NULL)") into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated to be 1,000,000 rows. The estimated time for this step is 4 minutes and 27 seconds.
- > The contents of Spool 1 are sent back to the user as the result of statement 1.

The following query is provided as an example to show how the join index might be used when an additional search condition is added on the join indexed rows. An EXPLAIN of the SELECT statement includes a RETRIEVE step from join index table order_join_line with a range constraint of ("order_join_line.Field_1026 > 971101") with a residual condition of ("(NOT (order_join_line.o_orderdate IS NULL)) AND (order_join_line.Field_1026 > 971101)."

```
SELECT o_orderdate, o_custkey, l_partkey, l_quantity,
       l_extendedprice
FROM lineitem, orders
WHERE l_orderkey = o_orderkey
AND o_orderdate > '1997-11-01';
```

The following query is provided as an example to show how the join index might be used when aggregation is performed on the join indexed rows. An EXPLAIN of the SELECT statement includes a SUM step to aggregate from join index table order_join_line with a range constraint of ("order_join_line.Field_1026 > 971101") with a residual condition of ("(order_join_line.Field_1026 > 971101) AND (NOT (order_join_line.o_orderdate IS NULL))") and the grouping identifier.

```
SELECT l_partkey, avg(l_quantity), AVG(l_extendedprice)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey
AND o_orderdate > '1997-11-01'
GROUP BY l_partkey;
```

The following query is provided as an example to show how the join index might be used in a query when join indexed rows are used to join with another base table. An EXPLAIN of the SELECT

statement includes a RETRIEVE step from join index table `order_join_line` with a condition of ("NOT (order_join_line.o_orderdate IS NULL)") and a SORT to order by row hash. This is followed by a JOIN step from `customer` using a merge join, with a join condition of ("o_custkey =customer.c_custkey").

```
SELECT o_orderdate, c_name, c_phone, l_partkey, l_quantity,
       l_extendedprice
FROM lineitem, orders, customer
WHERE l_orderkey = o_orderkey
AND o_custkey = c_custkey;
```

The following query is provided as an example to show how the join index might be used in a query of a single table. An EXPLAIN of the SELECT statement includes a RETRIEVE step from join index table `order_join_line` with a condition of ("order_join_line.l_partkey = 1001").

```
SELECT l_orderkey, l_partkey, l_quantity, l_extendedprice
FROM lineitem
WHERE l_partkey = 1001;
```

Example: Creating and Using a Simple Row-Compressed Join Index

This example creates a simple row-compressed join index based on the following *customer* and *orders* tables.

```
CREATE TABLE customer (
  c_custkey    INTEGER,
  c_name       CHARACTER(26) not null,
  c_address    VARCHAR(41),
  c_nationkey  INTEGER,
  c_phone      CHARACTER(16),
  c_acctbal    DECIMAL(13,2),
  c_mktsegment CHARACTER(21),
  c_comment    VARCHAR(127))
PRIMARY INDEX( c_custkey );
CREATE TABLE orders (
  o_orderkey    INTEGER,
  o_date        DATE FORMAT 'yyyy-mm-dd',
  o_status      CHARACTER(1),
  o_custkey     INTEGER,
  o_totalprice  DECIMAL(13,2),
  o_orderpriority CHARACTER(21),
  o_clerk       CHARACTER(16),
  o_shippriority INTEGER,
```

```
o_comment      VARCHAR(79))
UNIQUE PRIMARY INDEX(o_orderkey);
```

Assume that you frequently submit this join query on *customer* and *orders*.

```
SELECT o_custkey, c_name, o_status, o_date, o_comment
FROM orders, customer
WHERE o_custkey = c_custkey;
```

Without a join index, a typical execution plan for this query would redistribute the *orders* table into a spool file, sort the spool on *o_custkey*, and then doing a merge join between the spool file and the *customer* table.

If a join index like *ord_cust_idx* were defined on *customer* and *orders*, the resulting execution plan could just scan the join index.

The fixed column set in the join index definition is *o_custkey* and *c_name* (corresponding results rows highlighted in orange) , and the repeating column set in the join index definition is *o_status*, *o_date*, and *o_comment*, (corresponding results rows highlighted in blue).

```
CREATE JOIN INDEX ord_cust_idx AS
  SELECT (o_custkey, c_name
), (
  o_status, o_date, o_comment
)
  FROM orders, customer
  WHERE o_custkey = c_custkey;
```

The way *ord_cust_idx* is defined makes it a row-compressed join index, with the fixed column set being *o_custkey* and *c_name* and the repeating column set being *o_status*, *o_date*, and *o_comment*.

Assume that the base tables *customer* and *orders* contain the following data.

customer		
c_custkey	c_name	c_address
100	Louis	Rancho Bernardo
101	Pekka	Helsinki
102	Felipe	San Juan

orders				
o_orderkey	o_date	o_status	o_custkey	o_comment
5000	2011-10-01	S	102	rush order

5001	2011-10-01	S	100	big order
5002	2011-10-03	D	102	delayed
5003	2011-10-05	U	?	unknown customer
5004	2011-10-05	S	100	credit
5005	2011-10-08	P	101	discount

The corresponding rows in the row-compressed join index *ord_cust_idx* are the following.

ord_cust_idx	
fixed columns	repeated columns
100 Louis	S 2011-10-01 big order 2011-10-05 credit
101 Pekka	P 2011-10-08 discount
102 Felipe	S 2011-10-03 rush order D 2011-10-03 delayed

Notice that the fixed set columns are contained in the first column of *ord_cust_idx* and the repeated set columns are contained in the second column of *ord_cust_idx*, just as the join index definition had defined.

A drawback of join results generated from inner joins is that unmatched rows are not preserved. In the first part of this example, information about order 5003 is not stored in *ord_cust_idx*, thus limiting the types of queries that could be resolved using the index. To make join indexes applicable to as many queries as possible, you should define your join indexes using outer joins whenever possible. This enables a join index to satisfy queries with fewer join conditions than those used to generate the index.

Suppose you changed the definition of *ord_cust_idx* to use an outer join as the following CREATE JOIN INDEX request does.

```
CREATE JOIN INDEX ord_cust_idx AS
SELECT (o_custkey, c_name), (o_status, o_date, o_comment)
FROM orders LEFT OUTER JOIN customer ON o_custkey = c_custkey;
```

The resulting rows in *ord_cust_idx* are the following:

ord_cust_idx	
fixed columns	repeated columns
100 Louis	S 2011-10-01 big order S 2011-10-05 credit
101 Pekka	P 2011-10-08 discount
102 Felipe	S 2011-10-03 rush order D 2011-10-03 delayed
? ?	U 2011-10-05 unknown customer

By redefining *ord_cust_idx* using a left outer join, the following query could also be resolved using only the join index.

```
SELECT o_status, o_date, o_comment
FROM orders;
```

Example: Defining and Using a Simple Join Index With an n -way Join Result

This statement creates a join index defined with a multiway join result:

```
CREATE JOIN INDEX cust_order_join_line AS
SELECT (l_orderkey, o_orderdate, c_nationkey, o_totalprice),
       (l_partkey, l_quantity, l_extendedprice, l_shipdate)
FROM (lineitem
LEFT JOIN orders ON l_orderkey = o_orderkey)
INNER JOIN customer ON o_custkey = c_custkey
PRIMARY INDEX (l_orderkey);
```

The following query is provided as an example to show how the Optimizer uses the join index to process a query on the base tables for which it is defined. An EXPLAIN of the SELECT statement includes RETRIEVE step from join index table *cust_order_join_line* by way of an all-rows scan with a condition of ("cust_order_join_line.c_nationkey = 10").

```
SELECT l_orderkey, o_orderdate, o_totalprice,
       l_partkey, l_quantity, l_extendedprice, l_shipdate
FROM lineitem, orders, customer
WHERE l_orderkey = o_orderkey
AND o_custkey = c_custkey
AND c_nationkey = 10;
```

Example: Using the EXTRACT Function With a Join Index Definition

Join index definitions support the EXTRACT function.

The following example illustrates the use of the EXTRACT function in the definition of an aggregate join index:

```
CREATE JOIN INDEX ord_cust_idx_2 AS
SELECT c_nationkey, SUM(o_totalprice(FLOAT)) AS price,
       EXTRACT(YEAR FROM o_orderdate) AS o_year
FROM orders, customer
```

```
WHERE o_custkey = c_custkey
GROUP BY c_nationkey, o_year
ORDER BY o_year;
```

The aggregation is based only on the year of o_orderdate, which has fewer groups than the entire o_orderdate, so ord_cust_idx_2 is much smaller than ord_cust_idx.

On the other hand, the use for ord_cust_idx_2 is more limited than ord_custidx. In particular, ord_cust_idx_2 can only be used to satisfy queries that select full years of orders.

For example, ord_cust_idx_2 cannot be used for the query analyzed in [Example: Creating and Using an Aggregate Join Index](#), but the following query does profit from its use:

```
SELECT COUNT(*), SUM(o_totalprice)
FROM orders, customer
WHERE o_custkey = c_custkey
AND   o_orderdate > DATE '1998-01-01'
AND   o_orderdate < DATE '1998-12-31'
GROUP BY c_nationkey;
```

Example: Creating and Using an Aggregate Join Index

This example set uses the following table definitions:

```
CREATE TABLE customer (
  c_custkey    INTEGER NOT NULL,
  c_name       CHARACTER(26) CASESPECIFIC NOT NULL,
  c_address    VARCHAR(41),
  c_nationkey  INTEGER,
  c_phone      CHARACTER(16),
  c_acctbal    DECIMAL(13,2),
  c_mktsegment CHARACTER(21),
  c_comment    VARCHAR(127))
UNIQUE PRIMARY INDEX (c_custkey);
CREATE TABLE orders (
  o_orderkey    INTEGER NOT NULL,
  o_custkey     INTEGER,
  o_orderstatus CHARACTER(1) CASESPECIFIC,
  o_totalprice  DECIMAL(13,2) NOT NULL,
  o_orderdate   DATE FORMAT 'YYYY-MM-DD' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk       CHARACTER(16),
  o_shippriority INTEGER,
```



```
o_comment      VARCHAR(79))
UNIQUE PRIMARY INDEX (o_orderkey);
```

Consider the following aggregate join query.

```
SELECT COUNT(*), SUM(o_totalprice)
FROM orders, customer
WHERE o_custkey = c_custkey
AND   o_orderdate > DATE '1998-09-20'
AND   o_orderdate < DATE '1998-10-15'
GROUP BY c_nationkey;
```

Without an aggregate join index, a typical execution plan for this query might involve the following stages:

1. Redistribute orders into a spool file.
2. Sort the spool file on o_custkey.
3. Merge join the sorted spool file and the customer file.
4. Aggregate the result of the merge join.

Suppose you define the following aggregate join index, which aggregates o_totalprice over a join of orders and customer :

```
SELECT c_nationkey, SUM(o_totalprice(FLOAT)) AS price, o_orderdate
FROM orders, customer
WHERE o_custkey = c_custkey
GROUP BY c_nationkey, o_orderdate
ORDER BY o_orderdate;
```

The execution plan produced by the Optimizer for this query includes an aggregate step on the aggregate join index, which is much smaller than either one of the join tables. An EXPLAIN of the SELECT statement includes a SUM step to aggregate from join index table ord_cust_idx with a condition of ("(ord_cust_idx.O_ORDERDATE > DATE '1998-09-20') AND (ord_cust_idx.O_ORDERDATE < DATE '1998-10-15')"), grouped by c_nationkey.

```
EXPLAIN SELECT COUNT(*), SUM(o_totalprice)
FROM orders, customer
WHERE o_custkey = c_custkey
AND   o_orderdate > DATE '1998-09-20'
AND   o_orderdate < DATE '1998-10-15'
GROUP BY c_nationkey;
```

Example: Creating an Aggregate Join Index Using the MIN and MAX Functions

This example demonstrates a simple aggregate join index that uses the MIN and MAX functions defined on the join result of the base tables *customer* and *order_tbl*.

```
CREATE TABLE customer (
  c_custkey    INTEGER not null,
  c_name       CHARACTER(26) CASESPECIFIC NOT NULL,
  c_address    VARCHAR(41),
  c_nationkey  INTEGER,
  c_phone      CHARACTER(16),
  c_acctbal    DECIMAL(13,2),
  c_mktsegment CHARACTER(21),
  c_comment    VARCHAR(127))
UNIQUE PRIMARY INDEX( c_custkey );
```

```
CREATE TABLE order_tbl (
  o_orderkey    INTEGER NOT NULL,
  o_custkey     INTEGER,
  o_orderstatus CHARACTER(1) CASESPECIFIC,
  o_totalprice  DECIMAL(13,2) NOT NULL,
  o_orderdate   DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk       CHARACTER(16),
  o_shippriority INTEGER,
  o_comment     VARCHAR(79))
UNIQUE PRIMARY INDEX(o_orderkey);
```

You define the following aggregate join index *ord_cust_idx* on *order_tbl* and *customer* using the MIN and MAX functions.

```
CREATE JOIN INDEX ord_cust_idx AS
  SELECT  c_nationkey, o_orderdate, MIN(o_totalprice) AS min_price,
          MAX(o_totalprice) AS max_price
  FROM    order_tbl, customer
  WHERE   o_custkey = c_custkey
  GROUP   BY c_nationkey, o_orderdate
  ORDER   BY o_orderdate;
```

The following query is provided as an example to show how the Optimizer rewrites the query to use *ord_cust_idx* to replace the base tables. The phrase *SUM step* represents any aggregate expression.

This example performs a MIN aggregation on the aggregate join index, grouping on the *c_nationkey* column. An EXPLAIN of the SELECT statement includes a SUM step to aggregate from join index table *ord_cust_idx* with a condition of ("(ord_cust_idx.O_ORDERDATE > DATE '1998-09-20') AND (ord_cust_idx.O_ORDERDATE < DATE '1998-10-15')").

```
SELECT COUNT(*), MIN(o_totalprice)
      FROM order_tbl, customer
      WHERE o_custkey = c_custkey
      AND o_orderdate > DATE '1998-09-20'
      AND o_orderdate < DATE '1998-10-15'
      GROUP BY c_nationkey;
```

Example: Creating a Join Index With a Single-Level Partitioned Primary Index

Assume the following base table definition:

```
CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk         CHARACTER(16),
  o_shippriority  INTEGER,
  o_comment       VARCHAR(79))
UNIQUE PRIMARY INDEX (o_orderkey, o_orderdate);
```

The following SQL text creates a single-level PPI join index on the base table named *orders*. Because the *o_totalprice* column has a data type of DECIMAL(13,2), *o_totalprice* values can have more digits than an INTEGER type can handle, so you might see errors when you insert values into *o_totalprice* because the definition of *ordJI1* casts *o_totalprice* values as INTEGER values in its partitioning expression.

```
CREATE JOIN INDEX ordJI1 AS
  SELECT o_custkey, o_totalprice
  FROM orders
PRIMARY INDEX (o_custkey)
PARTITION BY RANGE_N(CAST(o_totalprice AS INTEGER)
                     BETWEEN 0
                     AND 999999
                     EACH 100, NO RANGE);
```

Example: SLPPI Join Index and Performance

The following example shows how an uncompressed PPI join index can help query performance.

Assume a star schema with the following fact and dimension tables defined in the physical data model:

- A fact table named `sales` with columns `sale_date`, `store_id`, `prod_id`, and `amount`.
- The following three dimension tables.
 - `calendar`, with columns `dayofmth`, `mth`, and `yr` and a primary index defined on `yr` and `mth`.
 - `product`, with columns `prod_id` and `prod_category`.
 - `org`, with columns `store_id`, `area`, `division`, and `business_unit`.

You might want to create an aggregate join index with daily summary data like the following join index named `sales_summary` to answer a set of ad hoc queries:

```
CREATE JOIN INDEX sales_summary AS
  SELECT sale_date, prod_category, division,
         SUM(amount) AS daily_sales
  FROM calendar AS c, product AS p, org AS o, sales AS s
  WHERE c.dayofmth = s.sale_date
  AND   p.prod_id = s.prod_id
  AND   o.store_id = s.store_id
  AND   s.sale_date BETWEEN DATE '1991-01-01' AND DATE '2006-12-31'
  GROUP BY sale_date, prod_category, division
  PRIMARY INDEX(sale_date, prod_category, division)
  PARTITION BY RANGE_N(sale_date BETWEEN DATE '1991-01-01'
                       AND DATE '2006-12-31'
                       EACH INTERVAL '1' MONTH);
```

A wide range of queries can make use of this join index, especially when there are foreign key-primary key relationships defined between the fact table and the dimension tables that enable the join index to be used as a broad join index to cover queries over a subset of dimension tables.

Significant performance gains can be achieved with this join index when the Optimizer uses it to answer the queries that have equality or range constraints on the `sale_date` column. The system applies optimizations such as static row partition elimination and rowkey-based merge join to a PPI join index in the same way it does to a PPI base table.

For example, the following query needs to access only 12 of the 192 total partitions, which saves as much as 93.75 percent on disk reads, and proportional savings on elapsed time compared with an otherwise identical nonpartitioned primary index join index, which requires a full-table scan.

```
SELECT prod_category, SUM(amount) AS daily_sales
FROM calendar c, product p, sales s
WHERE c.dayofmth = s.sale_date
```

```

AND    p.prod_id = s.prod_id
AND    sale_date BETWEEN DATE '2006-01-01' AND DATE '2006-12-31'
GROUP BY prod_category;

```

While a join index often significantly improves query response times, you must also consider the overhead of maintaining the index when inserts, deletes, updates, and merges occur to the base table on which it is defined.

An uncompressed PPI join index might see significant improvement in maintenance performance because of PPI-related optimizations. For example, partitioning on the DATE column in the join index also helps insert performance if the transaction data are inserted into the sales table according to a time sequence.

The Teradata Parallel Transporter UPDATE operator, MultiLoad, and FastLoad cannot be used to load rows into base tables with join indexes. Because of this limitation, you must use either of the following workarounds:

- Use the Teradata Parallel Data Pump utility to load the rows into sales
- FastLoad the rows into a staging table and then use either an INSERT ... SELECT or MERGE request with error logging to load the rows into sales .

Because the inserted rows are clustered in the data blocks corresponding to the appropriate partitions, the number of data blocks the system must read and write is reduced compared to a nonpartitioned primary index join index, where the inserted rows scatter among all the data blocks.

On the other hand, it is possible that the maintenance of a PPI join index makes the index less efficient than it would be with a nonpartitioned primary index. For example, if the primary index of the join index does not include the partitioning columns, and a DELETE or UPDATE statement specifies a constraint on the primary index, the system must probe all partitions on the AMP to maintain the PPI join index as opposed to the fast primary index access for the join index when defined with a nonpartitioned primary index.

Example: Join Index Coverage

The following set of base tables, join indexes, queries, and EXPLAIN reports demonstrate how the referential integrity relationships among the underlying base tables in a join index definition influence whether the Optimizer selects the index for queries that reference fewer base tables than are referenced by the join index:

```

CREATE SET TABLE t1, NO FALLBACK, NO BEFORE JOURNAL,
                NO AFTER JOURNAL (
    x1 INTEGER NOT NULL,
    a1 INTEGER NOT NULL,
    b1 INTEGER NOT NULL,
    c1 INTEGER NOT NULL,
    d1 INTEGER NOT NULL,

```

```

    e1 INTEGER NOT NULL,
    f1 INTEGER NOT NULL,
    g1 INTEGER NOT NULL,
    h1 INTEGER NOT NULL,
    i1 INTEGER NOT NULL,
    j1 INTEGER NOT NULL,
    k1 INTEGER NOT NULL,
    CONSTRAINT ri1 FOREIGN KEY (a1, b1, c1) REFERENCES t2 (a2, b2, c2),
    CONSTRAINT ri2 FOREIGN KEY (d1) REFERENCES t3(d3),
    CONSTRAINT ri3 FOREIGN KEY (e1, f1) REFERENCES t4 (e4, f4),
    CONSTRAINT ri4 FOREIGN KEY (g1, h1, i1, j1)
        REFERENCES t5(g5, h5, i5, j5),
    CONSTRAINT ri5 FOREIGN KEY (k1) REFERENCES t6(k6));

CREATE SET TABLE t2, NO FALLBACK, NO BEFORE JOURNAL,
        NO AFTER JOURNAL (
    a2 INTEGER NOT NULL,
    b2 INTEGER NOT NULL,
    c2 INTEGER NOT NULL,
    x2 INTEGER)
    UNIQUE PRIMARY INDEX(a2, b2, c2);

CREATE SET TABLE t3, NO FALLBACK, NO BEFORE JOURNAL,
        NO AFTER JOURNAL (
    d3 INTEGER NOT NULL,
    x3 INTEGER)
    UNIQUE PRIMARY INDEX(d3);

CREATE SET TABLE t4, NO FALLBACK, NO BEFORE JOURNAL,
        NO AFTER JOURNAL (
    e4 INTEGER NOT NULL,
    f4 INTEGER NOT NULL,
    x4 INTEGER)
    UNIQUE PRIMARY INDEX(e4, f4);

CREATE SET TABLE t5, NO FALLBACK, NO BEFORE JOURNAL,
        NO AFTER JOURNAL (
    g5 INTEGER NOT NULL,
    h5 INTEGER NOT NULL,
    i5 INTEGER NOT NULL,
    j5 INTEGER NOT NULL,
    x5 INTEGER)
    UNIQUE PRIMARY INDEX(g5, h5, i5, j5);

```

```
CREATE SET TABLE t6, NO FALLBACK, NO BEFORE JOURNAL,
                    NO AFTER JOURNAL (
    k6 INTEGER NOT NULL,
    x6 INTEGER)
UNIQUE PRIMARY INDEX(k6);
```

The following join index definition left outer joins t1 to t3 on d1=d3 and then left outer joins that result to t6 on k1=k6:

```
CREATE JOIN INDEX ji_out AS
  SELECT d1, d3, k1, k6, x1
  FROM t1 LEFT OUTER JOIN t3 ON d1=d3
  LEFT OUTER JOIN t6 ON k1=k6;
```

Following is an example of a query where the Optimizer can use the join index, ji_out, because the outer joins in the join index are inner joined to the query tables on unique columns. The columns d1 and k1 are declared as foreign keys in t1. Columns d3, the primary key for t3, and k6 the primary key for t6, are declared as the unique primary index for those tables. An EXPLAIN of the SELECT statement includes a SUM step to aggregate from ji_out.

```
SELECT d1, SUM(x1)
FROM t1, t3
WHERE d1=d3
GROUP by 1;
```

The following simple join index definition specifies inner joins on tables t1, t3 and t6:

```
CREATE JOIN INDEX ji_in AS
  SELECT d1, d3, k1, k6, x1
  FROM t1, t3, t6
  WHERE d1=d3
  AND k1=k6;
```

In the query below, the Optimizer can use the join index, ji_in. An EXPLAIN of the SELECT statement includes a SUM step to aggregate from ji_in.

```
SELECT d1, SUM(x1)
FROM t1, t3
WHERE d1=d3
GROUP BY 1;
```

This aggregate join index definition specifies inner joins on tables t1, t2, and t4:

```
CREATE JOIN INDEX ji_in_aggr AS
SELECT a1, e1, SUM(x1) AS total
FROM t1, t2, t4
WHERE a1=a2
AND   e1=e4
AND   a1>1
GROUP BY 1, 2;
```

The Optimizer can use join index, `ji_in_aggr`, in its plan for the following query because it has the same join term as the query. An EXPLAIN of the SELECT statement includes a RETRIEVE step from `ji_in_aggr` with a condition of `("ji_in_aggr.a1 > 2) AND (ji_in_aggr.a1 >= 3")`.

```
SELECT a1, e1, SUM(x1) AS total
FROM t1, t2, t4
WHERE a1=a2
AND   e1=e4
AND   a1>2
GROUP BY 1, 2;
```

The Optimizer does not use the join index, `ji_in_aggr`, for the query below because the conditions `b1=b2` and `f1=f4` are not covered by the join index. The Optimizer specifies a full-table scan to retrieve the specified rows. An EXPLAIN of the SELECT statement includes a RETRIEVE step from `t1` by way of an all-rows scan with a condition of `("t1.a1 > 2")`, a JOIN step from `t2` using a product join, with a join condition of `("b1 = t2.b2")`, and then a JOIN step from `t4`.

```
SELECT a1, e1, SUM(x1) AS total
FROM t1, t2, t4
WHERE b1=b2
AND   f1=f4
AND   a1>2
GROUP BY 1, 2;
```

The aggregate join index, `ji_in_aggr`, below specifies inner joins on tables `t1`, `t3`, and `t6` using conditions that exploit a foreign key-primary key relationship between table `t1` and tables `t3` and `t6`, respectively:

```
CREATE JOIN INDEX ji_in_aggr AS
SELECT d1, k1, SUM(x1) AS total
FROM t1, t3, t6
WHERE d1=d3
AND   k1=k6
GROUP BY 1, 2;
```


The Optimizer includes the join index, `ji_in_aggr`, in its access plan for the query below even though `ji_in_aggr` is defined with an inner joined table, `t6`, that the query does not reference. This is acceptable to the Optimizer because of the foreign key-primary key relationship between `t1` and the extra table, `t6`, on columns `k1` and `k6`, which have a foreign key-primary key relationship and are explicitly defined as a foreign key and as the unique primary index for their respective tables. An EXPLAIN of the SELECT statement includes a SUM step to aggregate from `ji_in_aggr`.

```
SELECT d1, SUM(x1)
FROM t1, t3
WHERE d1=d3
GROUP BY 1;
```

The following join index definition left outer joins table `t1` with, in succession, tables `t2`, `t3`, `t4`, `t5`, and `t6` on a series of equality conditions made on foreign key-primary key relationships among the underlying base tables:

```
CREATE JOIN INDEX ji_out AS
SELECT a1, b1, c1, c2, d1, d3, e1, e4, f1, g1, h1, i1, j1, j5, k1,
       k6, x1
FROM t1
LEFT OUTER JOIN t2 ON  a1=a2
                   AND b1=b2
                   AND c1=c2
LEFT OUTER JOIN t3 ON  d1=d3
LEFT OUTER JOIN t4 ON  e1=e4
                   AND f1=f4
LEFT OUTER JOIN t5 ON  g1=g5
                   AND h1=h5
                   AND i1=i5
                   AND j1=j5
LEFT OUTER JOIN t6 ON  k1=k6;
```

Even though the query below references fewer tables than are defined in the join index, `ji_out`, the Optimizer includes the join index in its access plan because all the extra outer joins are defined on unique columns and the extra tables are the inner tables in the outer joins. An EXPLAIN of the SELECT statement indicates that the Optimizer plan includes a SUM step to aggregate from `ji_out`.

```
SELECT a1, b1, c1, SUM(x1)
FROM t1, t2
WHERE a1=a2
AND   b1=b2
AND   c1=c2
GROUP BY 1, 2, 3;
```

The following join index definition specifies all inner joins on tables t1, t2, t3, t4, t5 and t6 and specifies equality conditions on all the foreign key-primary key relationships among those tables:

```
CREATE JOIN INDEX ji_in AS
SELECT a1, b1, c1, c2, d1, d3, e1, e4, f1, g1, g5, h1, i1, j1, k1,
       k6, x1
FROM t1, t2, t3, t4, t5, t6
WHERE a1=a2
AND   b1=b2
AND   c1=c2
AND   d1=d3
AND   e1=e4
AND   f1=f4
AND   g1=g5
AND   h1=h5
AND   i1=i5
AND   j1=j5
AND   k1=k6;
```

Even though the join index, ji_in, references six tables with inner joins, the Optimizer includes the join index for the query below, which only references two of the six tables, because all of the conditions in the join index definition are based on foreign key-primary key relationships among the underlying base tables. An EXPLAIN of the SELECT statement indicates that the Optimizer plan includes a SUM step to aggregate from ji_in.

```
SELECT a1, b1, c1, SUM(x1)
FROM t1, t2
WHERE a1=a2
AND   b1=b2
AND   c1=c2
GROUP BY 1, 2, 3;
```

The following example shows how the Optimizer uses a join index with an extra inner join when the connections among the tables in its definition are appropriately defined.

Suppose you have the following table definitions:

```
CREATE SET TABLE fact (
  f_d1 INTEGER NOT NULL,
  f_d2 INTEGER NOT NULL,
  FOREIGN KEY (f_d1) REFERENCES WITH NO CHECK OPTION dim1 (d1),
  FOREIGN KEY (f_d2) REFERENCES WITH NO CHECK OPTION dim2 (d2))
UNIQUE PRIMARY INDEX (f_d1,f_d2);
```

```

CREATE SET TABLE dim1 (
  a1 INTEGER NOT NULL,
  d1 INTEGER NOT NULL,
  FOREIGN KEY (a1) REFERENCES WITH NO CHECK OPTION dim1_1 (d11))
  UNIQUE PRIMARY INDEX (d1);

CREATE SET TABLE dim2 (
  d1 INTEGER NOT NULL,
  d2 INTEGER NOT NULL)
  UNIQUE PRIMARY INDEX (d2);

CREATE SET TABLE dim1_1 (
  d11 INTEGER NOT NULL,
  d22 INTEGER NOT NULL)
  UNIQUE PRIMARY INDEX (d11);

```

In the join index defined below, the fact table is joined with dimension tables dim1 and dim2 by foreign key-primary key joins on fact.f_d1=dim1.d1 and fact.f_d2=dim2.d2. Dimension table dim1 is also joined with its dimension subtable dim1_1 by a foreign key-primary key join on dim1.a1=dim1_1.d11. A query on the fact and dim2 tables uses the join index, ji_all, because of its use of foreign key-primary key relationships among the tables:

```

CREATE JOIN INDEX ji_all AS
SELECT (COUNT(*) (FLOAT)) AS countstar, dim1.d1, dim1_1.d11, dim2.d2,
       (SUM(fact.f_d1) (FLOAT)) AS sum_f1
FROM fact, dim1, dim2, dim1_1
WHERE ((fact.f_d1 = dim1.d1)
AND    (fact.f_d2 = dim2.d2))
AND    (dim1.a1 = dim1_1.d11)
GROUP BY dim1.d1, dim1_1.d11, dim2.d2
PRIMARY INDEX (d1);

```

The results of the aggregate operations COUNT and SUM are typed as FLOAT. See CREATE JOIN INDEX in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

The Optimizer includes the join index, ji_all, in the query below. Even though table fact is the parent table of both tables dim1 and dim2, those tables are joined with fact on foreign key columns in the join index definition. Similarly, dim1 is joined to dim1_1 in the join index definition on a foreign key column. An EXPLAIN of the SELECT statement includes SUM step to aggregate from ji_all.

```

SELECT d2, SUM(f_d1)
FROM fact, dim2
WHERE f_d2=d2
GROUP BY 1;

```

Example: Exceptions to Join Index Coverage

Below are examples of exceptions to the general coverage rules for extra tables in a join index definition. See CREATE JOIN INDEX in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

In the following example, table t9 is the parent table of tables, t7 and t8. Generally, this relationship disqualifies a join index from covering any query with fewer tables than are referenced in the definition for that index. However, because t7 and t8 are joined on the FK columns ($y7 = x8$) in the join index definition, the Optimizer uses the index, ji, to cover the query:

```
CREATE SET TABLE t7(
  x7 INTEGER NOT NULL,
  y7 INTEGER NOT NULL,
  z7 INTEGER NOT NULL,
  CONSTRAINT r7 FOREIGN KEY (y7) REFERENCES WITH NO CHECK OPTION
    t9 (y9))
PRIMARY INDEX (x7);
```

```
CREATE SET TABLE t8(
  x8 INTEGER NOT NULL,
  y8 INTEGER NOT NULL,
  z8 INTEGER NOT NULL,
  CONSTRAINT r8 FOREIGN KEY (x8) REFERENCES WITH NO CHECK OPTION
    t9 (x9));
```

```
CREATE SET TABLE t9(
  x9 INTEGER NOT NULL UNIQUE,
  y9 INTEGER NOT NULL,
  z9 INTEGER NOT NULL)
UNIQUE PRIMARY INDEX(y9);
```

```
CREATE JOIN INDEX ji AS
  SELECT x7, y7, x8, y8, x9, y9
  FROM t7, t8, t9
  WHERE y7=x8
  AND   y7=y9
  AND   x8=x9;
```

An EXPLAIN of the SELECT statement includes a RETRIEVE step from ji with a condition of ("ji.x1 > 1").

```
SELECT x7, y7, x8, y8
FROM t7, t8
```

```
WHERE y7=x8
AND   x7>1;
```

As with the previous example, this example demonstrates how a join index can be used to cover a query when one table in the join index definition is a parent of two others if the tables are joined on foreign key-primary key relationships. t9 is the parent table of both t7 and t10. But because t7 and t10 are joined with t9 on the same PK column, by transitive closure, t7 and t10 are joined on $y7=x10$. The Optimizer can use the join index ji to cover the query:

```
CREATE SET TABLE t10(
  x10 INTEGER NOT NULL,
  y10 INTEGER NOT NULL,
  z10 INTEGER NOT NULL,
  CONSTRAINT r10 FOREIGN KEY (x10) REFERENCES WITH NO CHECK OPTION
    t9 (y9))
PRIMARY INDEX x10;
```

```
CREATE JOIN INDEX ji AS
  SELECT x7, y7, x10, y10, x9, y9
  FROM t7, t10, t9
  WHERE y7=y9
  AND   x10=y9;
```

An EXPLAIN of the SELECT statement includes a RETRIEVE step from ji with a condition of ("ji.x1 > 1").

```
SELECT x7, y7, x10, y10
FROM t7, t10
WHERE y7=x10
AND   x7>1;
```

Example: Creating a Join Index With a Partitioned PPI Using a CURRENT_DATE Function

Suppose you create the following table.

```
CREATE SET TABLE customer, NO FALLBACK (
  cust_name          CHARACTER(10),
  cust_no            INTEGER,
  policy_expiration_date DATE FORMAT 'YYYY/MM/DD')
PRIMARY INDEX (cust_no)
PARTITION BY CASE_N(policy_expiration_date >= CURRENT_DATE,
  policy_expiration_date <  CURRENT_DATE
```

```
AND policy_expiration_date >= CURRENT_DATE-
      INTERVAL '3' MONTH);
```

This example defines a PPI sparse join index named `j_sales` on `customer` to contain only the data of the current quarter:

```
CREATE JOIN INDEX j_sales AS
  SELECT *
  FROM sales
  WHERE sale_amt>=2000.00
  PRIMARY INDEX (store_ID)
  PARTITION BY CASE_N(sale_date>=CURRENT_DATE, NO CASE);
```

Example: Creating a Join Index With a Period Column in its Select List

This example creates a join index with a Period column, `d1`, in its select list.

Assume you have created table `t1` with the following definition.

```
CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER,
  c1 INTEGER,
  d1 PERIOD(DATE),
  e1 DATE
  f1 CHARACTER(100));
```

You can create a join index on this table that incorporates the Period column `d1` as follows.

```
CREATE JOIN INDEX ji_pdt AS
  SELECT b1, c1, d1
  FROM t1;
```

If you then submit a `HELP JOIN INDEX` request on `ji_pdt`, Vantage returns a report like the following.

```
HELP JOIN INDEX ji_pdt;
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.
```

Column Name	Type	Comment
b1	I	?
c1	I	?
d1	PD	?

The Optimizer can use column d1 in ji_pdt to map all query expressions that involve t1.d1 to the join index. For example, it can map `BEGIN(t1.d1)` to `BEGIN(ji_pdt.d1)`.

Example: Creating a Join Index With a BEGIN Bound Function in its Select List

This example also uses table t1 as defined in [Example: Creating a Join Index With a Period Column in its Select List](#) to create a join index with a BEGIN bound function in its select list.

```
CREATE JOIN INDEX ji_begin AS
  SELECT b1, c1, BEGIN(d1) AS b
  FROM t1;
```

If you then submit a `HELP JOIN INDEX ji_begin` request on `ji_begin`, the system returns a report like the following.

```
HELP JOIN INDEX ji_begin;
Column Name          Type      Comment
-----
b1                   I         ?
c1                   I         ?
b                    DA        ?
```

Column b in `ji_begin` can be used to map an identical query expression such as `BEGIN(d1)`.

For example, the Optimizer can take the following request:

```
SELECT b1, c1, BEGIN(d1)
FROM t1;
```

and map it to this request that uses the join index:

```
SELECT b1, c1, b
FROM ji_begin;
```

Or it can take the following request:

```
SELECT b1,c1
FROM t1
WHERE BEGIN(d1) < '2008-05-01';
```

and map it to this request that uses the join index:

```
SELECT b1,c1
FROM ji_begin
WHERE b < '2008-05-01';
```

Example: Creating a Join Index with a Multicolumn CASE Expression in its Select List

This example uses table t1 as defined in [Example: Creating a Join Index With a Period Column in its Select List](#) as well as the following tables to create a join index with a multicolumn CASE expression in its select list.

```
CREATE TABLE t2 (
  a2 INTEGER,
  b2 INTEGER,
  c2 INTEGER,
  d2 PERIOD(DATE),
  e2 DATE,
  f2 CHARACTER(100))
UNIQUE INDEX(b2);
CREATE TABLE t3 (
  a3 INTEGER,
  b3 INTEGER,
  c3 INTEGER);
CREATE JOIN INDEX ji_case AS
  SELECT a1,b1,c1,CASE
                                WHEN t1.b1=t2.b2
                                THEN t2.a2
                                ELSE t3.a3
                                END AS a
  FROM t1,t2,t3
  WHERE a1=a2
  AND a2=a3;
```

If you then submit a HELP JOIN INDEX request on ji_case, Vantage returns a report like the following:

```
HELP JOIN INDEX ji_case;
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.
Column Name          Type      Comment
-----
a1                    I         ?
b1                    I         ?
```


c1	I	?
a	I	?

As was done for column b in [Example: Creating a Join Index With a BEGIN Bound Function in its Select List](#), column a in this example can be used to map an identical query expression to the join index.

Multicolumn expressions in a join index definition affect its maintenance. For `ji_case`, the value of column a depends on `t1.b1`, `t2.a2`, `t2.b2`, and `t3.a3`, so join index maintenance is required whenever any of those columns is updated, and the join index update uses a spool file to resolve the expression. For example, a simple update on base table `t1` like the following request cannot be mapped to a direct update on `ji_case`, although the delete condition `a1=10` can be evaluated directly on the join index.

```
UPDATE t1
  SET b1=10
 WHERE a1=10;
```

Example: Creating a Join Index With a P_INTERSECT Expression in the Select List

This example creates a join index with a `P_INTERSECT` Period expression in its select list.

```
CREATE JOIN INDEX ji_p_intersect AS
  SELECT b1,c1,d1 P_INTERSECT PERIOD(
    DATE '2010-08-01',
    DATE '2011-08-01') AS prd
 FROM t1;
```

If you then submit a `HELP JOIN INDEX` request on `ji_p_intersect`, the system returns a report like the following:

```
HELP JOIN INDEX ji_p_intersect;
```

Column Name	Type	Comment
-----	-----	-----
b1	I	?
c1	I	?
prd	PD	?

Similar to [Example: Creating a Join Index With a BEGIN Bound Function in its Select List](#) and [Example: Creating a Join Index with a Multicolumn CASE Expression in its Select List](#), column `prd` in this example can be used to map identical query expressions to the join index. In addition, `ji_p_intersect.prd` can be used for the following mappings:

- The expression `t1.d1 P_INTERSECT <period>`, where `<period>` is a constant, can be mapped to expression `ji_p_intersect.prd P_INTERSECT <period>` if `<period>` is within the defined interval ('2010-08-01', '2011-08-01').
- The range condition `END(t1.d1) >= <prd_start> AND BEGIN(t1.d1) <= <prd_end>`, where `<prd_start>` and `<prd_end>` are constants, can be mapped to `END(ji_p_intersect.prd) >= <prd_start> AND BEGIN(ji_p_intersect.PRD)` if `PERIOD(<prd_start>, <prd_end>)` is within the defined interval ('2010-08-01', '2011-08-01').

Example: Creating an Aggregate Join Index With a Nested Expression in its Select List

This example creates an aggregate join index with a nested expression in its select list.

```
CREATE JOIN INDEX aji_extract_begin AS
  SELECT EXTRACT(MONTH FROM BEGIN(d1)) AS mth, SUM(b1) AS sumb
  FROM t1
  GROUP BY 1;
```

If you then submit a `HELP JOIN INDEX` request on `aji_extract_begin`, the system returns a report like the following:

```
HELP JOIN INDEX aji_extract_begin;
Column Name          Type      Comment
-----
CountStar            F         ?
mth                  I         ?
sumb                 F         ?
```

Similarly to [Example: Creating a Join Index With a BEGIN Bound Function in its Select List](#) and [Example: Creating a Join Index with a Multicolumn CASE Expression in its Select List](#), column `mth` in this example can be used to map identical query expressions to the join index.

Example: Creating a Unique Join Index

The examples in this set are based on the following database objects definitions.

```
CREATE TABLE t3 (
  a3 INTEGER,
  b3 INTEGER,
  c3 INTEGER);
CREATE TABLE upi_t4 (
  a4 INTEGER,
```

```

    b4 INTEGER,
    c4 INTEGER)
UNIQUE PRIMARY INDEX(a4);
CREATE TABLE usi_t5 (
    a5 INTEGER,
    b5 INTEGER,
    c5 INTEGER)
UNIQUE INDEX (b5);
CREATE JOIN INDEX uji AS
SELECT b3, ROWID
FROM t3
WHERE c3 BETWEEN 200 AND 1000
UNIQUE PRIMARY INDEX (b3);

```

Suppose you populate table *t3* with the following rows.

	t3		
	a3	b3	c3
	--	--	---
	5	6	200
	6	6	200
	7	8	300
	8	9	400
	9	10	500
	10	11	600

You then attempt to create the following unique join index, *uji*, on *t3*:

```

CREATE JOIN INDEX uji AS
  SELECT b3, ROWID
  FROM t3
  WHERE c3 BETWEEN 200 AND 1000
  UNIQUE PRIMARY INDEX (b3);
*** Failure 2801 Duplicate unique prime key error in  user_name.target table.
      Statement# 1, Info =0
*** Total elapsed time was 1 second.

```

The request aborts and returns an error because the first two existing rows in *t3* have duplicate values for column *b3*, which was defined as the UPI for *uji*. Requests that attempt to insert a row that has a duplicate value for the UPI of an underlying unique join index.

To remedy this situation, delete the first row in *t3* as follows.

```
DELETE t3
WHERE a3=5;
*** Delete completed. One row removed.
*** Total elapsed time was 2 seconds.
```

Create the unique join index *uji* successfully.

```
CREATE JOIN INDEX uji AS
  SELECT b3, ROWID
  FROM t3
  WHERE c3 BETWEEN 200 AND 1000
  UNIQUE PRIMARY INDEX (b3);
*** Index has been created.
*** Total elapsed time was 11 seconds.
```

Attempt to insert the following row into *t3*:

```
INSERT INTO t3 VALUES (5,6,200);
*** Failure 2801 Duplicate unique prime key error in  user_name
.uji.
Statement# 1, Info =0
*** Total elapsed time was 3 seconds.
```

This insert fails because the row you attempted to insert has a duplicate value for *b3*, which is the UPI for *uji*.

This example demonstrates how the Optimizer can use a unique join index, in this case *uji*, as a means to a two-AMP access path retrieval. The relevant EXPLAIN text is highlighted in boldface type.

```
EXPLAIN SELECT *
  FROM t3
  WHERE b3=10
  AND c3 BETWEEN 300 AND 400;
*** Help information returned. 6 rows.
*** Total elapsed time was 1 second.
```

Explanation

```
-----
1) First, we do a two-AMP RETRIEVE step from  user_name.t3  by way of
   uji "user_name.t3.b3 = 10" with a residual condition of
   ("(user_name.t1.c3 >= 300) AND (user_name.t1.c3 <= 400)").  The
   estimated time for this step is 0.01 seconds.
```

-> The row is sent directly back to the user as the result of statement 1. The total estimated time is 0.01 seconds.

The following EXPLAIN text demonstrates that the Optimizer can also use *uji* as a two-AMP access path for other queries against table *t3*. The relevant EXPLAIN text is highlighted in boldface type.

```
EXPLAIN SELECT c3, SUM(a3)
  FROM t3
 WHERE b3=10
 AND   c3 BETWEEN 300 AND 400
 GROUP BY 1;
```

Explanation

-
- 1) First, we do a two-AMP SUM step to aggregate from user_name.t3 **by way of uji** "user_name.t3.b3 = 10" with a residual condition of ("(user_name.t1.c3 <= 400) AND (user_name.t1.c3 >= 300)") , grouping by field1 (user_name.t3.c3). Aggregate Intermediate Results are computed locally, then placed in Spool 3. The size of Spool 3 is estimated with low confidence to be 1 row (25 bytes). The estimated time for this step is 0.09 seconds.
 - 2) Next, we do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 1 row (43 bytes). The estimated time for this step is 0.03 seconds.
 - 3) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

This example demonstrates how the Optimizer can use the unique join index *uji* to qualify for an access path in a two-AMP join from table *t3* to the UPI of table *upi_t4*. The relevant EXPLAIN text is highlighted in boldface type.

```
EXPLAIN SELECT *
  FROM t3, upi_t4
 WHERE t3.b3=1
 AND   t3.c3 BETWEEN 300 AND 400
 AND   t3.c3=upi_t4.a4;
```

Explanation

-
- 1) First, we do a two-AMP JOIN step from user_name.t3 **by way of uji** "user_name.t3.b3 = 1" with a residual condition of (

"(user_name.t3.c3 >= 300) AND (user_name.t3.c3 <= 400)"), which is joined to user_name.upi_t4 by way of the primary index "user_name.upi_t4.a4 = user_name.t3.c3" with a residual condition of "(user_name.upi_t4.a4 >= 300) AND (user_name.upi_t4.a4 <= 400)". user_name.t3 and user_name.upi_t4 are joined using a nested join, with a join condition of "(1=1)". The result goes into Spool 1 (one-amp), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (111 bytes). The estimated time for this step is 0.03 seconds.

-> The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.03 seconds.

This example demonstrates how the Optimizer can use the unique join index of table *t3* to qualify for an access path in a two-AMP join from table *t3* to the UPI of table *usi_t5*, *indicating the similarity of using a USI for two-AMP access with using a unique join index for the same purpose*. The relevant EXPLAIN text is highlighted in boldface type.

```
EXPLAIN SELECT *
      FROM t3, usi_t5
     WHERE t3.b3=1
      AND  t3.c3 BETWEEN 300 AND 400
      AND  t3.c3=usi_t5.b5;
```

Explanation

1) First, we do a two-AMP JOIN step from user_name.t3 **by way of uji** "user_name.t3.b3 = 1" with a residual condition of "(user_name.t3.c3 >= 300) AND (user_name.t3.c3 <= 400)", **which is joined to user_name.usi_t5 by way of uji** "user_name.usi_t5.b5 = user_name.t3.c3" with a residual condition of "(user_name.usi_t5.b5 >= 300) AND (user_name.usi_t5.b5 <= 400)". user_name.t3 and user_name.usi_t5 are joined using a nested join, with a join condition of ("user_name.t3.c3 = user_name.usi_t5.b5"). The result goes into Spool 1 (one-amp), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (87 bytes). The estimated time for this step is 0.03 seconds.

-> The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.03 seconds.

This example demonstrates how the Optimizer can use the UPI of table *upi_t4* to qualify for a join to table *t3* using the unique join index *uji* as an access path in a two-AMP nested join from table *t3* to the UPI of table *upi_t4*. The relevant EXPLAIN text is highlighted in boldface type.

```

EXPLAIN SELECT *
      FROM t3,upi_t4
     WHERE upi_t4.a4=1
     AND   upi_t4.b4=t3.b3
     AND   t3.c3 BETWEEN 300 AND 400;

```

Explanation

-
- 1) First, we do a single-AMP JOIN step from user_name.upi_t4 by way of the unique primary index "user_name.upi_t4.a4 = 1" with no residual conditions, which is joined to user_name.t3 **by way of unique join index uji** "user_name.t3.b3 = user_name.upi_t4.b4" with a residual condition of ("(NOT (user_name.t3.b3 IS NULL)) AND ((user_name.t3.c3 >= 300) AND (user_name.t3.c3 <= 400))"). user_name.upi_t4 and user_name.t3 are joined using a nested join, with a join condition of ("(1=1)"). The result goes into Spool 1 (one-amp), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (177 bytes). The estimated time for this step is 0.03 seconds.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.03 seconds.

This example qualifies for a two-AMP join qualified on the USI of *usi_t5* and then joins to base table *t3* via *uji*.

```

EXPLAIN SELECT *
      FROM t3, usi_t5
     WHERE usi_t5.b5=1
     AND   usi_t5.c5=t3.b3
     AND   t3.c3 BETWEEN 300 AND 400;

```

Explanation

-
- 1) First, we do a two-AMP JOIN step from user_name.usi_t5 **by way of unique join index uji** "user_name.usi_t5.b5 = 1" with a residual condition of ("NOT (user_name.usi_t5.c5 IS NULL)"), which is joined to user_name.t3 by way of unique index uji "user_name.t3.b3 = user_name.usi_t5.c5" with a residual condition of ("(user_name.t3.c3 <= 400) AND (user_name.t3.c3 >= 300)"). user_name.usi_t5 and user_name.t3 are joined using a nested join, with a join condition of ("(1=1)"). The result goes into Spool 1 (one-amp), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 1 row (87 bytes). The estimated time for this step is 0.03

seconds.

-> The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.03 seconds.

Example: Creating a Join Index Using an Updatable CURRENT_DATE Function

You can define the partitioning expression for a join index that uses an updatable current date or current timestamp function. This enables the partition that contains the most recent data to be as narrow as possible, thus enabling more efficient access to the rows in that partition.

An additional benefit of updatable current dates and updatable current timestamps is that a partitioning expression based on them does not need to be redefined as time passes. Instead, you can just update the expressions and partitioning using an ALTER TABLE TO CURRENT request (see [ALTER TABLE TO CURRENT](#)) to reconcile the date or timestamp values to a newer date or timestamp, providing a simple way to define and maintain the partitioning expression for a join index and avoiding the need to drop and recreate an index whose current date or timestamp values are no longer current.

Suppose you create the following table on January 1, 2009.

```
CREATE SET TABLE customer, NO FALLBACK (
  cust_name          CHARACTER(10),
  cust_no            INTEGER,
  policy_expiration_date DATE FORMAT 'YYYY/MM/DD')
PRIMARY INDEX (cust_no)
PARTITION BY CASE_N(policy_expiration_date>=CURRENT_DATE,
                    policy_expiration_date<CURRENT_DATE
                    AND policy_expiration_date>=CURRENT_DATE-
                    INTERVAL '3' MONTH);
```

On January 1, 2009, you decide to create a join index on *customer* with a partitioning expression that specifies a updatable CURRENT_DATE function. The index is designed to contain only the data for the current quarter:

```
CREATE JOIN INDEX j_sales AS
  SELECT *
  FROM sales
  WHERE sale_amt>=2000.00
PRIMARY INDEX (store_ID)
PARTITION BY CASE_N(sale_date >= CURRENT_DATE, NO CASE);
```

With a resolved CURRENT_DATE of January 1, 2009, join index *j_sales* contains the following rows.

store_ID	amount	sale_date	PARTITION
1	2000.00	2009-01-01	1
1	3500.00	2009-01-15	1
1	2000.00	2008-12-15	2
1	5000.00	2009-04-01	1

On April 1, 2009, you submit the following ALTER TABLE TO CURRENT request for *j_sales*.

```
ALTER TABLE j_sales TO CURRENT;
```

The rows in the join index are reconciled as follows.

store_ID	amount	sale_date	PARTITION
1	2000.00	2009-01-01	2
1	3500.00	2009-01-15	2
1	2000.00	2008-12-15	2
1	5000.00	2009-04-01	1

See [Example: Altering the Partitioning of a Join Index Using ALTER TABLE TO CURRENT](#) for an example of how you can reconcile the partitioning for this join index using an ALTER TABLE TO CURRENT request.

Example: Altering the Partitioning of a Join Index Using ALTER TABLE TO CURRENT

Assume that you have created the following sales table on January 1, 2009.

```
CREATE SET TABLE sales, NO FALLBACK (
  storeID    INTEGER,
  amount     DECIMAL(10,2),
```

```
sale_date DATE FORMAT 'YYYY/MM/DD')
PRIMARY INDEX (storeID)
PARTITION BY CASE_N(sale_date>=CURRENT_DATE/*latest quarter data*/,
                    sale_date<CURRENT_DATE
                    AND sale_date>=CURRENT_DATE-INTERVAL '3' MONTH,
                    NO CASE);
```

You then define a PPI sparse join index `j_sales` on January 1, 2009 to contain only the data of the current quarter.

```
CREATE JOIN INDEX j_sales AS
SELECT *
FROM sales
WHERE sale_date > CURRENT_DATE - INTERVAL '3' DAY
PRIMARY INDEX (store_ID)
PARTITION BY CASE_N(sale_date >= CURRENT_DATE, NO CASE);
```

Join index `j_sales` contains the following rows assuming that the resolved `CURRENT_DATE` is January 1, 2009:

j_sales			
store_ID	amount	sale_date	PARTITION
1	2000.00	2009-01-01	1
1	3500.00	2009-01-15	1
1	2000.00	2008-12-15	2
1	5000.00	2009-04-01	1

On April 1, 2009, you submit the following ALTER TABLE TO CURRENT request on `j_sales`.

```
ALTER TABLE j_sales TO CURRENT;
```

The rows in join index `j_sales` are reconciled as follows:

j_sales			
store_ID	amount	sale_date	PARTITION
1	2000.00	2009-01-01	2
1	3500.00	2009-01-15	2
1	2000.00	2008-12-15	2
1	5000.00	2009-04-01	1

Example: Creating a Join Index Defined On a Period Column

In this example, column *d1* has a Period data type.

```
CREATE JOIN INDEX ji_pdt AS
  SELECT b1, c1, d1
  FROM t1;
```

Example: Creating a Join Index With a BEGIN Expression in its Select List

This example specifies an aliased BEGIN expression in its select list.

```
CREATE JOIN INDEX ji_begin AS
  SELECT b1, c1, BEGIN(d1) AS b
  FROM t1;
```

Example: Creating a Join Index With a Character Partitioning Expression

This example creates join index *t_cppi* with a character partitioning expression. The data type for column *j* in this example is CHARACTER and the type of column *k* is INTEGER.

```
CREATE JOIN INDEX t_cppi AS
  SELECT j, k
  FROM t1
  PRIMARY INDEX(j)
  PARTITION BY (CASE_N(j BETWEEN 'aaaa'
                        AND      'bbbb',
                        j BETWEEN 'cccc'
                        AND      'dddd'),
                RANGE_N(k BETWEEN 1
                        AND 10
                        EACH 1));
```

Example: Partitioning Expression that Specifies an AT LOCAL Date

Assume that column *tsz* has a TIMESTAMP(0) WITH TIME ZONE data type and the current session time zone is INTERVAL '-03:00' HOUR TO MINUTE when the join index that contains *tsz* is created with the following PARTITION BY clause. The data type of the partitioning expression is INTEGER.

```
PARTITION BY RANGE_N(CAST(tsz AS DATE AT LOCAL)
                     BETWEEN DATE '2003-01-01'
                     AND      DATE '2009-12-31'
                     EACH INTERVAL '1' MONTH)
```

Vantage implicitly rewrites this partitioning expression, replacing LOCAL with a specific time zone as follows.

```
PARTITION BY RANGE_N(CAST(tsz AS DATE AT '-03:00')
                     BETWEEN DATE '2003-01-01'
                     AND      DATE '2009-12-31'
                     EACH INTERVAL '1' MONTH)
```

Example: Creating Join Indexes With a UDT Column as a Primary or Secondary Index

These examples show how UDT columns can be used to define primary and secondary indexes for a join index.

First, define the UDTs that are used for the example columns.

The type `tbl_integer` is a distinct UDT based on the `INTEGER` data type.

```
CREATE TYPE tbl_integer AS INTEGER FINAL;
```

The type `tbl_char50` is also a distinct UDT and is based on the `CHARACTER` data type.

```
CREATE TYPE tbl_char50 AS CHARACTER(50) FINAL;
```

Then, you create the following tables.

```
CREATE TABLE t1_jupi (
  id      tbl_integer,
  emp_name tbl_char50)
UNIQUE PRIMARY INDEX(id);
CREATE TABLE t1_jusi (
  emp_name tbl_char50,
  id      tbl_integer)
UNIQUE INDEX(id);
```

The first example creates an uncompressed join index with both a NUPI and a NUSI defined on a UDT column.

```
CREATE JOIN INDEX t1_jindx AS
  SELECT t1_jupi.id AS id_1, t1_jupi.emp_name AS name_1,
         t1_jusi.id AS id_2, t1_jusi.emp_name AS name_2
  FROM t1_jupi, t1_jusi
  WHERE t1_jupi.id = t1_jusi.id
  PRIMARY INDEX (id_1),
  INDEX (id_2);
```

The second example creates a row-compressed join index with both a NUPI and a NUSI defined on a UDT column. In this example, the repeating column set is t1_jusi.id and t1_jusi.emp_name. The fixed column set is t1_jupi.id and t1_jupi.emp_name.

```
CREATE JOIN INDEX t3_jindx AS
  SELECT (t1_jusi.id AS id_2, t1_jusi.emp_name AS name_2)
         (t1_jupi.id AS id_1, t1_jupi.emp_name AS name_1),

  FROM t1_jusi, t1_jupi
  WHERE t1_jusi.id = t1_jupi.id
  PRIMARY INDEX(id_1),
  INDEX(id_2);
```

The third example creates a join index with a UPI defined on a UDT column.

```
CREATE JOIN INDEX t1_upi_jindx AS
  SELECT t1_jupi.id AS id_1, t1_jupi.emp_name AS name_1
  FROM t1_jupi
  UNIQUE PRIMARY INDEX (name_1);
```

Example: Creating a Join Index for a Table with JSON Columns

The table definition for this example includes two JSON columns, jsn1 and jsn2. The jsn1 column is non-LOB and jsn2 in an LOB column.

```
CREATE TABLE jsonTable (id INTEGER,
                        jsn1 JSON(1000) CHARACTER SET LATIN,
                        jsn2 JSON(1M) INLINE LENGTH 30000 CHARACTER SET LATIN);
```

The join index includes the id column and the jsn1 column.

```
CREATE JOIN INDEX jsonJI AS SELECT id, jsn1 FROM jsonTable;
```

Example: Creating Column-Partitioned Join Indexes

Assume that the following table has been created for this example set.

```
CREATE TABLE t1 (
  a INTEGER,
  b INTEGER,
  c INTEGER,
  d INTEGER)
PRIMARY INDEX (a);
```

The following CREATE JOIN INDEX requests are equivalent to one another.

This example creates a sparse join index but does not specify an explicit format for the column partition that contains *rw*. The format for the partition is system-determined to be COLUMN.

```
CREATE JOIN INDEX jt1 AS
  SELECT ROWID AS rw, a, b
  FROM t1
  WHERE a < 10
  PARTITION BY COLUMN (rw NO AUTO COMPRESS, ROW(a, b));
```

This example is identical to the previous example except that it explicitly specifies both COLUMN format for the column partition that contains *rw* and the optional NO PRIMARY INDEX clause.

```
CREATE JOIN INDEX jt1 AS
  SELECT ROWID AS rw, a, b
  FROM t1
  WHERE a < 10
  NO PRIMARY INDEX
  PARTITION BY COLUMN (COLUMN rw NO AUTO COMPRESS, ROW(a, b));
```

This example does not specify an explicit format for the column partition that contains *rw*. In this case, the format for the partition that contains *rw* is system-determined to be COLUMN. The example also adds the optional NO PRIMARY INDEX clause followed by an optional COMMA character to the request in the first example.

```
CREATE JOIN INDEX jt1 AS
  SELECT ROWID AS rw, a, b
  FROM t1
  WHERE a < 10
```

```
NO PRIMARY INDEX,
PARTITION BY COLUMN (rw NO AUTO COMPRESS, ROW(a,b));
```

This example is identical to the previous example except that it explicitly specifies COLUMN format for the column partition that contains *rw* in the partitioning expression and it moves the NO PRIMARY INDEX specification after the partitioning.

```
CREATE JOIN INDEX jt1 AS
  SELECT ROWID AS rw, a, b
  FROM t1
  WHERE a < 10
  PARTITION BY COLUMN (COLUMN rw NO AUTO COMPRESS, ROW(a, b)),
  NO PRIMARY INDEX;
```

This example is identical to the previous example except that it specifies (ROWID AS *rw*) as a group, specifies NO AUTO COMPRESS for (ROWID AS *rw*) and it does not specify NO PRIMARY INDEX.

```
CREATE JOIN INDEX jt1 AS
  SELECT (ROWID AS rw) NO AUTO COMPRESS, ROW(a, b)
  FROM t1
  WHERE a<10
  PARTITION BY COLUMN;
```

This example is identical to the previous example except that it explicitly specifies NO PRIMARY INDEX before the PARTITION BY clause.

```
CREATE JOIN INDEX jt1 AS
  SELECT (ROWID AS rw) NO AUTO COMPRESS, ROW(a, b)
  FROM t1 WHERE a<10
  NO PRIMARY INDEX
  PARTITION BY COLUMN;
```

This example is identical to the previous example except that it specifies a COMMA character between the NO PRIMARY INDEX and PARTITION BY clauses. The COMMA character is required if the preceding item in the index list is a partitioning clause that is not part of an index clause, but is otherwise optional.

```
CREATE JOIN INDEX jt1 AS
  SELECT (ROWID AS rw) NO AUTO COMPRESS, ROW(a, b)
  FROM t1
  WHERE a<10
  NO PRIMARY INDEX,
  PARTITION BY COLUMN;
```

This example is identical to the previous example except that it reorders the placement of the PARTITION BY and NO PRIMARY INDEX clauses from the request in the previous example.

```
CREATE JOIN INDEX jt1 AS
  SELECT (ROWID AS rw) NO AUTO COMPRESS, ROW(a, b)
  FROM t1
  WHERE a < 10
  PARTITION BY COLUMN,
  NO PRIMARY INDEX;
```

The following set of CREATE JOIN INDEX requests is equivalent to the previous set of CREATE JOIN INDEX requests except that the format for the column partition containing *rw* is explicitly specified to be COLUMN rather than system-determined to be COLUMN.

```
CREATE JOIN INDEX jt1 AS
  SELECT ROWID AS rw, a, b
  FROM t1
  WHERE a<10
  PARTITION BY COLUMN (COLUMN rw NO AUTO COMPRESS, ROW(a,b));
CREATE JOIN INDEX jt1 AS
  SELECT ROWID AS rw, a, b
  FROM t1
  WHERE a<10
  NO PRIMARY INDEX
  PARTITION BY COLUMN (COLUMN rw NO AUTO COMPRESS, ROW(a,b));
CREATE JOIN INDEX jt1 AS
  SELECT ROWID AS rw, a, b
  FROM t1
  WHERE a<10
  NO PRIMARY INDEX,
  PARTITION BY COLUMN (COLUMN rw NO AUTO COMPRESS, ROW(a,b));
CREATE JOIN INDEX jt1 AS
  SELECT ROWID AS rw, a, b
  FROM t1
  WHERE a<10
  PARTITION BY COLUMN (COLUMN rw NO AUTO COMPRESS, ROW(a,b)),
  NO PRIMARY INDEX;
CREATE JOIN INDEX jt1 AS
  SELECT COLUMN(ROWID AS rw) NO AUTO COMPRESS, ROW(a, b)
  FROM t1
  WHERE a<10
  PARTITION BY COLUMN;
CREATE JOIN INDEX jt1 AS
  SELECT COLUMN(ROWID AS rw) NO AUTO COMPRESS, ROW(a, b)
```



```

FROM t1
WHERE a<10
NO PRIMARY INDEX PARTITION BY COLUMN;
CREATE JOIN INDEX jt1 AS
  SELECT COLUMN(ROWID AS rw) NO AUTO COMPRESS, ROW(a, b)
  FROM t1
  WHERE a<10
NO PRIMARY INDEX,
PARTITION BY COLUMN;
CREATE JOIN INDEX jt1 AS
  SELECT COLUMN(ROWID AS rw) NO AUTO COMPRESS, ROW(a, b)
  FROM t1
  WHERE a<10
PARTITION BY COLUMN,
NO PRIMARY INDEX;

```

Example: Creating Column-Partitioned Sparse Join Index With Autocompression Based on the Default

This example uses the following base table.

```

CREATE TABLE acd_t1 (
  a INTEGER,
  b INTEGER,
  c INTEGER,
  d INTEGER)
PRIMARY INDEX(a);

```

You create the following column-partitioned sparse join index on *t1*.

```

CREATE JOIN INDEX j_acd_t1 AS
  SELECT ROWID AS rw, a, b
  FROM acd_t1
  WHERE a < 10
PARTITION BY COLUMN;

```

This defines a column-partitioned join index where each column is contained within its own column partition and with system-determined COLUMN format.

Because neither AUTO COMPRESS nor NO AUTO COMPRESS is specified in the join index definition, partitions will have autocompression, which is the default.

Example: Creating a Join Index with a Sparse Map

This example uses the following customer and orders tables.

```
CREATE TABLE customer (
  c_custkey INTEGER,
  c_name CHARACTER(26) not null,
  c_address VARCHAR(41),
  c_nationkey INTEGER,
  c_phone CHARACTER(16),
  c_acctbal DECIMAL(13,2),
  c_mktsegment CHARACTER(21),
  c_comment VARCHAR(127))
PRIMARY INDEX( c_custkey );
```

```
CREATE TABLE orders (
  o_orderkey INTEGER,
  o_date DATE FORMAT 'yyyy-mm-dd',
  o_status CHARACTER(1),
  o_custkey INTEGER,
  o_totalprice DECIMAL(13,2),
  o_orderpriority CHARACTER(21),
  o_clerk CHARACTER(16),
  o_shippriority INTEGER,
  o_comment VARCHAR(79))
UNIQUE PRIMARY INDEX(o_orderkey);
```

This statement creates a row-compressed join index using a sparse map.

```
CREATE JOIN INDEX ord_cust_idx, MAP=SmallTableMap AS
SELECT (o_custkey, c_name), (o_status, o_date, o_comment)
FROM orders, customer
WHERE o_custkey = c_custkey;
```

Related Information

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for a capsule overview of join indexes and *Teradata Vantage™ - Database Design*, B035-1094 for more examples, an extended discussion of the functions and features of join and partitioned primary indexes, and an overview of disk I/O integrity checking.

See *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for descriptions of the various forms of partition elimination.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for descriptions of the various functions and expressions that can be used to define a join index.

See *Teradata Vantage™ - Database Design*, B035-1094 for information about the various design issues for join indexes.

See CREATE JOIN INDEX in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for information about join indexes beyond syntax and examples.

Also see [ALTER TABLE TO CURRENT](#).

CREATE HASH INDEX

Creates a hash index on a base table.

Note:

Create an equivalent single-table join index rather than a hash index.

ANSI Compliance

CREATE HASH INDEX is a Teradata extension to the ANSI SQL:2011 standard. The ANSI SQL standard does not define DDL for creating indexes.

Required Privileges

You must have the CREATE TABLE privilege on the database in which the hash index is created.

You must have the INDEX or DROP TABLE privilege on the indexed base table or its containing database.

The creator of a hash index is granted the following table privileges on the index:

- DROP TABLE
- INDEX
- DUMP
- RESTORE

Privileges Granted Automatically

None.

CREATE HASH INDEX Syntax

```
CREATE HASH INDEX [ database_name. | user_name. ] hash_index_name [, index ]
  ( column_name_1 [,...] ) ON table_name
  [ BY ( column_name_2 [,...] ) ]
  [ ORDER BY {
    VALUES [ ( column_name_3 [,...] ) ] |
    HASH ( column_name_3 [,...] ) |
```

```

    ( column_name_3 [, ...] )
  }
] [;]

```

index

```

{ MAP map_name [ COLOCATE USING colocation_name ] |
  [NO] FALLBACK [PROTECTION] |
  CHECKSUM = { ON | DEFAULT | OFF } |
  BLOCKCOMPRESSION = { AUTOTEMP | DEFAULT | MANUAL | NEVER }
} [, ...]

```

CREATE HASH INDEX Syntax Elements

database_name***user_name***

Name of the containing database or user for *hash_index_name* if other than the current database or user.

hash_index_name

Name given to the hash index created by this request.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Hash index names conform to the rules for naming tables, including explicit qualification with a database or user name.

A hash index need not be defined in the same database as the base table represented in its definition.

column_name_1

Name of a column to be contained in the hash index.

The maximum number of columns you can specify for a hash index, including the columns implicitly added to the index upon creation, is 64.

All columns defined in the *column_name_1* list must be from the base table on which the hash index is defined.

If you specify more than one column name, the index is created on the combined values of each column named. A combined maximum of 32 secondary, hash, and join indexes can be created for one table. This includes the system-defined secondary indexes used to implement PRIMARY KEY and UNIQUE constraints for nontemporal tables and the

single-table join indexes used to implement PRIMARY KEY and UNIQUE constraints for temporal tables.

The number of system-defined single-table join indexes contributed by PRIMARY KEY and UNIQUE constraints on temporal table columns counts against the combined limit of 32 secondary, hash, and join indexes per base data table.

Just as it does for secondary index definitions, Vantage extends the hash index definition transparently with additional elements that make it possible to provide an access path to the corresponding rows in the base table on which it is defined.

A column in a hash index cannot have any of the following data types:

- BLOB
- CLOB
- ARRAY/VARRAY
- VARIANT_TYPE
- Period
- LOB UDT
- Geospatial
- JSON
- DATASET

You cannot compress column values for a hash index.

If the base table is a row level security table, you must include all security constraint columns in the index definition.

You cannot qualify the columns in the *column_name_1* list.

You cannot specify the system-derived columns PARTITION or PARTITION#L *n* in the column name list. However, you can specify a user-defined column named *partition* or *partition#L n*, where *n* is a value from 1 through 62.

Each multicolumn NUSI defined with an ORDER BY clause counts as two consecutive indexes in this calculation.

See CREATE INDEX in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

table_name

Name of the base table on which the hash index is defined.

You cannot define a hash index on any of the following database objects:

- Column-partitioned tables
- Global temporary tables
- Global temporary trace tables
- Hash indexes

- Join indexes
- Journal tables
- NoPI tables
- Queue tables
- Views (neither recursive nor nonrecursive)
- Volatile tables

You cannot create a hash index on a PPI table if the partitions for the table are defined on PERIOD bound functions.

column_name_2

An optional, explicitly specified column set on which the hash index rows are distributed across the AMPs.

A column in a hash index cannot have a data type of BLOB, CLOB, LOB UDT, VARIANT_TYPE, ARRAY/VARRAY, Period, Geospatial, JSON, or DATASET.

You cannot specify the system-derived columns PARTITION or PARTITION#L *n* in the column name list. However, you can specify a user-defined column named partition or partition#L *n*, where *n* ranges from 1 through 62.

The column set you specify must be drawn from the column set specified by *column_name_1*.

If a hash index is defined without either a BY clause or an ORDER BY clause, then Vantage distributes its rows by the hash of the primary index of its base table if the primary index is defined on a UDT column.

You cannot specify the same column twice in the *column_name_2* list.

You cannot qualify any of the column names in the *column_name_2* list.

However, you can specify a user-defined column named *partition*.

You cannot compress column values for a hash index.

This column set acts as the primary index for the hash index: its rows are hashed on the column set that you specify.

If you do not specify this column set, then the hash index rows are hashed on the primary index column set of the base table on which the index is defined, which makes it hash-local with respect to its base table.

You cannot specify a partitioned column set. This means that you cannot create a PPI for hash indexes.

ORDER BY

Row ordering on each AMP: either value-ordered or hash-ordered.

If you do not specify an ORDER BY clause, then the hash index rows have the same ordering as that specified for the base table on which the index is defined.

You must specify an ORDER BY clause that includes an explicit column list for a hash index defined on a partitioned primary index table.

If you specify a BY clause, you must specify an ORDER BY clause for the hash index definition.

If you specify neither a BY clause nor an ORDER BY clause, Vantage distributes the hash index rows by the hash of the primary index column set of the base table on which they are defined.

As a result, the hash index is AMP-local and its rows are ordered by the hash of those columns. This is also true if the primary index of the base table is defined on a UDT column.

If you specify the system-derived PARTITION column or the system-derived PARTITION#L *n* columns, where *n* is a number from 1 through 62, the request returns an error to the requestor.

You can specify a user-defined column named partition or partition#L*n*.

If you specify ORDER BY VALUES with a *column_name_3* list, the *column_name_3* list is limited to a single numeric column with a size of 4 or fewer bytes.

In this case, the primary index of the base table must consist of a single column having a length of 4 or fewer bytes.

The base table primary index column does not have to be included in the *column_name_1* set.

You cannot specify ORDER BY VALUES on a UDT column.

The hash index rows are stored in ascending order by the values of the column specified as the primary index of the base table.

The values specified for the *column_name_3* list must have one of the following data types:

- BYTEINT
- DATE
- DECIMAL
- INTEGER
- SMALLINT

If you specify ORDER BY HASH, you must specify a BY clause and the columns specified for the *column_name_2* and *column_name_3* sets must be identical.

The hash index rows are ordered on the row hash value of these columns.

You can specify ORDER BY HASH on a UDT column.

If you specify ORDER BY without specifying either the HASH or the VALUES keywords, Vantage orders the rows by the default, which is VALUES.

In this case, you must specify a *column_name_3* list.

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for complete documentation of the ORDER BY clause.

VALUES

Value-ordering for the ORDER BY column.

Select VALUES to optimize queries that return a contiguous range of values, especially for a workload that requires a covering index or that commonly uses a nested join in its query plans.

You cannot specify ORDER BY VALUES on a UDT column.

HASH

Hash-ordering for the ORDER BY column.

Hash-ordering for the ORDER BY column.

column_name_3

Column set on which the index is to be ordered.

If you specify ORDER BY VALUES, then you can specify only one column for *column_name_3*.

The specified column set must be a subset of *column_name_1*.

You cannot specify the system-derived columns PARTITION or PARTITION#L *n* in the column name list. However, you can specify a user-defined column named partition or partition#L *n*, where *n* ranges from 1 through 62.

You cannot specify the same column twice in the *column_name_3* list.

You cannot compress column values for a hash index.

You cannot order an index by a Period, BLOB, CLOB, LOB UDT, Geospatial, VARIANT_TYPE, or ARRAY/VARRAY column.

index

MAP

You can specify an existing contiguous or sparse map for the hash index.

For a sparse map, you can specify a colocation name. A hash index can have different map or colocation name than the base tables.

map_name

Name of an existing contiguous or sparse map.

You cannot specify `TD_DataDictionaryMap` or `TD_GlobalMap`.

colocation_name

Name for colocating the hash index on the same AMPs with other tables, join indexes, or hash indexes. For example, you can colocate two tables, then join the tables on the primary index or primary AMP index columns.

You can only specify this option for a sparse map. For a contiguous map, the *colocation_name* is not needed for colocation and is set to `NULL`.

If you do not specify a colocation name, the name defaults to *database_index*, where *database* is the name of the database or user followed by an underscore (`_`) and *index* is the name of the hash index. If *database* exceeds 63 characters, *database* is truncated to 63 characters. If *index* exceeds 64 characters, *index* is truncated to 64 characters.

NO

The hash index does not use fallback protection.

Note:

You cannot use the `NO FALLBACK` option and the `NO FALLBACK` default on platforms optimized for fallback.

If a hash index is not fallback protected, an AMP failure prevents the following events from occurring:

- Processing queries on the hash index
- Updating the table on which the hash index is defined

FALLBACK

The hash index uses fallback protection.

The default is to use the fallback specification for the database in which the hash index is defined.

When a hardware read error occurs, the file system reads the fallback copy of the data and reconstructs the rows in memory on their home AMP. Support for Read From Fallback is limited to the following cases:

- Requests that do not attempt to modify data in the bad data block
- Primary subtable data blocks
- Reading the fallback data in place of the primary data. In some cases, Active Fallback can repair the damage to the primary data dynamically. In situations where the bad data block cannot be repaired, Read From Fallback substitutes an error-free fallback copy of the corrupt rows each time the read error occurs.

PROTECTION

Optional keyword.

CHECKSUM

A table-specific disk I/O integrity checksum level. The checksum setting applies to primary and fallback data rows for the hash index.

If you do not specify a value, the system assumes the system-wide default value for this table type. The result is identical to specifying DEFAULT. If you are changing the checksum for this table to the system-wide default value, then specify DEFAULT.

ON

Calculate checksums using the entire disk block. Sample 100% of the disk blocks to generate a checksum.

DEFAULT

The default setting is the current DBS Control checksum setting specified for this table type.

OFF

Disables checksum disk I/O integrity checks.

BLOCKCOMPRESSION

Specifies whether the data in the hash index should be block-compressed based on the temperature of the cylinders on which it is stored.

You can specify the following settings for *block_compression_option*.

AUTOTEMP

The compressed state of the data in the hash index can be changed by Vantage at any time based on its temperature.

You can still issue query band options or Ferret commands, but if the compressed state of the data does not match its temperature, such changes might be undone by the system over time.

DEFAULT

The hash index uses the compression option (MANUAL, AUTOTEMP or NEVER) set in the DBS Control parameter DefaultTableMode. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102. Note that value of DefaultTableMode is not saved in the hash index definition as part of a CREATE HASH INDEX request, so a hash index set to DEFAULT is affected by any future change to the DefaultTableMode parameter.

MANUAL

That block level compression is applied based on the default for the hash index at the time the hash index is created. Hash indexes can be compressed or uncompressed at any time after loading by using the Ferret COMPRESS and UNCOMPRESS commands. Data inserted into the existing hash index inherits the current compression status of the hash index at the time the data is inserted.

NEVER

The hash index is not compressed even if the DBS Control block compression settings indicate otherwise. Vantage rejects Ferret commands to manually compress the hash index, but Ferret commands to decompress the index are valid.

Usage Notes

Default Map for the User, Database, or Profile

The immediate owner of the hash index can be a user or a database. See the CREATE USER [DEFAULT MAP](#) option, the MODIFY USER [DEFAULT MAP](#) option, the CREATE DATABASE [DEFAULT MAP](#) option, the MODIFY DATABASE [DEFAULT MAP](#) option, the CREATE PROFILE [DEFAULT MAP](#) option, or the MODIFY PROFILE [DEFAULT MAP](#) option.

User, Database, or Profile **DEFAULT MAP OVERRIDE ON ERROR** Option

If the map you specify is not valid for any reason, (such as it does not exist, has not been granted to you, or is not in the same secure zone), Vantage either substitutes a default map or returns an error, subject to the values of the DEFAULT MAP settings for your PROFILE, USER, or DATABASE.

Default Map for a **CREATE HASH INDEX** Statement

If you do not specify the MAP option when creating a hash index, the system determines a default map for the hash index in following order of precedence:

- If the immediate owner is not the creator:
 - Default map, if defined, for the profile of the immediate owner.
 - Default map, if defined, for the immediate owner.
 - System-default map.
- Default map, if defined, for the profile of the creator.
- Default map, if defined, for the creator.
- System-default map.

Examples

Examples for Base Table With Unique Primary Index

The subsequent examples demonstrate hash indexes based on the following table definition:

```
CREATE TABLE orders (
  o_orderkey      INTEGER      NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE          FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk         CHARACTER(16),
  o_shippriority  INTEGER,
  o_comment       VARCHAR(79))
UNIQUE PRIMARY INDEX (o_orderkey);
```

Example: Hash Index With Checksum

The following hash index is created with disk I/O checksum ON:

```
CREATE HASH INDEX line_hidx_6, CHECKSUM=ON
  (l_shipdate) ON lineitem
BY (l_shipdate)
ORDER BY VALUES;
```

This creates an index that is equivalent in structure to the following join index:

```
CREATE JOIN INDEX line_jidx_6, CHECKSUM=ON AS
  SELECT (l_orderkey), (l_shipdate, lineitem.ROWID)
  FROM lineitem
  ORDER BY l_orderkey
  PRIMARY INDEX (l_shipdate);
```

Example: Creating a Hash Index According to Order Date

The following hash index is distributed on its primary index o_orderdate and stored in value order on o_orderdate.

```
CREATE HASH INDEX ord_hidx_1 (o_orderdate) ON orders
BY (o_orderdate)
ORDER BY (o_orderdate);
```

You could also specify ORDER BY VALUES instead of ORDER BY.

Note that the ORDER BY column list complies with the rules that it be limited to a single column, that the column must be in the column_name_1 list, must be a numeric type, and must be four or fewer bytes.

This creates an index that is equivalent in structure to the following single-table join index:

```
CREATE JOIN INDEX ord_jidx_1 AS
  SELECT (o_orderdate), (o_orderkey, orders.ROWID)
  FROM orders
  ORDER BY o_orderdate
  PRIMARY INDEX (o_orderdate);
```

Example: Creating a Hash Index According to Customer ID

The following hash index is distributed on its primary *index o_custkey* and stored in row-hash order on *o_custkey*:

```
CREATE HASH INDEX ord_hidx_2 (o_custkey) ON orders
BY (o_custkey)
ORDER BY HASH (o_custkey);
```

This creates an index that is equivalent in structure to the following join index:

```
CREATE JOIN INDEX ord_jidx_2 AS
  SELECT (o_custkey), (o_orderkey, orders.ROWID)
  FROM orders
  PRIMARY INDEX (o_custkey);
```

Example: Creating a Hash Index According to Order Date by Order ID

The following hash index is distributed on *o_orderkey* (the primary index for the base table Orders) and stored in value order on *o_orderkey* (the order key for the base table orders):

```
CREATE HASH INDEX ord_hidx_3 (o_orderdate) ON orders
ORDER BY VALUES;
```

Note that the ordering column of the base table complies with the rules that it be limited to a single numeric column four or fewer bytes in length.

This creates an index that is equivalent in structure to the following join index:

```
CREATE JOIN INDEX ord_jidx_3 AS
  SELECT o_orderkey, o_orderdate, orders.ROWID
  FROM orders
  ORDER BY o_orderkey
  PRIMARY INDEX (o_orderkey);
```

Example: Creating a Hash Index According to Order Date

The following hash index is distributed on o_orderkey (the primary index for the base table Orders) and stored in hash order on o_orderkey:

```
CREATE HASH INDEX ord_hidx_4 (o_orderdate) ON orders;
```

This creates an index that is equivalent in structure to the following join index:

```
CREATE JOIN INDEX ord_jidx_4 AS
  SELECT o_orderkey, o_orderdate, orders.ROWID
  FROM orders
  PRIMARY INDEX (o_orderkey);
```

Example: Creating a Hash Index on Orders by Order Date

The following hash index is distributed on o_orderkey (the primary index for the base table Orders) and stored in value order on o_orderdate:

```
CREATE HASH INDEX ord_hidx_5 (o_orderdate) ON orders
  ORDER BY (o_orderdate);
```

Equivalently, you could specify ORDER BY VALUES instead of ORDER BY.

This creates an index that is equivalent in structure to the following join index:

```
CREATE JOIN INDEX ord_jidx_5 AS
  SELECT (o_orderdate), (o_orderkey, orders.ROWID)
  FROM orders
  ORDER BY o_orderdate
  PRIMARY INDEX (o_orderkey);
```

Example: Creating a Hash Index According to Order Date

The following hash index is distributed on its primary index o_orderdate and stored in value order on o_orderkey (the primary index for the base table orders):

```
CREATE HASH INDEX ord_hidx_6 (o_orderdate) ON orders
  BY (o_orderdate)
  ORDER BY VALUES;
```

Note that the ordering column of the base table complies with the rules that it be limited to a single numeric column four or fewer bytes in length.

This creates an index that is equivalent in structure to the following join index.

```
CREATE JOIN INDEX ord_jidx_6 AS
  SELECT o_orderkey, o_orderdate, orders.ROWID
  FROM orders
  ORDER BY o_orderkey
  PRIMARY INDEX (o_orderdate);
```

Examples for Base Table With Nonunique Primary Index

The subsequent examples demonstrate hash indexes based on the following table definition:

```
CREATE TABLE lineitem (
  l_orderkey      INTEGER      NOT NULL,
  l_partkey       INTEGER      NOT NULL,
  l_suppkey       INTEGER,
  l_linenum       INTEGER,
  l_quantity      INTEGER      NOT NULL,
  l_extendedprice DECIMAL(13,2) NOT NULL,
  l_discount      DECIMAL(13,2),
  l_tax           DECIMAL(13,2),
  l_returnflag    CHARACTER(1),
  l_linestatus    CHARACTER(1),
  l_shipdate      DATE          FORMAT 'yyyy-mm-dd',
  l_commitdate    DATE          FORMAT 'yyyy-mm-dd',
  l_receiptdate   DATE          FORMAT 'yyyy-mm-dd',
  l_shipinstruct  VARCHAR(25),
  l_shipmode      VARCHAR(10),
  l_comment       VARCHAR(44))
  PRIMARY INDEX (l_orderkey);
```

Example: Creating a Hash Index According to Shipping Date

The following hash index is distributed explicitly on `l_shipdate` and stored explicitly in value order on `l_shipdate`:

```
CREATE HASH INDEX line_hidx_1 (l_shipdate) ON lineitem
  BY (l_shipdate)
  ORDER BY (l_shipdate);
```

Equivalently, you could specify `ORDER BY VALUES` instead of `ORDER BY`.

This creates an index that is equivalent in structure to the following join index:

```
CREATE JOIN INDEX line_jidx_1 AS
  SELECT (l_shipdate), (l_orderkey, lineitem.ROWID)
  FROM lineitem
  ORDER BY l_shipdate
  PRIMARY INDEX (l_shipdate);
```

Example: Creating a Hash Index According to Part Number

The following hash index is distributed explicitly on `l_partkey` and stored explicitly in row-hash order on `l_partkey`:

```
CREATE HASH INDEX line_hidx_2 (l_partkey) ON lineitem
  BY (l_partkey)
  ORDER BY HASH (l_partkey);
```

This creates an index that is equivalent in structure to the following join index.

```
CREATE JOIN INDEX line_jidx_2 AS
  SELECT (l_partkey), (l_orderkey, lineitem.ROWID)
  FROM lineitem
  PRIMARY INDEX (l_partkey);
```

Example: Creating a Hash Index According to Ship Date

The following hash index is distributed implicitly on `l_orderkey`, the primary index for the base table `lineitem`, and stored explicitly in value order on `l_orderkey`:


```
CREATE HASH INDEX line_hidx_3 (l_shipdate) ON lineitem
ORDER BY VALUES;
```

The ordering column of the base table complies with the rules that it be limited to a single numeric column four or fewer bytes in length.

This creates an index that is equivalent in structure to the following join index:

```
CREATE JOIN INDEX line_jidx_3 AS
  SELECT (l_orderkey), (l_shipdate, lineitem.ROWID)
  FROM lineitem
  ORDER BY l_orderkey
  PRIMARY INDEX (l_orderkey);
```

Example: Creating a Hash Index According to Ship Date by Order ID

The following hash index is distributed implicitly on l_orderkey, the primary index for the base table lineitem, and also stored implicitly in hash order on l_orderkey:

```
CREATE HASH INDEX line_hidx_4 (l_shipdate) ON lineitem;
```

This creates an index that is equivalent in structure to the following join index:

```
CREATE JOIN INDEX line_jidx_4 AS
  SELECT (l_orderkey), (l_shipdate, lineitem.ROWID)
  FROM lineitem
  PRIMARY INDEX (l_orderkey);
```

Example: Creating a Hash Index According to Ship Date

The following hash index is distributed implicitly on l_orderkey, the primary index for the base table lineitem, and stored explicitly in value order on l_shipdate:

```
CREATE HASH INDEX line_hidx_5 (l_shipdate) ON lineitem
ORDER BY (l_shipdate);
```

Equivalently, you could specify ORDER BY VALUES instead of ORDER BY.

Note that the ORDER BY column list complies with the rules that it be limited to a single column, that the column must be in the column_name_1 list, must be a numeric type, and must be four or fewer bytes.

This creates an index that is equivalent in structure to the following join index:

```
CREATE JOIN INDEX line_jidx_5 AS
  SELECT (l_shipdate), (l_orderkey, lineitem.ROWID)
  FROM lineitem
  ORDER BY l_shipdate
  PRIMARY INDEX (l_orderkey);
```

Example: Creating a Hash Index According to Ship Date by Order Number

The following hash index is distributed explicitly on `l_shipdate` and stored implicitly in value order on `l_orderkey`, the primary index for the base table `lineitem`:

```
CREATE HASH INDEX line_hidx_6 (l_shipdate) ON lineitem
  BY (l_shipdate)
  ORDER BY VALUES;
```

Note that the ordering column of the base table complies with the rules that it be limited to a single numeric column four or fewer bytes in length.

This creates an index that is equivalent in structure to the following join index:

```
CREATE JOIN INDEX line_jidx_6 AS
  SELECT (l_orderkey), (l_shipdate, lineitem.ROWID)
  FROM lineitem
  ORDER BY l_orderkey
  PRIMARY INDEX (l_shipdate);
```

Example: Hash Indexes With a UDT Column in their Column Lists

These examples show how UDT columns can be specified in the *column_name_list1*, *column_name_list2*, and *column_name_list3* for a hash index.

First define the UDTs that are used for the example columns.

`TBL_INTEGER` is a distinct UDT based on the `INTEGER` data type.

```
CREATE TYPE TBL_INTEGER AS INTEGER FINAL;
```

`TBL_CHAR50` is also a distinct UDT and is based on the `CHARACTER` data type.

```
CREATE TYPE TBL_CHAR50 AS CHARACTER(50) FINAL;
```

Assume you define the following table.

```
CREATE TABLE t1_hash_index (
  id          TBL_INTEGER,
  emp_name TBL_CHAR50)
PRIMARY INDEX(id);
```

The first hash index definition specifies the UDT column *emp_name* in all three of its column lists.

```
CREATE HASH INDEX tb_index_1 (emp_name)
ON t1_hashindex
BY (emp_name)
ORDER BY HASH (emp_name);
```

The second hash index definition specifies the UDT column *emp_name* in its *column_name_1* list.

This example shows how a hash index created without defining either a BY or ORDER BY clause is valid and its rows are distributed by the hash of the primary index of the base table on which it is defined when the primary index of the table is created on a UDT column.

```
CREATE HASH INDEX tb_index_2 (emp_name)
ON t1_hashindex;
```

Related Information

For an overview of Vantage hash indexes, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

For more examples and discussion of the functions and features of hash indexes, and for an overview of disk I/O integrity checking, see *Teradata Vantage™ - Database Design*, B035-1094.

Also, see [CREATE JOIN INDEX](#).

ALTER JOIN INDEX

Moves a join index from one map to another or changes the colocation for a sparse map.

Required Privileges

You must have been granted the specified map, except when the map you specify is the current map for the table, you are the table owner and you specify your default map, or you specify your default map. See GRANT MAP in *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

ALTER JOIN INDEX Syntax

```
ALTER JOIN INDEX [ database_name. | user_name. ] join_index_name ,
  MAP = map_name [ COLOCATE USING colocation_name ] [;]
```

Usage Notes

Altering the Join Index to a Sparse Map Without Specifying the Colocation Option

If you alter the join index to use a sparse map without specifying the COLOCATE USING clause and the join index previously used a sparse map, the colocation name does not change. Otherwise, the colocation name defaults to *database_index*, where *database* is the name of the database or user followed by an underscore (`_`) and *index* is the name of the join index. If *database* exceeds 63 characters, *database* is truncated to 63 characters. If *index* exceeds 64 characters, *index* is truncated to 64 characters.

Access Locks During the Alter Join Index Operation

A read lock is placed on the join index during the copy to the new map. The lock is upgraded to an exclusive lock while DBC.TVM is updated and the transaction is committed.

Indexes and the Alter Table Map Operation

Secondary indexes for the table are moved or rebuilt using the new map.

Join indexes on the base table are not moved when the table is moved to a new map. You must use ALTER JOIN INDEX.

System-defined join indexes for the table are moved or rebuilt using the new map.

Secure Zones and Sparse Maps

For a sparse map, you must be in the same secure zone as the sparse map.

ALTER HASH INDEX

Moves a hash index from one map to another or changes the colocation for a sparse map.

Required Privileges

You must have been granted the specified map, except when the map you specify is the current map for the table, you are the table owner and you specify your default map, or you specify your default map. See GRANT MAP in *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

ALTER HASH INDEX Syntax

```
ALTER HASH INDEX [ database_name. | user_name. ] hash_index_name ,
  MAP = map_name [ COLOCATE USING colocation_name ] [ ; ]
```

Usage Notes

Altering the Hash Index to a Sparse Map Without Specifying the Colocation Option

If you alter the hash index to use a sparse map without specifying the `COLOCATE USING` clause and the hash index previously used a sparse map, the colocation name does not change. Otherwise, the colocation name defaults to *database_index*, where *database* is the name of the database or user followed by an underscore (`_`) and *index* is the name of the hash index. If *database* exceeds 63 characters, *database* is truncated to 63 characters. If *index* exceeds 64 characters, *index* is truncated to 64 characters.

Locks During the Alter Hash Index Operation

A read lock is placed on the hash index during the copy to the new map. The lock is upgraded to an exclusive lock while `DBC.TVM` is updated and the transaction is committed.

Indexes and the Alter Table Map Operation

Secondary indexes for the table are moved or rebuilt using the new map.

Hash indexes on the base table are not moved when the table is moved to a new map. You must use `ALTER HASH INDEX`.

Secure Zones and Sparse Maps

For a sparse map, you must be in the same secure zone as the sparse map.

DROP INDEX

Drops a secondary index on a table or join index.

You can drop a named index by specifying the index name or index definition.

When you drop a secondary index, the system frees the disk space that was used by the index and removes information about the index from the table header. If statistics have been collected on the index, the system sets the value for the `IndexNumber` column in `DBC.StatsTbl` to 0, but does not drop the statistics that have been collected on the index column.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the `INDEX` or `DROP TABLE` privilege on the table.

To drop an index from a join index, you must have one of the following privileges.

- `DROP TABLE` on the join index
- `DROP TABLE` on one of the base tables

- INDEX on the join index
- INDEX on one of the base tables

DROP INDEX Syntax

```
DROP INDEX { index_specification | index_definition } [;]
```

index_specification

```
[ database_name. ] index_name  
ON { table_specification | join_index_specification }
```

index_definition

```
{ ( column_name [,...] ) | [ database_name. ] index_name }  
[ order_clause ]  
ON { table_specification | join_index_specification }
```

table_specification

```
[ TEMPORARY ] [ database_name. ] table_name
```

join_index_specification

```
[ database_name. ] join_index_name
```

order_clause

```
ORDER BY [ VALUES | HASH ] ( column_name_2 )
```

DROP INDEX Syntax Elements

index_specification

database_name

The containing database for *index_name* if different from the current database.

index_name

The name of the secondary index to be dropped.

table_specification**TEMPORARY**

That *table_name* refers to a global temporary table.

database_name

The containing database for *table_name* if different from the current database.

table_name

The table on which *index_name* is defined.

join_index_specification***database_name***

the containing database for *join_index_name* if different from the current database.

join_index_name

the join index on which *index_name* is defined.

index_definition***column_name***

A list of columns defined for the index.

The maximum number of columns you can specify is 64.

database_name

The name of the containing database for *index_name* if different from the current database.

index_name

The name of the index to be dropped.

table_specification

table on which the secondary index is to be dropped.

The named table cannot be one of the following table types.

- global temporary function trace
- journal
- volatile

TEMPORARY

You are dropping an index on a materialized global temporary table.

You can only specify this keyword when dropping an index on a global temporary table.

If you do not specify TEMPORARY, and *table_name* identifies a global temporary table, then the request is applied to the base global temporary table.

There must not be any materialized instances of the named table or you cannot drop the index on the base global temporary table.

database_name

The name of the containing database for *table_name* if different from the current database.

table_name

The table on which *index_name* is defined.

join_index_specification

The name of the join index from which this index is to be dropped.

database_name

The name of the containing database for *join_index_name* if different from the current database.

join_index_name

The join index on which *index_name* is defined.

order_clause

Index rows are ordered on each AMP by a single value- or hash-ordered column.

- If you specify ORDER BY, then you must also specify *column_name_2*.
- If you do not specify ORDER BY, Vantage orders rows by the hash value of all the columns in the *column_name* list.

VALUES

Value-ordering for the ORDER BY column.

If neither VALUES nor HASH is specified, VALUES is used as the default.

HASH

Hash-ordering for the ORDER BY column.

column_name_2

A single column from the column list defined by *column_name* that specifies the sort order to be used for either HASH or VALUES.

The size of the column is limited to four or fewer bytes. Supported numeric date types for *column_name_2* are the following.

- BYTEINT
- DATE
- DECIMAL
- INTEGER
- SMALLINT

If you specify ORDER BY without also specifying an explicit *column_name_2*, then the system assumes the first column specified in the *column_name* list.

Examples

Example: Dropping a Named Secondary Index

This request drops the existing named index index_1 from table table_5 .

```
DROP INDEX index_1 ON table_5;
```

Example: Dropping a Simple Unnamed Secondary Index

This request drops a secondary index that was defined on the name column of the employee table.

```
DROP INDEX (name) ON Employee;
```

Example: Dropping a Composite Unnamed Secondary Index

This request drops the existing index on columns column_1 and column_2 from table table_6 .

```
DROP INDEX (column_1, column_2) ON table_6;
```

Related Information

- CREATE INDEX in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- [CREATE TABLE and CREATE TABLE AS](#)
- CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - Database Design*, B035-1094

DROP JOIN INDEX

Drops the join index.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Other SQL dialects support similar non-ANSI standard statements with names such as the following.

- DROP MATERIALIZED VIEW

Required Privileges

You must have either of the following sets of privileges to drop a join index.

- DROP TABLE on the join index
- INDEX or DROP TABLE on at least one of the base tables

DROP JOIN INDEX Syntax

```
DROP JOIN INDEX [ database_name. ] join_index_name [;]
```

DROP JOIN INDEX Syntax Elements

database_name

Name of containing database for *join_index_name*.

This specification is required only if the join index to be dropped is contained in a different database than the current database.

join_index_name

Name of the join index to be dropped.

Example: Dropping a Join Index

The following request drop the definition of the join index named `ji_emp` from the data dictionary.

```
DROP JOIN INDEX ji_emp;
```

Related Information

- CREATE JOIN INDEX in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - Database Design*, B035-1094

DROP HASH INDEX

Drops a hash index on a table.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have either of the following sets of privileges to drop a hash index:

- DROP TABLE on the join index
- INDEX or DROP TABLE on at least one of the base tables

DROP HASH INDEX Syntax

```
DROP HASH INDEX [ database_name. ] hash_index_name [;]
```

DROP HASH INDEX Syntax Elements

database_name

Name of the containing database for *hash_index_name* if different from the current database.

hash_index_name

Name of the hash index to be dropped.

Example: Dropping a Hash Index

This request drops the definition for the hash index named *OrdHIdx_1* from the data dictionary.

```
DROP HASH INDEX OrdHIdx_1;
```

Related Information

- CREATE HASH INDEX in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - Database Design*, B035-1094

HELP INDEX

Displays the attributes for the primary and secondary indexes defined for a base data table, hash index, or join index.

Because hash indexes cannot have secondary indexes, a HELP INDEX request run against a hash index reports the attributes of its primary index only.

HELP INDEX does not display the attributes for hash or join indexes. See “HELP HASH INDEX” and “HELP JOIN INDEX” for information about how to report the attributes for hash and join indexes, respectively.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must either own the table or join index on which the index is defined or have at least one privilege on that table, hash index, or join index.

The SHOW privilege enables a user to perform HELP or SHOW requests only against a specified index.

HELP INDEX Syntax

```
HELP INDEX {
  [TEMPORARY] [ database_name_1. | user_name_1. ] table_name |
  [ database_name_2. | user_name_2. ] { join_index_name | hash_index_name |
  view_name }
} [ ( column_name [,...] ) ] [;]
```

HELP INDEX Syntax Elements

TEMPORARY

Requested indexes are defined on a materialized global temporary table.

This keyword is valid only for global temporary tables.

database_name_1***user_name_1***

Containing database or user for *table_name* if something other than the current database or user.

table_name

Table on which a report of all primary and secondary indexes is desired.

database_name_2***user_name_2***

Containing database or user for *join_index_name*, *hash_index_name*, or *view_name* if something other than the current database or user.

join_index_name

Join index on which all primary and secondary indexes are reported.

hash_index_name

Hash index on which a report of all primary indexes is desired.

view_name

Name of a single-table view for which index information is required.

You can only request HELP INDEX reports for single-table views.

column_name

Column in a column list for which an index or index list is desired. Use this syntax to exclude primary, secondary, join, and hash indexes defined on columns for which you are not interested.

HELP INDEX Examples

Example: HELP INDEX on Table Columns

The following example returns information about the specified index or indexes defined on the table columns named *cname_1*, *cname_2*, and *cname_3*.

```
HELP INDEX tname (cname_1, cname_2, cname_3);
```

Example: HELP INDEX on a Table

The following request and response show that one index is defined for the Department table.

```

HELP INDEX Personnel.Department;
      Primary
      or
Unique Secondary Column Names      Index Id Approximate Count
-----
Y       P       DeptNo              1              7

```

Example: Partitioned Primary Index

This example uses the following table definition.

```

CREATE TABLE orders (
  o_orderkey      INTEGER NOT NULL,
  o_custkey       INTEGER,
  o_orderstatus   CHARACTER(1) CASESPECIFIC,
  o_totalprice    DECIMAL(13,2) NOT NULL,
  o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  o_orderpriority CHARACTER(21),
  o_clerk         CHARACTER(16),
  o_shippriority  INTEGER,
  o_comment       VARCHAR(79))
PRIMARY INDEX OrdPI (o_orderkey)
PARTITION BY RANGE_N(o_orderdate BETWEEN DATE '1992-01-01'
                        AND      DATE '1998-12-31'
                        EACH INTERVAL '1' MONTH)

UNIQUE INDEX OrdUSI (o_orderkey)
INDEX OrdX1 (o_orderdate)
INDEX OrdX2 (o_orderdate)
  ORDER BY VALUES (o_orderdate)
INDEX OrdX3 (o_custkey, o_orderdate)
  ORDER BY VALUES (o_orderdate)
INDEX OrdX4 (o_custkey, o_orderdate)
  ORDER BY HASH (o_orderdate)
INDEX OrdX5 ALL (o_custkey, o_orderstatus)
  ORDER BY VALUES (o_custkey)
INDEX OrdX6 ALL (o_custkey, o_orderstatus)
  ORDER BY HASH (o_custkey)
INDEX OrdX7 ALL (o_custkey)
  ORDER BY VALUES (o_custkey)
INDEX OrdX8 ALL (o_custkey)
  ORDER BY HASH (o_custkey)
INDEX OrdX9 (o_shippriority)
  ORDER BY HASH (o_shippriority)
INDEX OrdXA ALL (o_clerk)

```

```
ORDER BY HASH (o_clerk)
INDEX OrdxB (o_orderkey, o_orderdate);
```

HELP INDEX generates the following report on this table. The BTEQ commands are included to show how the report format was derived. Attributes of particular interest for this example are highlighted in boldface.

```
BTEQ -- Enter your DBC/SQL request or BTEQ command:
.sidetitles
.sidetitles
BTEQ -- Enter your DBC/SQL request or BTEQ command:
.foldline
.foldline
BTEQ -- Enter your DBC/SQL request or BTEQ command:
HELP INDEX orders;
HELP INDEX orders;
*** Help information returned. 13 rows.
*** Total elapsed time was 1 second.
```

```
Unique? N
Primary//or//Secondary? P
    Column Names o_orderkey
    Index Id 1
    Approximate Count 0
    Index Name OrdPI
Ordered//or//Partitioned? P
    CDT Index Type N
    Unique? Y
Primary//or//Secondary? S
    Column Names o_orderkey
    Index Id 4
    Approximate Count 0
    Index Name OrdUSI
Ordered//or//Partitioned? H
    CDT Index Type N
    Unique? N
Primary//or//Secondary? S
    Column Names o_orderdate
    Index Id 8
    Approximate Count 0
    Index Name OrdX1
Ordered//or//Partitioned? H
    CDT Index Type N
    Unique? N
Primary//or//Secondary? S
    Column Names o_orderdate
```

```

        Index Id 12
        Approximate Count 0
        Index Name Ord2
Ordered//or//Partitioned? V
        CDT Index Type N
        Unique? N
        Primary//or//Secondary? S
        Column Names o_custkey, o_orderdate
        Index Id 16
        Approximate Count 0
        Index Name Ord3
Ordered//or//Partitioned? V
        CDT Index Type N
        Unique? N
        Primary//or//Secondary? S
        Column Names o_custkey, o_orderdate
        Index Id 24
        Approximate Count 0
        Index Name Ord4
Ordered//or//Partitioned? H
        CDT Index Type N
        Unique? N
        Primary//or//Secondary? S
        Column Names o_custkey, o_orderstatus
        Index Id 32
        Approximate Count 0
        Index Name Ord5
Ordered//or//Partitioned? V
        CDT Index Type N
        Unique? N
        Primary//or//Secondary? S
        Column Names o_custkey, o_orderstatus
        Index Id 40
        Approximate Count 0
        Index Name Ord6
Ordered//or//Partitioned? H
        CDT Index Type N
        Unique? N
        Primary//or//Secondary? S
        Column Names o_custkey
        Index Id 48
        Approximate Count 0
        Index Name Ord7
Ordered//or//Partitioned? V

```



```

      CDT Index Type N
      Unique? N
Primary//or//Secondary? S
      Column Names o_custkey
      Index Id 52

```

Example: HELP INDEX for a Geospatial Index

This example demonstrates a HELP INDEX report on table *gt1* that has a NUSI named *gt1_geoidx* defined on the geospatial column *geo* that has the ST_Geometry data type.

```

CREATE SET TABLE jw.gt1 (
  a  INTEGER,
  geo SYSUDTLIB.ST_Geometry,
  c  INTEGER)
PRIMARY INDEX (a)
INDEX gt1_geoidx (geo);

```

A HELP INDEX request on the NUSI named *gt1_geoidx* for this table returns the following.

```

HELP INDEX gt1_geoidx (geo);
Unique? N
Primary//or//Secondary? S
      Column Names geo
      Index Id 4
      Approximate Count 14
      Index Name gt1_idx
Ordered//or//Partitioned? H
      CDT Index Type G
      Approximate Count 0
      Index Name Ord8
Ordered//or//Partitioned? H
      CDT Index Type N
      Unique? N
Primary//or//Secondary? S
      Column Names o_shippriority
      Index Id 56
      Approximate Count 0
      Index Name Ord9
Ordered//or//Partitioned? H
      CDT Index Type N
      Unique? N
Primary//or//Secondary? S
      Column Names o_clerk

```

```

        Index Id 60
    Approximate Count 0
        Index Name OrdxA
Ordered//or//Partitioned? H
        CDT Index Type N
        Unique? N
    Primary//or//Secondary? S
        Column Names o_orderkey, o_orderdate
        Index Id 64
    Approximate Count 0
        Index Name OrdxB
Ordered//or//Partitioned? H
        CDT Index Type N

```

Note:

The CDT Index Type attribute is G to signify that the index has a Geospatial data type. If the attribute is N, the index is not a CDT Index Type.

Related Information

- [COLLECT STATISTICS \(Optimizer Form\)](#)
- [CREATE TABLE and CREATE TABLE AS](#)
- [SHOW object](#)
- *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ - Database Design*, B035-1094
- *Teradata Vantage™ - Data Types and Literals*, B035-1143

HELP JOIN INDEX

Displays the attributes of the columns defined by a particular join index.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must either own the tables on which the join index is defined or have at least one privilege on each of those tables.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against a specified join index.

HELP JOIN INDEX Syntax

```
HELP JOIN INDEX [ database_name. | user_name. ] join_index_name [;]
```

HELP JOIN INDEX Syntax Elements

database_name

user_name

Name of the containing database or user for *join_index_name* if something other than the current database or user.

join_index_name

Name of the join index for which information is desired.

Example: HELP JOIN INDEX on a Join Index

The following example shows the output of a HELP JOIN INDEX request performed on the join index named ord_cust_idx.

```
HELP JOIN INDEX ord_cust_idx;
Column Name      Type      Comment
-----
c_custkey        I         ?
c_name           CF        ?
o_status         CF        ?
o_date           DA        ?
o_comment        CV        ?
```

Related Information

- [COLLECT STATISTICS \(Optimizer Form\)](#)
- [SHOW object](#)
- *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ - Database Design*, B035-1094

HELP HASH INDEX

Displays the data types of the columns defined by a particular hash index.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must either own the table on which the hash index is defined or have at least one privilege on that table.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only for a specified hash index.

HELP HASH INDEX Syntax

```
HELP HASH INDEX [ database_name. | user_name. ] hash_index_name [;]
```

HELP HASH INDEX Syntax Elements

database_name

user_name

Containing database or user for *hash_index_name* if different from the current database or user.

hash_index_name

Name of the hash index for which information is desired.

Example: HELP HASH INDEX

This example shows the output of a HELP HASH INDEX request performed on the hash index named ord_hidx.

```
HELP HASH INDEX ord_hidx;
Column Name      Type Comment
-----
o_orderdate      DA      ?
```

Related Information

- [COLLECT STATISTICS \(Optimizer Form\)](#)
- [SHOW object](#)
- *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ - Database Design*, B035-1094

Authorization Statements for External Routines

CREATE AUTHORIZATION and REPLACE AUTHORIZATION

Creates or replaces an authorization object.

An authorization object associates a database user with an OS platform user identification, allowing an external routine to run in secure mode using the context, privileges, and access control granted to the specified OS user.

External routines include UDFs, table UDFs, methods, and external SQL procedures. See [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#), [CREATE FUNCTION and REPLACE FUNCTION \(Table Form\)](#), [CREATE METHOD](#), and [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#).

You specify an authorization in the external security clause of a foreign table or function mapping. See [CREATE FOREIGN TABLE](#) or [CREATE FUNCTION MAPPING and REPLACE FUNCTION MAPPING](#).

You can also specify an authorization in the AUTHORIZATION clause of the READ_NOS table operator. See *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

Note:

If an AWS IAM credential provides access, you can omit the AUTHORIZATION clause.

For information about the CREATE AUTHORIZATION trusted extensions that you use to access remote systems with QueryGrid, see your Teradata QueryGrid documentation.

You can specify DEFINER or INVOKER, but not both.

ANSI Compliance

CREATE AUTHORIZATION is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have either the CREATE/REPLACE AUTHORIZATION or AUTHORIZATION privilege to perform the CREATE/REPLACE AUTHORIZATION statement.

Privileges Granted Automatically

DROP AUTHORIZATION to the creator of the authorization.

CREATE AUTHORIZATION and REPLACE AUTHORIZATION Syntax

```
{ CREATE | REPLACE } AUTHORIZATION [ database_name. |
user_name. ] authorization_name
  [ AS [ DEFINER | INVOKER ] TRUSTED ]
  USER 'user_name'
  PASSWORD 'password'
  [ ; ]
```

CREATE AUTHORIZATION and REPLACE AUTHORIZATION Syntax Elements

database_name

user_name

Optional name of a database or user other than the current or default in which the authorization being defined or replaced is to be contained.

authorization_name

A name for the authorization so you can specify the authorization in an external routine definition or function mapping.

The name of an authorization object must conform to object naming rules. For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

The following rules apply to authorization names:

- Authorizations belong to the database or user in which they are created and are not valid in other databases or users. For information about using authorization objects with the Script Table Operator, see *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.
- An authorization object name must be unique within its containing database or user.
- An authorization object name cannot begin with a digit.
- An authorization object name cannot be an SQL keyword.

DEFINER

Specify DEFINER to share an authorization object with multiple users of the database in which it resides. You can create the authorization in any database.

DEFAULT

An optional keyword modifier for the DEFINER keyword that associates this authorization with all external routines that do not specify the authorization name in the EXTERNAL SECURITY DEFINER clause of the following statements.

- [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#).
- [CREATE FUNCTION and REPLACE FUNCTION \(Table Form\)](#).
- [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#).

You can assign only one default DEFINER object per database. All others must have specific definer names.

INVOKER

Specify INVOKER to allow exclusive access by a user. You must create the authorization in the database of the current user.

TRUSTED

Required keyword.

user_name

A clause that specifies a string literal that is the name of the database user to whom this authorization is being assigned.

Public buckets (or *public containers*) in external object stores such as Amazon S3, Azure Blob storage, or Google Cloud Storage, do not require credentials for access. To create an authorization for these, use an empty string delimited by single quotes: ''

password

A clause that specifies a string literal that is the name of the operating system platform password assigned to *user_name*.

For AWS, AZURE, and GCP, *password* can have at most 4096 bytes.

The password is used to authenticate the user when the secure server process is created. Best practices suggest that any session that uses this statement should be set up to use the encrypted transport protocol. See *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

Public buckets (or *public containers*) in external object stores such as Amazon S3, Azure Blob storage, or Google Cloud Storage, do not require credentials for access. To create an authorization for these, use an empty string delimited by single quotes: ''

While it is never desirable for a password to be typed in the clear, it only becomes an issue when you enter your password using an interface such as BTEQ. The issue becomes moot if you are prompted to enter your password by an application that requests the information in a secure manner such as a GUI or World Wide Web interface that displays ASTERISK

characters to represent the password as it is typed. This is the recommended practice for all security conscious sites.

The following table shows the supported credentials for USER and PASSWORD:

System/Scheme	USER	PASSWORD
AWS	Access Key ID	Access Key Secret
Azure / Shared Key	Storage Account Name	Storage Account Key
Azure Shared Access Signature (SAS)	Storage Account Name	Account SAS Token
Google Cloud (S3 interop mode)	Access Key ID	Access Key Secret
Google Cloud (native)	Client Email	Private Key
On-premises object stores	Access Key ID	Access Key Secret
Public access object stores	<empty string> Enclose the empty string in single straight quotes: USER ''	<empty string> Enclose the empty string in single straight quotes: PASSWORD ''

Note:

Amazon Identity and Access Management (IAM) is an alternative to using an access key and password to secure S3 buckets.

Examples

Example: Creating a DEFINER Authorization

The following example creates a DEFINER authorization object with the name *sales_processing* for the user name *salesmng* and the user password *mysecret*.

```
CREATE AUTHORIZATION sales_processing
AS DEFINER TRUSTED
USER 'salesmng'
PASSWORD 'mysecret';
```

The database uses this OS platform user context to execute all external routines that contain an EXTERNAL SECURITY DEFINER clause when run under the same OS user authorization independent of the user who runs the routine.

Example: Creating a DEFINER Authorization for an External Security Clause

This example creates a specific definer authorization name *sales* with all external routines that contain the external security clause `EXTERNAL SECURITY DEFINER sales`. The authorization object name *sales* is part of the text specified for the external security clause. This specification causes the system to validate the authorization object `SYSLIB.sales` in the data dictionary and then, once it has been validated, to use the specified user context to run the external routine.

```
CREATE AUTHORIZATION SYSLIB.sales
AS DEFINER TRUSTED
USER 'SalesDept'
PASSWORD 'ikcerednep';
```

Example: Creating a Default DEFINER Authorization

This example creates a definer authorization with the name *accounting* in the SYSLIB database for the client user name *accdept* and user password *nesuahkcotsk*.

```
CREATE AUTHORIZATION SYSLIB.accounting
AS DEFINER DEFAULT TRUSTED
USER 'accdept'
PASSWORD 'nesuahkcotsk';
```

The example associates *accounting* as the DEFAULT authorization in the SYSLIB database with the specified client user name *accdept*.

Because *accdept* is the DEFAULT authorization object, the system associates it with all external routines that contain the `EXTERNAL SECURITY DEFINER` clause. All external routines in database SYSLIB that contain an `EXTERNAL SECURITY DEFINER` clause are executed using the user context defined by this request.

Recall that there can only be one default definer object in a database. All others must have specific names. If the DEFAULT already exists, the system returns an error to the requestor.

Example: Creating an INVOKER Authorization

This example creates an INVOKER authorization object with the name *sales* for the user name *sam johnson* and the user password *tercesym*.

```
CREATE AUTHORIZATION sales
AS INVOKER TRUSTED
```

```
USER 'sam johnson'
PASSWORD 'tercesym';
```

The system uses this OS platform user context to execute all external routines that contain an `EXTERNAL SECURITY INVOKER` clause when invoked by the database user that created this authorization.

Example: Creating a Google Cloud Authorization

This example creates an authorization object for accessing Teradata Vantage on Google Cloud.

```
CREATE AUTHORIZATION Auth_S3_USR_IVO
USER 'service-account-name@project-id.iam.gserviceaccount.com'
PASSWORD '-----BEGIN PRIVATE KEY-----\n
MIIIEvQIBADANBgkqhkiG9w0BAQEFAASCBCwggSjAgEAAoIBAQCyfg3398i0xjt...a1mSAvk9SqPoyZ
R7JJFs=\n
-----END PRIVATE KEY-----\n';
```

Depending on the user credentials, Advanced SQL Engine uses either the S3-compatible connector or the native Google connector.

Example: Creating an AUTHORIZATION OBJECT for use with NOS

Run the following command to create an authorization object to access external object store:

```
CREATE AUTHORIZATION MyAuthObj
USER 'user_name'
PASSWORD 'password';
```

Where:

- *user_name* is the `ACCESS_KEY` or Storage Account Name to your external object store.
- *password* is the secret access key, Key, SAS Token, or Secure Email and Secret Access Key depending on your external object store.

If you are accessing a public bucket or container, put empty strings for `USER` and `PASSWORD`, enclosed in straight quotes, such as:

```
CREATE AUTHORIZATION MyAuthObj
USER ''
PASSWORD '';
```

Associate the **AUTHORIZATION OBJECT** with a Foreign Table

```
CREATE FOREIGN TABLE riverflow_csv
, EXTERNAL SECURITY MyAuthObj
USING ( LOCATION('/s3/td-usgs-public.s3.amazonaws.com/CSVDATA/') );
```

Related Information

See CREATE AUTHORIZATION in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for more information about creating authorization objects.

See [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#), [CREATE FUNCTION and REPLACE FUNCTION \(Table Form\)](#), and [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#) for information about how authorization objects are used with external UDFs and external SQL procedures.

See the cufconfig utility in *Teradata Vantage™ - Database Utilities*, B035-1102 for information about how to configure external routine server processes and how to alter secure group membership.

DROP AUTHORIZATION

Drops an authorization object.

If you drop an authorization object, any external routine defined using the name of that authorization object is not accessible. To correct the issue, you must have the CREATE AUTHORIZATION privilege.

When you drop an INVOKER authorization by its authorization name, the system also drops the INVOKER_DEFAULT authorization entry.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the DROP AUTHORIZATION privilege on the authorization object.

DROP AUTHORIZATION Syntax

```
DROP AUTHORIZATION [ database_name. ] authorization_name [;]
```

DROP AUTHORIZATION Syntax Elements

database_name

Containing database for *authorization_name*.

authorization_name

Existing authorization object to be dropped.

Example: Dropping an Authorization Object

This example drops the authorization object named sales from the dictionary.

```
DROP AUTHORIZATION sales;
```

Related Information

See [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#) for information about creating or replacing the definition of an authorization object.

See [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#) and [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#) for information about how authorization objects are used with external UDFs and external SQL procedures.

Global and Persistent (GLOP) Data Statements

CREATE GLOP SET

Creates the definition of a global persistent memory set.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the CREATE GLOP privilege on the containing database or user.

Privileges Granted Automatically

The creator of a GLOP set is automatically granted the following privileges on the created GLOP set.

- GLOP MEMBER
- DROP GLOP

CREATE GLOP SET Syntax

```
CREATE GLOP SET [ database_name. | user_name. ] GLOP_set_name [;]
```

CREATE GLOP SET Syntax Elements

database_name

user_name

Name of the containing database or user for *GLOP_set_name* if different from the current database or user.

GLOP_set_name

Name of the GLOP set being created.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Examples

Example: Creating a GLOP Set

This example creates a GLOP set named UDF_glop_1 in *user_name*.

```
CREATE GLOP SET  user_name.UDF_glop_1;
```

Related Information

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about how to use GLOPs with external routines.

DROP GLOP SET

Drops the definition of the specified GLOP set from the Data Dictionary.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the DROP GLOP privilege on the database or user containing *GLOP_set_name* to drop the specified GLOP set.

Privileges Granted Automatically

None.

DROP GLOP SET Syntax

```
DROP GLOP SET [ database_name. | user_name. ] GLOP_set_name [;]
```

DROP GLOP SET Syntax Elements

database_name

user_name

Name of the containing database or user for *GLOP_set_name* if different from the current database or user.

GLOP_set_name

Name of the GLOP set to be dropped.

Example: Dropping a GLOP Set

This example drops the GLOP set named *UDF_glop_1* contained by *user_name*.

```
DROP GLOP SET  user_name.UDF_glop_1;
```

Related Information

- [CREATE GLOP SET](#)
- *Teradata Vantage™ - SQL External Routine Programming*, B035-1147

Procedure Statements

CREATE PROCEDURE and REPLACE PROCEDURE (External Form)

Compiles and installs an external SQL procedure routine and creates or replaces the SQL definition used to invoke the procedure.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

Required Privileges

The privileges required to perform CREATE PROCEDURE (External Form) and REPLACE PROCEDURE differ:

- You must have explicit CREATE EXTERNAL PROCEDURE privileges on the database in which the external procedure is to be contained to perform the CREATE PROCEDURE (External Form) request.

The system does not grant the CREATE EXTERNAL PROCEDURE privilege automatically when you create a database or user: you must grant that privilege explicitly.

CREATE EXTERNAL PROCEDURE is not granted implicitly on the databases and external procedures owned by a database or user unless the owner also has an explicit WITH GRANT OPTION privilege defined for itself.

See [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#) for further information.

- You must have explicit DROP PROCEDURE privileges on the external procedure or on the database in which the external procedure is contained to perform the REPLACE PROCEDURE statement on an existing external procedure. You do not need the CREATE EXTERNAL PROCEDURE privilege to replace an external procedure.
- You must have explicit CREATE EXTERNAL PROCEDURE privileges on the database in which the external procedure is to be contained to perform the REPLACE PROCEDURE statement to create a new external procedure.

If the parameter type is a UDT or an Array/Varray with a UDT element type, you must have at least one of the following privileges:

- UDTUSAGE on the specified UDT
- UDTUSAGE on the SYSUDTLIB database
- UDTTYPE on the SYSUDTLIB database

See [CREATE PROCEDURE and REPLACE PROCEDURE \(SQL Form\)](#) for information about how to use the SQL SECURITY clause.

Privileges Granted Automatically

When you create a new procedure, Vantage automatically grants the following privileges on it.

- DROP PROCEDURE
- EXECUTE PROCEDURE

CREATE PROCEDURE and REPLACE PROCEDURE Syntax (External Form)

```
{ CREATE | REPLACE } PROCEDURE [ database_name. | user_name. ] procedure_name
  ( parameter_specification [ , ... ] )
  [ DYNAMIC RESULT SETS number_of_sets ]
  language_and_access_specification
  parameter_style_specification
  [ USING GLOP SET GLOP_set_name ]
  [ SQL SECURITY privilege_option ]
  EXTERNAL [ NAME { external_procedure_name | 'code_specification [delimiter...]'
  | 'JAR_ID_specification' } ]
  [ parameter_style_specification ]
  [ EXTERNAL SECURITY { DEFINER [ authorization_name ] | INVOKER } ] [ ; ]
```

Note:

You can specify *language_and_access_specification* and *parameter_style_specification* in the reverse order.

parameter_specification

```
[ IN | OUT | INOUT ] parameter_name data_type
```

language_and_access_specification

```
{ language_clause [ SQL_data_access ] |
  SQL_data_access [ language_clause ] |
  external_data_access
}
```

parameter_style_specification

```
PARAMETER STYLE { SQL | TD_GENERAL | JAVA }
```

code_specification

```
{ F delimiter function_entry_name |
  D |
  { S | C } path_specification
}
```

JAR_ID_specification

```
JAR_ID:java_class_name.java_method_name
  [ ( java_parameter_class [,...] ) returns java_parameter_class ]
```

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [( fractional_seconds_precision)] [WITH TIME
ZONE] |

  INTERVAL YEAR [( precision)] [TO MONTH] |

  INTERVAL MONTH [( precision)] |

  INTERVAL DAY [( precision)]
    [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL HOUR [(precision)]
    [TO { MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL MINUTE [(precision)]
    [ TO SECOND [(fractional_seconds_precision)] ] |

  INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) |

  PERIOD (DATE) |

  PERIOD ({ TIME | TIMESTAMP } [(precision)] [ WITH TIME ZONE ]) |

  REAL |

  DOUBLE PRECISION |
```

```

FLOAT [(integer)] |

NUMBER [( { integer | *} [, integer ]... ) ] |

{ DECIMAL | NUMERIC } [(integer [, integer ]... ) ] |

{ CHAR | BYTE | GRAPHIC } [(integer)] |

{ VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [(integer)] |

LONG VARCHAR |

LONG VARGRAPHIC |

{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
(integer [ G | K | M ]) |

[SYSUDTLIB.] { XML | XMLTYPE } [(integer [ G | K | M ])] [ INLINE
LENGTH integer ] |

[SYSUDTLIB.] JSON [(integer [ K | M ])] [ INLINE LENGTH integer ]
[ CHARACTER SET { UNICODE | LATIN } ] |

[SYSUDTLIB.] ST_GEOMETRY [(integer [ K | M ])] [ INLINE LENGTH
integer ] |

[SYSUDTLIB.] DATASET [(integer [ K | M ])]
[ INLINE LENGTH integer ] storage_format |

[SYSUDTLIB.] { UDT_name | MBR | ARRAY_name | VARRAY_name }
}

```

path_specification

```

{ I delimiter name_on_server delimiter include_name |
  L delimiter library_name |
  O delimiter name_on_server delimiter object_name |
  P delimiter package_name |
  S delimiter name_on_server delimiter source_name |
  NS delimiter source_file delimiter include_file
}

```

java_parameter_class

```
{ primitive [ ] [ [ ] ] ] | object [ [ ] ] }
```

Note:

You must type the colored or bold braces.

storage_format

```
STORAGE FORMAT { Avro | CSV [ CHARACTER SET { UNICODE | LATIN } ] }  
[ WITH SCHEMA [ database. ] schema_name ]
```

CREATE PROCEDURE and REPLACE PROCEDURE Syntax Elements (External Form)

database_name

An optional database name specified if the procedure is to be created or replaced in a non-default database.

If you do not specify a database name, Vantage creates or replaces the procedure within the current database.

user_name

An optional user name specified if the procedure is to be created or replaced in a non-default user.

If you do not specify a user name, Vantage creates or replaces the procedure within the current user.

procedure_name

The calling name for the external procedure.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, method, or Java stored procedure object even if the new object name contains only single-byte characters. If you attempt to do so, Vantage aborts the request and returns an error to the requestor. Instead, use a multibyte session character set.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

A procedure name is mandatory for all external procedures.

procedure_name must match the spelling and case of the C, C++, or Java procedure name exactly if you do not specify an *external_procedure_name*. This applies only to the definition of the procedure, not to its use.

number_of_sets

Specifies that *number_of_sets* of dynamic result sets can be returned.

The range of valid values for *number_of_sets* is 0 through 15, inclusive.

external_procedure_name

The entry point for the procedure object.

The name for an external procedure must conform to object naming rules. See *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Case is significant and must match the C or C++ procedure name.

Note:

This option is not valid for Java external procedures. Instead, you must specify the EXTERNAL NAME *external_Java_reference_string* option.

EXTERNAL SECURITY

This clause is recommended for procedures that perform operating system I/O operations because it permits you to specify a particular OS user under whom the function runs. Otherwise, a protected mode procedure runs under the generic user *tdatuser*.

See also [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#).

DEFINER

The UDF runs in the client user context of the associated security authorization object created for this purpose, which is contained within the same database as the procedure.

- If you specify an authorization name, you must define an authorization object with that name before you can invoke the procedure.
- If you do not specify an authorization name, you must define a default DEFINER authorization object.

The default authorization object must be defined before a user can run the procedure.

Vantage reports a warning if the specified authorization name does not exist at the time the procedure is created, stating that no authorization name exists.

If you then attempt to execute the procedure, the request aborts and the system returns an error to the requestor.

authorization_name

An optional authorization name for this DEFINER as defined by CREATE AUTHORIZATION.

INVOKER

The procedure runs using the INVOKER authorization associated with the logged on user who is running the function.

parameter_specification

A parenthetical comma-separated list of data types, including UDTs, and parameter names for the variables to be passed to the procedure.

You must specify opening and closing parentheses even if no parameters are to be passed to the procedure.

The maximum number of parameters you can specify in the parameter list depends on the language in which the external routine for the procedure is written:

Language	Maximum Number of Parameters
C or C++	256
Java	255

IN

Parameter is input only. IN is the default parameter type. If the parameter type is not specified, the parameter is assumed to be of the IN type.

OUT

Parameter is output only.

INOUT

Parameter can be input and output.

Note:

Vantage does not default to the data type you assign to an INOUT parameter when the procedure is called. Instead, it defaults to the smallest data type that can contain the specified input parameter. As a result, memory overflow errors can occur if an output parameter returned to an INOUT parameter cannot be contained by the default data type set for that parameter by the system.

parameter_name

Name of parameter name to pass to the procedure.

data_type

The data type associated with each parameter is the type of the parameter. For C and C++ procedures, all Vantage data types are valid. For Java procedures, all Vantage data types are valid except GRAPHIC and VARGRAPHIC.

For data types that take a length or size specification, like BYTE, CHARACTER, DECIMAL, VARCHAR, and so on, the length of the parameter indicates the longest string that can be passed.

Character data can also specify a CHARACTER SET clause.

Note:

You cannot specify a character server data set of KANJI1. Otherwise, the system returns an error to the requestor.

BLOB and CLOB types must be represented by a locator. For a description of locators, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146. The system does not support in-memory LOB parameters. An AS LOCATOR phrase must be specified for each LOB parameter and return value in a non-Java procedure.

For Java procedures, the simple data type references BLOB and CLOB implicitly refer to BLOB AS LOCATOR and CLOB AS LOCATOR, respectively. Because of this, you cannot specify an explicit AS LOCATOR for BLOB and CLOB columns in Java procedures.

Note:

When a LOB that requires data type conversion is passed to an external procedure, the LOB must be materialized for the conversion to take place.

VARIANT_TYPE

You can only specify the `VARIANT_TYPE` UDT type for callable input parameters within the body of the procedure using a `NEW VARIANT_TYPE` expression to pass dynamic UDTs into UDFs. You cannot declare a `VARIANT_TYPE` as an IN parameter data type for a procedure.

TD_ANYTYPE

You can specify the system-defined `TD_ANYTYPE` data type for an IN, OUT, or INOUT parameter.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for a list of the data type encodings that SQL procedure IN, INOUT, and OUT parameters can return to a client application.

language_clause

A code that represents the programming language in which the external procedure is written.

This clause is mandatory for all external procedures. The valid languages for writing external SQL procedures are C, C++, and Java.

You must specify one of these codes even if the external procedure is supplied in object form. If the external procedure object is not written in C, C++, or Java it must be compatible with C, C++, or Java object code.

LANGUAGE

Keyword to introduce the programming language.

C

The external procedure is written in C.

CPP

The external procedure is written in C++.

JAVA

The external procedure is written in Java.

SQL_data_access

Specifies whether the external procedure body accesses the database or contains SQL statements.

This clause is mandatory for all external procedures.

Note:

SQL DCL and DDL statements for administering row-level security are not allowed in a stored procedure.

Any of the listed option strings is valid for external routines written in the C, C++, or Java languages.

See *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for more information about the SQL data access options.

CONTAINS SQL

The external procedure body contains SQL statements.

MODIFIES SQL DATA

The external procedure body accesses the database.

NO SQL

The default is NO SQL.

READS SQL DATA

The external procedure body accesses the database.

external_data_access

An optional clause that determines whether the function or external stored procedure can read or modify data that is external to Vantage.

MODIFIES EXTERNAL DATA

The function modifies data that is external to Vantage.

Note:

In the event of a database error, the system does not redrive requests involving a function or external stored procedure defined with this option. For details about Redrive functionality, see *Teradata Vantage™ - Database Administration*, B035-1093.

NO EXTERNAL DATA

The function does not access data that is external to Vantage. This is the default.

READS EXTERNAL DATA

The function reads, but does not modify data that is external to Vantage.

parameter_style_specification

The parameter passing convention to be used when passing parameters to the procedure.

This clause is mandatory for all external procedures. The specified parameter style must match the parameter passing convention of the external procedure.

If you do not specify a parameter style at this point, you can specify one with the external body reference.

You cannot specify parameter styles more than once in the same CREATE PROCEDURE or REPLACE PROCEDURE request.

SQL

Uses indicator variables to pass arguments.

As a result, you can always pass nulls as inputs and return nulls in results.

SQL is the default parameter style.

TD_GENERAL

Uses parameters to pass arguments.

Can neither be passed nor return nulls.

JAVA

Mandatory for all Java procedures.

If the Java procedure must accept null arguments, then the EXTERNAL NAME clause must include the list of parameters and specify data types that map to Java objects.

GLOP_set_name

The name of the GLOP set to associate with this procedure.

The name for a GLOP set object must conform to object naming rules. See *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

The specified GLOP set does not have to exist at the time the procedure is created.

You can specify this clause anywhere between the parameter list and the EXTERNAL clause.

privilege_option

The SQL SECURITY clause is supported only for those external procedures where both of the following statements are true:

- The procedure has one of the following AppCategory codes in *DBC.UDFInfo* :

- C (CLv2)
- J (Java)
- N (.NET)
- O (ODBC)

The only non-valid AppCategory code is S.

See *Teradata Vantage™ - Data Dictionary*, B035-1092 for details.

The SQL ACCESS clause in the procedure definition specifies one of the following options that supports SQL.

- CONTAINS SQL
- MODIFIES SQL DATA
- READS SQL DATA

The SQL ACCESS option cannot be NO SQL. If you specify NO SQL, Vantage aborts the request and returns an error to the requestor.

Note:

SQL DCL and DDL statements for administering row-level security are not allowed in a stored procedure.

The following bullets list the valid privilege options.

- CREATOR
- DEFINER
- INVOKER
- OWNER

See CREATE PROCEDURE (External Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for the definitions of the privilege options.

Vantage treats procedures as follows when the procedure is defined as a DEFINER whether explicitly or by default. This is for the case when the CREATOR of the procedure is not its OWNER.

- Vantage checks the privileges of the CREATOR during compilation of the procedure. The CREATOR must have the appropriate privilege for any object that a statement in the procedure references.
- Vantage checks the privileges of the OWNER (the containing database or user for the procedure) during compilation, and if the OWNER is missing any privileges for any object that is referenced by a statement in the procedure, Vantage returns a warning message.
- During execution, Vantage checks the OWNER privileges for any object referenced by any statement in the procedure. Any access failures caused by not having the appropriate privileges return an error to the procedure, which it can handle if it is written to do so.
- Vantage grants the DROP and EXECUTE privileges to a UDF created in a database or user different from the CREATOR. An OWNER always has the implicit privilege to drop any object it owns. If the OWNER wants to execute the UDF, then it must grant the EXECUTE FUNCTION privilege to that function.

The following rules apply to all procedures:

- Objects referenced by any statement in the procedure need not have the WITH GRANT OPTION privilege. They require only the GRANT privilege on the referenced object.
- Vantage check CREATOR and OWNER privileges on DDL statements during execution.

The following paragraphs summarize how Vantage treats privilege violations for procedures during procedure compilation.

For DCL and DDL requests, Vantage checks the CREATOR and OWNER privileges during compilation.

The system returns warning messages to the procedure for any privilege violations that occur.

For DML requests, Vantage checks the CREATOR and OWNER privileges during compilation:

- For the CREATOR, the system returns error messages to the procedure for any privilege violations that occur.

- For the OWNER, the system returns warning messages to the procedure for any privilege violations that occur.

For dynamic SQL requests, Vantage does not check any privileges.

The following bullets summarize how Vantage treats privilege violations for procedures during procedure execution.

- For DCL and DDL requests, Vantage checks the CREATOR and OWNER privileges during execution.

The system returns error messages to the procedure for any privilege violations that occur.

- For DML requests, Vantage checks the OWNER privileges during execution.

The system returns error messages to the procedure for any privilege violations that occur.

- For dynamic SQL requests, Vantage checks the CREATOR and OWNER privileges during execution.

code_specification

F

Function object. The string that follows is the entry point name of the C or C++ procedure object.

```
F!function_entry_point_name
```

delimiter

Specify a delimiter character, such as !. You must use the same delimiter throughout the string specification.

function_entry_name

Name the file on the server. Include files must have the same name specified in the include statement in the C source, without the extension.

D

Enables symbolic debugging for the external procedure, which shows source code and displays variables by name. Without this option, external procedures can only be debugged at the machine instruction level. You should always specify this option for debugging purposes when external procedures are being tested. This option adds -g to the C compiler command line. See [SET SESSION DEBUG FUNCTION](#) and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

The D option only applies only to C and C++ code, not to Java UDFs.

Note:

You should not use this option when installing debugged external procedures on production system because it increases the size of the external procedure library.

S

The source or object code for the external procedure is stored on the server.

C

The source or object code for the external procedure is stored on the client.

path_specification

Location (path) and name of the source, include file, object, or library. Because packages and libraries must be preinstalled, you must use the server option (S). Path specifications can use forward slashes (/) or backward slashes (\) regardless of whether the function is being created on a Unix or Windows platform.

I

Include file (.h).

```
I!name_on_server!include_name
```

L

Library name for a nonstandard library files needed by the UDF.

```
L!library_name
```

O

Object file.

```
O!name_on_server!object_name
```

P

Package name. You cannot use the package option with any other options except F, the C function name option.

```
P!package_name
```

S

Source file.

```
S!name_on_server!source_name
```

NS

No source file. Source files and include files are not stored in the function table. This option only affects how source code is processed in the creation of a new function and applies to all source code specified in the external string literal.

```
NS!source_file!include_file
```

*JAR_ID_specification**java_class_name*

The name of the Java class contained within the JAR identified by *jar_name* that contains the Java method to execute for this procedure.

If the Java class and method are contained in an external package, you must fully qualify *java_class_name* with the appropriate package name and path.

java_method_name

The name of the Java method that is performed when this procedure executes.

Examples**Example: Creating a Procedure Using PARAMETER STYLE TD_GENERAL**

The following CREATE PROCEDURE request installs the external SQL procedure named *GetRegionXSP* on the Teradata platform:

```
CREATE PROCEDURE GetRegionXSP
  (INOUT region VARCHAR(64))
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!getregion!xspsrc/getregion.c!F!xsp_getregion'
PARAMETER STYLE TD_GENERAL;
```

The only difference between this definition and the definition provided in [Example: Creating a Procedure Using PARAMETER STYLE SQL](#) is the PARAMETER STYLE declaration: this procedure uses parameter style TD_GENERAL.

The following excerpt shows the fragment of the C procedure code that declares a parameter style of TD_GENERAL:

```

/***** C source file name: getregion.c *****/

#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"
#include <string.h>

void xsp_getregion( VARCHAR_LATIN *region,
                  char          sqlstate[6])
{
    ...
}

```

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details about coding external SQL procedures.

Example: Creating a Procedure Using PARAMETER STYLE SQL

The following CREATE PROCEDURE request installs the external SQL procedure named *GetRegionXSP* on the Teradata platform:

```

CREATE PROCEDURE GetRegionXSP
  (INOUT region VARCHAR(64))
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!getregion!xspsrc/getregion.c!F!xsp_getregion'
PARAMETER STYLE SQL;

```

The only difference between this definition and the definition provided in [Example: Creating a Procedure Using PARAMETER STYLE TD_GENERAL](#) is the PARAMETER STYLE declaration: this procedure uses parameter style SQL.

The following excerpt shows the fragment of the C procedure code that declares a parameter style of SQL:

```

/***** C source file name: getregion.c *****/

#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"
#include <string.h>

void xsp_getregion( VARCHAR_LATIN *region,

```



```

        int          *region_isnull,
        char          sqlstate[6],
        SQL_TEXT      extname[129],
        SQL_TEXT      specific_name[129],
        SQL_TEXT      error_message[257] )

{
    ...
}

```

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for details about coding external SQL procedures.

Example: Creating an External Procedure that Can Modify SQL Data Using CLlv2

The following C language external procedure definition creates a procedure named `xsp_abc` that can modify SQL data using the CLlv2 API:

```

CREATE PROCEDURE xsp_abc(IN param1 INTEGER,
                        OUT result1 INTEGER)

LANGUAGE C
MODIFIES SQL DATA
PARAMETER STYLE SQL
EXTERNAL NAME 'SP!CLI!CS!xsp_abc!xsp_abc.c';

```

Example: Creating a Java External Stored Procedure Using a Distinct UDT

This statement creates a distinct UDT named `MONEY`:

```
CREATE TYPE MONEY AS numeric(10,2) FINAL;
```

This statement creates a Java external stored procedure using the `MONEY` distinct UDT as the data type for the input parameter `A1` and the output parameter `A2`:

```

CREATE PROCEDURE MyMoney(IN A1 MONEY, OUT A2 MONEY)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME
'UDF_JAR:UserDefinedFunctions.mymoney';
public static void mymoney(java.lang.BigDecimal A1,
                           java.lang.BigDecimal[] A2) throws SQLException

```

Example: Creating a Java External Stored Procedure Using a Structured UDT

This statement creates a structured UDT named CIRCLE:

```
CREATE TYPE CIRCLE AS (x double, y double, r double);
```

This statement creates a Java external stored procedure using the CIRCLE structured UDT as the data type for the input parameter A1:

```
CREATE PROCEDURE MyCircle(IN A1 CIRCLE, OUT A2 INTEGER)
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
  EXTERNAL NAME
    'UDF_JAR:UserDefinedFunctions.mycircle';
public static void mycircle(java.sql.Struct A1, int[] A2) throws SQLException
```

Example: Creating a Java External Stored Procedure Using a Period Data Type

This statement creates a Java external stored procedure using the PERIOD data type for:

- Input parameters P1 and P2
- Output parameter P3

```
REPLACE PROCEDURE PDT_UDF(IN P1 PERIOD(DATE), IN P2 PERIOD(DATE), OUT P3
PERIOD(DATE))
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
  EXTERNAL NAME
    'UDF_JAR:UserDefinedFunctions.pdt_udf';

public static void pdt_udf(java.sql.Struct p1, java.sql.Struct p2,
java.sql.Struct[] p3) throws SQLException
```

Example: Creating a Java External Stored Procedure Using a Data Type of ST_Geometry

This statement creates a Java external stored procedure using the ST_Geometry data type for the input parameter G1:

```
CREATE PROCEDURE get_Geom(IN G1 ST_Geometry, OUT A1 INTEGER)
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
  EXTERNAL NAME
  'UDF_JAR:UserDefinedFunctions.get_Geom';

public static void get_Geom(java.sql.Clob G1, int[] A1) throws SQLException
```

Example: Creating a Java External Stored Procedure Using an ARRAY or VARRAY Data Type

This statement creates a UDT named phonenumbers_ary:

```
CREATE TYPE phonenumbers_ary AS CHAR(10) ARRAY[5];
```

This statement creates a Java external stored procedure using the phonenumbers_ary UDT as the data type for the input parameter A1:

```
REPLACE PROCEDURE getPhoneNum(IN A1 phonenumbers_ary,
                               IN EMPID INTEGER, OUT myphonenum INTEGER)
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
  EXTERNAL NAME
  'UDF_JAR:UserDefinedFunctions.getPhoneNums';

public static void getPhoneNums(java.sql.Array A1, int empid, int[] myphonenum)
```

Example: Creating a Java External Stored Procedure Using a DATASET Data Type

This statement creates a Java external stored procedure using the DATASET data type for the input parameter dataset_in and output parameter dataset_out:

```
CREATE PROCEDURE dst_xsp(IN dataset_in DATASET(8000) Storage Format Avro,
                        OUT dataset_out DATASET(8000) Storage Format Avro)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'JarXSP1:dst_xsp1.dst_xsp(java.sql.Blob, byte[][]);

public static void dst_xsp(java.sql.Blob b,
                          byte[][]vb_out) throws SQLException
```

Example: Creating a Procedure Using PARAMETER STYLE SQL

The following CREATE PROCEDURE request installs the external SQL procedure named *GetRegionXSP* on the Teradata platform:

```
CREATE PROCEDURE GetRegionXSP
  (INOUT region VARCHAR(64))
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!getregion!xspsrc/getregion.c!F!xsp_getregion'
PARAMETER STYLE SQL;
```

The only difference between this definition and the definition provided in [Example: Creating a Procedure Using PARAMETER STYLE TD_GENERAL](#) is the PARAMETER STYLE declaration: this procedure uses parameter style SQL.

The following excerpt shows the fragment of the C procedure code that declares a parameter style of SQL:

```
/***** C source file name: getregion.c *****/

#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"
#include <string.h>

void xsp_getregion( VARCHAR_LATIN *region,
                   int           *region_isnull,
                   char          sqlstate[6],
                   SQL_TEXT      extname[129],
                   SQL_TEXT      specific_name[129],
                   SQL_TEXT      error_message[257] )
{
    ...
}
```

Example: Creating a Procedure Using PARAMETER STYLE TD_GENERAL

The following CREATE PROCEDURE request installs the external SQL procedure named *GetRegionXSP* on the Teradata platform:

```
CREATE PROCEDURE GetRegionXSP
  (INOUT region VARCHAR(64))
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!getregion!xspsrc/getregion.c!F!xsp_getregion'
PARAMETER STYLE TD_GENERAL;
```

The only difference between this definition and the definition provided in [Example: Creating a Procedure Using PARAMETER STYLE SQL](#) is the PARAMETER STYLE declaration: this procedure uses parameter style TD_GENERAL.

The following excerpt shows the fragment of the C procedure code that declares a parameter style of TD_GENERAL:

```
/****** C source file name: getregion.c *****/

#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"
#include <string.h>

void xsp_getregion( VARCHAR_LATIN *region,
                  char          sqlstate[6])
{
    ...
}
```

Example: Creating Java External SQL Procedures

The following set of examples shows CREATE PROCEDURE requests and relevant Java external procedure code for a variety of different parameter data types.

The following example is for a BYTEINT parameter data type.

```
CREATE PROCEDURE mybyteint
  (IN  b BYTEINT
   OUT c BYTEINT)
LANGUAGE JAVA
```

```

NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'User_jar:UnitTest.mybyteint';
public static void mybyteint(
    byte a,
    byte[] Result) throws SQLException

```

The following example is for a SMALLINT parameter data type.

```

CREATE PROCEDURE mysmallint
    (IN    b smallint,
     INOUT c smallint )
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'newint_jar:newint.mysmallint';
public static void mysmallint(
    short a,
    short[] Result ) throws SQLException

```

The following example is for an INTEGER parameter data type.

```

CREATE PROCEDURE myint
    (IN  b INTEGER,
     OUT c INTEGER )
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'user_jar:UnitTest.myint';
public static void myint(int i,
    int[] retval) throws SQLException

```

The following example is for a DECIMAL or NUMERIC parameter data type.

```

CREATE PROCEDURE mydecs
    (IN  c DECIMAL(8,2),
     OUT d DECIMAL(8,2))
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'user_jar:UnitTest.mydec';
public static void mydecs(BigDecimal c, BigDecimal[] d)
    throws SQLException

```

The following example is for a FLOAT or REAL parameter data type.

```
CREATE PROCEDURE myfloat
  (IN  i FLOAT,
   OUT j FLOAT )
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'user_jar:UnitTest.myfloat';
public static void myfloat(
  Double c,
  Double[] Result) throws SQLException
```

The following example is for a DATE parameter data type.

```
CREATE PROCEDURE mydatediagret
  (IN  d DATE,
   OUT o DATE )
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'user_jar:UnitTest.mydatediagret';
public static void mydatediagret(
  java.sql.Date date,
  java.sql.Date[] Result) throws SQLException
```

The following example is for a TIME WITH TIME ZONE parameter data type.

```
CREATE PROCEDURE mytimezdiagret
  (IN  t TIME WITH TIME ZONE,
   OUT d TIME WITH TIME ZONE )
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'user_jar:UnitTest.mytimezdiagret';
public static void mytimezdiagret(
  java.sql.Time time,
  java.sql.Time[] Result) throws SQLException
```

Because the Java Virtual Machine implementation does not support leap seconds, the maximum value of the SECOND field is 59.999999 rather than 61.999999.

The following example is for a TIMESTAMP WITH TIME ZONE parameter data type.

```

CREATE PROCEDURE mytszdiagret
  (IN  t TIMESTAMP WITH TIME ZONE,
   OUT d TIMESTAMP WITH TIME ZONE )
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'user_jar:UnitTest.mytszdiagret';
public static void mytszdiagret (
  java.sql.Timestamp p1,
  java.sql.Timestamp[] result) throws SQLException

```

Because the Java Virtual Machine implementation does not support leap seconds, the maximum value of the SECOND field is 59.999999 rather than 61.999999.

The following example is for an INTERVAL YEAR parameter data type.

```

CREATE PROCEDURE intcpy
  (IN  parameter_1 INTERVAL YEAR,
   OUT t           INTERVAL YEAR )
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'user_jar:UnitTest.intcpy';
public static void intcpy( String param_1,
  String[] result ) throws SQLException

```

The following example is for either a CHARACTER or a VARCHAR parameter data type.

```

CREATE PROCEDURE strcpy
  (IN  parameter_1 VARCHAR(15),
   OUT t           VARCHAR(15) )
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'user_jar:UnitTest.strcpy';
public static void strcpy( String param_1,
  String[] result ) throws SQLException

```

The following example is for a CLOB parameter data type.

```

CREATE PROCEDURE myclobdiagret2
  (IN  b CLOB)
LANGUAGE JAVA
MODIFIES SQL DATA

```



```

PARAMETER STYLE JAVA
EXTERNAL NAME 'user_jar:UnitTest.myclobdiagret2';
    public static void myclobdiagret2(
        java.sql.Clob data) throws SQLException
{
    Connection conn =      DriverManager.getConnection("jdbc:default:connection");
    PreparedStatement p = conn.prepareStatement("ins into tab2
                                                values(1,?)");

    p.setClob(1, data);
    p.executeUpdate();
}

```

The following example is for a VARBYTE parameter data type:

```

CREATE PROCEDURE mybvdiagret
    (IN c  varbyte(30),
     OUT d  varbyte(30))
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'user_jar:UnitTest.mybvdiagret';
public static void mybvdiagret(
    byte[] data,
    byte[][] Result ) throws SQLException

```

The following example is for a BLOB parameter data type:

```

CREATE PROCEDURE myblobdiagret
    (IN b  blob,
     OUT c blob)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'user_jar:UnitTest.myblobdiagret';
public static void myblobdiagret(
    java.sql.Blob data,
    java.sql.Blob[] Result) throws SQLException

```

Related Information

- [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#)
- [DROP AUTHORIZATION](#)
- [DROP MACRO](#)

- [RENAME MACRO](#)
- [SHOW object](#)

For information about coding external SQL procedures, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

For information about how to use the Teradata implementation of the JDBC API, see *Teradata JDBC Driver Reference*, available at <https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html>.

For information about how to use CLIV2 to issue SQL function calls, see the following:

- *Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems*, B035-2417
- *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418

CREATE PROCEDURE and REPLACE PROCEDURE (SQL Form)

Directs the SQL procedure compiler to create or replace a procedure from the SQL statements in the remainder of the source text and creates the SQL definition used to invoke the procedure.

REPLACE PROCEDURE directs the SQL procedure compiler to replace the definition of an existing SQL procedure.

If the specified SQL procedure does not exist, REPLACE PROCEDURE creates a new procedure by that name from the SQL statements in the remainder of the source text.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges: CREATE PROCEDURE

You must have the CREATE PROCEDURE privilege on the containing database or user for the procedure. The creator of an SQL procedure is automatically granted the DROP PROCEDURE and EXECUTE privileges WITH GRANT OPTION.

Vantage checks SQL procedure privileges differently depending on the definition of the SQL SECURITY clause option for the procedure.

Note that you cannot specify the OWNER option for the SQL SECURITY clause unless you have the explicitly granted CREATE OWNER PROCEDURE privilege to permit you to create an SQL procedure in a database or user other than your default database or user. This is an extremely important privilege for the Security Administrator to safeguard, ensuring that only those users who absolutely require the privilege be granted it.

If the parameter type is a UDT or an Array/Varray with a UDT element type, you must have at least one of the following privileges:

- UDTUSAGE on the specified UDT

- UDTUSAGE on the SYSUDTLIB database
- UDTTYPE on the SYSUDTLIB database

For more information, see [Rules for SQL Procedure Privileges](#).

Required Privileges: REPLACE PROCEDURE

You must have the DROP PROCEDURE privilege on the procedure or its containing database or user to replace it.

You must have the CREATE PROCEDURE privilege on the procedure or its containing database or user if it does not already exist.

The user replacing a procedure must have the privileges for all objects it accesses.

Once a procedure has been replaced, its immediate owner is its containing database or user, not the user who replaced it. The immediately owning database must have all the appropriate privileges for executing the procedure, including WITH GRANT OPTION.

Privileges Granted Automatically

When you create a new procedure, Vantage automatically grants the following privileges on it.

- DROP PROCEDURE
- EXECUTE PROCEDURE

CREATE PROCEDURE and REPLACE PROCEDURE Syntax (SQL Form)

```
{ CREATE | REPLACE } PROCEDURE [ database_name. | user_name. ] procedure_name
( [ parameter_specification [, ...] ] )
SQL_data_access
[ DYNAMIC RESULT SETS number_of_sets ]
[ SQL SECURITY privilege_option ]
statement [ ; ]
```

parameter_specification

```
[ IN | OUT | INOUT ] parameter_name data_type
```

SQL_data_access

```
{ CONTAINS SQL | MODIFIES SQL DATA | READS SQL DATA }
```

privilege_option

```
{ CREATOR | DEFINER | INVOKER | OWNER }
```

statement

```
{ SQL_statement |
  BEGIN SQL_multistatement_request END REQUEST |
  compound_statement |
  open_statement |
  fetch_statement |
  assignment_statement |
  condition_statement |
  [ label_name : ] iteration_statement [ label_name ] |
  diagnostic_statement |
  ITERATE label_name |
  LEAVE label_name
}
```

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [( fractional_seconds_precision)] [WITH TIME
ZONE] |

  INTERVAL YEAR [(precision)] [TO MONTH] |

  INTERVAL MONTH [(precision)] |

  INTERVAL DAY [(precision)]
    [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL HOUR [(precision)]
    [TO { MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL MINUTE [(precision)]
    [ TO SECOND [(fractional_seconds_precision)] ] |

  INTERVAL SECOND [( precision [, fractional_seconds_precision ] ) ] |

  PERIOD (DATE) |
```

```

PERIOD ({ TIME | TIMESTAMP } [(precision)] [ WITH TIME ZONE ]) |

REAL |

DOUBLE PRECISION |

FLOAT [(integer)] |

NUMBER [( { integer | *} [, integer ]...)] |

{ DECIMAL | NUMERIC } [(integer [, integer ]...)] |

{ CHAR | BYTE | GRAPHIC } [(integer)] |

{ VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [(integer)] |

LONG VARCHAR |

LONG VARGRAPHIC |

{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
(integer [ G | K | M ]) |

[SYSUDTLIB.] { XML | XMLTYPE }
[(integer [ G | K | M ])] [ INLINE LENGTH integer ] |

[SYSUDTLIB.] JSON [(integer [ K | M ])] [ INLINE LENGTH integer ]
[ CHARACTER SET { UNICODE | LATIN } ] |

[SYSUDTLIB.] ST_GEOMETRY [(integer [ K | M ])] [ INLINE LENGTH
integer ] |

[SYSUDTLIB.] DATASET [(integer [ K | M ])]
[ INLINE LENGTH integer ] storage_format |

[SYSUDTLIB.] { UDT_name | MBR | ARRAY_name | VARRAY_name }
}

```

compound_statement

```

[ label_name : ] BEGIN
[ local_declaration [...] ]

```

```

[ cursor_declaration [...] ]
[ condition_handler [...] ]
[ statement [...] ]
END [ label_name ]

```

open_statement

```

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

```

fetch_statement

```

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

```

assignment_statement

```

SET assignment_target = assignment_source

```

condition_statement

```

{ case_statement | if_statement }

```

iteration_statement

```

{ while_statement | loop_statement | for_statement | repeat_statement }

```

diagnostic_statement

```

{ { SIGNAL signal_specification | RESIGNAL [ signal_specification ] }
  [ SET condition_information_item = value ] |

  GET DIAGNOSTICS [ EXCEPTION condition_number ]
  diagnostic_assignment [,...]
} ;

```

storage_format

```

STORAGE FORMAT { Avro | CSV [ CHARACTER SET { UNICODE | LATIN } ] }
[ WITH SCHEMA [ database. ] schema_name ]

```

local_declaration

```

DECLARE { variable_name [,...] data_type [ DEFAULT { literal | NULL } ] |
        condition_name CONDITION [ FOR sqlstate_code ]
}

```

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 }
[ PREPARE statement_name
  FROM { 'statement_string' | statement_string_variable } ] [;]

```

condition_handler

```

DECLARE { { CONTINUE | EXIT } HANDLER | condition_name CONDITION }
[ FOR
  { sqlstate_specification [,...] handler_action_statement |

    { SQLEXCEPTION | SQLWARNING | NOT FOUND | condition_name } [,...]
    handler_action_statement |

    sqlstate_specification [,...]
  }
] ;

```

case_statement

```

CASE { operand_1 when_operand_clause [...] |
      when_condition_clause [...]
}

```

```

    [ ELSE statement; [...] ]
END CASE

```

if_statement

```

IF conditional_expression THEN statement; [...]
  [ ELSEIF conditional_expression THEN statement; [...] ][...]
  [ ELSE statement; [...] ]
END IF

```

while_statement

```

WHILE conditional_expression
  DO statement; [...]
END WHILE

```

loop_statement

```

LOOP
  statement; [...]
END LOOP

```

for_statement

```

FOR for_loop_variable AS [ cursor_name CURSOR FOR ] cursor_specification
  DO statement; [...]
END FOR

```

repeat_statement

```

REPEAT
  statement; [...] UNTIL conditional_expression
END REPEAT

```

signal_specification

```

{ condition_name | SQLSTATE [ VALUE ] SQLSTATE_code }

```


diagnostic_assignment

```
{ parameter_name | variable_name } = statement_information_item
```

cursor_specification

```
SELECT selection [,...] FROM source WHERE_clause [
other_SELECT_clause [...] ]
```

sqlstate_specification

```
SQLSTATE [ VALUE ] sqlstate_code
```

when_operand_clause

```
WHEN operand_2 THEN statement; [...]
```

when_condition_clause

```
WHEN conditional_expression THEN statement; [...]
```

selection

```
{ * | column_name [ [AS] alias_name ] | expression [AS] alias_name }
```

source

```
{ table_name [,...] |

  table_name { INNER | { LEFT | RIGHT | FULL } [ OUTER ] } JOIN
    table_name ON condition
}
```

CREATE PROCEDURE and REPLACE PROCEDURE Syntax Elements (SQL Form)

database_name

user_name

An optional qualifier for *procedure_name*.

If not specified, the current default database is used.

procedure_name

Name of the SQL procedure to be created.

procedure_name must be unique within the database.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

SQL_data_access

Specifies whether the procedure body accesses the database or contains SQL statements.

This clause is mandatory for all SQL procedures. See CREATE PROCEDURE (SQL Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for descriptions of the SQL data access options.

number_of_sets

Number of dynamic result sets that can be returned.

The range of valid values for *number_of_sets* is 0 through 15, inclusive.

The default value for *number_of_sets* is 0.

parameter_specification

IN

Parameter is for input only. IN is the default parameter type. If the parameter type is not specified, the parameter is assumed to be of the IN type.

OUT

Parameter is for output only.

INOUT

Parameter can be both input and output.

parameter_name

Name of a parameter or local variable that is replaced with an argument during procedure execution.

The maximum length of a parameter name is either 30 LATIN characters or 128 Unicode characters. For details about object name length limits, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

A maximum of 256 parameters can be specified with each procedure in a comma-separated list.

data_type

The data type of the parameter or local variable must be specified with each *parameter_name*.

You cannot specify a VARIANT_TYPE UDT as an IN parameter data type for an SQL procedure.

You can only specify the VARIANT_TYPE UDT type for callable input parameters within the body of the procedure using a NEW VARIANT_TYPE expression to pass dynamic UDTs into UDFs.

Note that the system does not default to the data type you assign to an INOUT parameter at the time the procedure is created when the procedure is called. Instead, it defaults to the smallest data type that can contain the specified input parameter. As a result, memory overflow errors can occur if the output parameter returned to an INOUT parameter cannot be contained by the default data type set for that parameter by the system. See CREATE PROCEDURE (External Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 and CALL in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for details.

For parameters, you can specify CHARACTER SET and either CASESPECIFIC or NOT CASESPECIFIC as an attribute of the character data type.

If you specify both, you must specify them in the following order.

- CHARACTER SET

You cannot specify a character server data set of KANJI1. If you attempt to do so, Vantage aborts the request and returns an error to the requestor.

- CASESPECIFIC or NOT CASESPECIFIC

You cannot specify other data type attributes, such as NOT NULL, UPPERCASE, or FORMAT, for SQL procedures.

The keyword DEFAULT is not allowed before the character set.

For local variables, you can specify CHARACTER SET and NOT CASESPECIFIC in any order after *data_type*.

If you do not specify CHARACTER SET, the character set defaults to the character set of the user creating or compiling the SQL procedure.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for a list of the data type encodings that SQL procedure IN, INOUT, and OUT parameters can return to a client application.

statement

The SQL procedure body.

You must specify either a single statement or a BEGIN-END compound statement.

You cannot specify any of the following as a single statement in an SQL procedure body:

- Local variable, cursor, or handler declarations
- Cursor statements

The following SQL statement sets can be specified as the SQL procedure body.

You can also specify any of the statements in these categories in a `statement_list` within compound statements, condition statements (CASE and IF), iteration statements, and in the handler action part of a condition handler declaration.

SQL_statement

One of the following:

- SQL DML, DDL, or DCL statement supported by SQL procedures. This includes dynamic SQL statements.
- Control statements.

Note:

SQL DCL and DDL statements for administering row-level security are not allowed in a stored procedure.

SQL_multistatement_request

An SQL multistatement request.

The format and rules for specifying multistatement requests in an SQL procedure are the same as those for all other multistatement request applications.

compound statement

BEGIN-END statement enclosing a set of declarations and statements.

You can specify local variable declarations, cursor declarations, condition handler declarations and SQL and control statements within a compound statement.

Nesting of compound statements is allowed.

You can specify these options for a compound statement.

label_name

Label for a BEGIN-END compound statement in the procedure, or for an iteration statement (WHILE, LOOP, FOR and REPEAT).

The beginning label must be suffixed with a COLON character (:). An ending label is not mandatory, but if an ending label is specified, you must also specify an equivalent beginning label.

The label name of the BEGIN-END compound statement cannot be reused in an iteration statement. A label name cannot be reused within a group of nested compound statements or nested iteration statements, but can be reused for different non-nested iteration statements or non-nested compound statements.

Using label names for each BEGIN-END compound statement is recommended if you specify nested compound statements in an SQL procedure.

A local variable can be optionally qualified with the label of the corresponding compound statement in which the variable is declared. This helps in avoiding conflicts that might be caused by duplicate local variables in nested compound statements.

BEGIN

Keyword introducing a compound statement.

A compound statement defining the procedure body contains all the SQL statements and declarations of the procedure.

You must terminate a compound statement with the END keyword.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

statement_list

Any of the following:

- SQL DML, DDL, or DCL statements supported by SQL procedures, including dynamic SQL statements.

You can also use the CALL statement invoking the *procedure_name* being created.

- control statements, including the BEGIN-END compound statement.

You can use any number of statements in a *statement_list*. Each must be terminated by a SEMICOLON (;) character.

END

Terminating keyword for a BEGIN-END compound statement.

open_statement

You can specify these options for open statements.

cursor_name

Name of the cursor to be opened.

USING

Variables used as input to the SQL statement by *cursor_name*.

The USING clause is valid only for result set cursors.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

SQL_identifier

parameter_reference

Set of variables to be supplied as input to the prepared SQL request.

The maximum length of a database object name is either 30 LATIN characters or 128 Unicode characters. For details about object name length limits, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

SQL_identifier or *parameter_reference* must be a valid SQL variable and must be declared prior to the OPEN statement.

fetch_statement

You can specify these options for fetch statements.

cursor_name

Name of the cursor.

parameter_reference

Name of an SQL parameter or string variable that contains the SQL text string that is to be executed dynamically.

assignment_statement

The SET statement.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

You can specify these options for assignment statements.

SET

A keyword introducing the SET statement used for assigning a value to a variable or parameter.

You can assign a UDT expression to a UDT variable or parameter; however, the mutator SET clause syntax (see UPDATE in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for details about mutator SET clause syntax) is not supported within a SET statement. The workaround for this is to use the standard non-mutator SET clause format, with the column reference on one side of the equal sign and a UDT expression containing the appropriate mutators on the other side.

assignment_target

Name of the variable or parameter to be assigned a value.

If you are assigning to a parameter, it cannot be an IN parameter.

If you are assigning to any variable, it cannot be named *QUERY_BAND*.

You cannot set values for status variables in SQL procedures.

assignment_source

Arithmetic and string expressions that contain the value to be assigned to a variable.

SQL procedure local variables, status variables, IN or INOUT parameters, and FOR loop column and correlation names can be specified in the *assignment_source*, which is also known as a value expression.

condition_statement

Either a CASE statement or an IF statement.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

You can specify these options for condition statements.

CASE

Keyword introducing the CASE conditional statement to qualify control statements.

You can use either the Simple Case statement form or the Searched Case statement.

operand_2

Value expressions or arithmetic and string expressions.

SQL procedure local variables, status variables, IN or INOUT parameters, and FOR loop column and correlation names can be specified in the value expression.

THEN

ELSE

Keywords introducing the statements to be performed in the CASE statement.

statement

A valid SQL statement or SQL control statement.

conditional_expression

A boolean search condition used to evaluate whether a statement or statements embedded within an IF, CASE, REPEAT, or WHILE clause should be performed.

Arithmetic expressions, logical expressions, UDT expressions, and status variables can all be specified as conditional expressions.

You cannot use IN and NOT IN operators if the conditional list contains any local variables, parameters, or cursor correlation names.

OUT parameters and subqueries are not allowed in *conditional_expression*.

END CASE

Keywords marking the end of a CASE statement.

IF

Keyword introducing a conditional expression to qualify SQL control statements.

You can also use these variations of IF with the following additional clauses:

- IF-THEN-END IF
- IF-THEN-ELSE-END IF
- IF-THEN-ELSEIF-END IF
- IF-THEN-ELSEIF-ELSE-END IF

ELSE

ELSEIF

A keyword introducing the THEN, ELSE, or ELSEIF clause in an IF statement.

END IF

Keywords marking the end of the IF statement.

iteration_statement

A WHILE, LOOP, FOR, or REPEAT statement.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

You can specify these options for iteration statements.

WHILE

Keyword introducing an iteration statement to repeat the execution of one or more statements within its defined scope.

The associated condition is checked before each iteration (including the first) and if true, the statements are performed. Otherwise, the WHILE statement completes with no further iterations.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

DO

Keyword introducing the sequence of statements to be performed in the WHILE statement.

END WHILE

Keywords indicating the ending of the sequence of statements in the WHILE statement.

LOOP

Keyword introducing an iteration statement that repeats the execution of one or more statements embedded within the defined loop.

END LOOP

Keywords that terminate a LOOP.

FOR

Keyword introducing an iteration statement to repeat the execution of one or more statements for each row fetched by the cursor.

for_loop_variable

Name of the FOR loop.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

cursor_name

Name of the cursor.

The maximum length of a database object name is either 30 LATIN characters or 128 Unicode characters. For details about object name length limits, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

CURSOR FOR

Optional keywords qualifying the *cursor_name* as the cursor for the statement set to follow.

cursor_specification

The SELECT statement specifying the column names or expressions from which to fetch data rows in the CURSOR statement.

DO

Keyword introducing the sequence of statements to be performed in the CURSOR statement.

END FOR

Keywords ending the CURSOR statement.

REPEAT

Keyword introducing the iteration statement that repeats the execution of one or more statements within the defined REPEAT statement.

UNTIL

Keyword introducing a conditional expression to qualify the statements in the loop.

END REPEAT

Keywords ending the REPEAT statement.

diagnostic_statement

diagnostic statement

A SIGNAL, RESIGNAL, or GET DIAGNOSTICS statement. For details, see *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148.

You can specify these options for diagnostic statements.

SIGNAL

Keyword introducing the SIGNAL SQL procedure diagnostic statement. See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

condition_name

Name of a variable declared to handle a condition within an SQL procedure.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

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.

RESIGNAL

Keyword introducing the RESIGNAL SQL procedure diagnostic statement. See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

condition_name

Name of a variable declared to handle a condition within an SQL procedure.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

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_code

Value for an SQLSTATE to be assigned to RETURNED_SQLSTATE in the Condition Area.

condition_information

One of the field names from the Condition Area of the Diagnostics Area.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

value

Text or numeric value, as appropriate, to be assigned to the specified condition information name.

GET DIAGNOSTICS

Keywords introducing the GET DIAGNOSTICS SQL procedure diagnostic statement. See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

parameter_name

Parameter whose value is set to the value contained in *statement_information_item*.

The maximum length of a database object name is either 30 LATIN characters or 128 Unicode characters. For details about object name length limits, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

variable_name

Variable whose value is set to the value contained in *statement_information_item*.

statement_information_item

One of the field names from the Statement Area of the Diagnostics Area. See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

EXCEPTION

A language element that indicates to return information from the Condition Area of the Diagnostics Area.

condition_number

Number, parameter, or variable that resolves to the number of the Condition Area from which information is to be retrieved.

parameter_name***variable_name***

An output parameter or variable to which the *condition_information_item* retrieved from the specified Condition Area is assigned.

The maximum length of a database object name is either 30 LATIN characters or 128 Unicode characters. For details about object name length limits, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

condition_information_item

Name of the Condition Area field from which condition information is to be retrieved. See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for a list of the valid condition information item names.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

ITERATE *label_name*

ITERATE

A keyword introducing the ITERATE statement to terminate the execution of the current iteration of a labeled iteration statement, and start the next iteration.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

label_name

The *label_name* references the iteration statement with which the label is associated, and whose current iteration is to be terminated.

The next iteration is started conditionally in WHILE or FOR, and unconditionally in LOOP and REPEAT.

LEAVE *label_name*

LEAVE

Keyword introducing the LEAVE statement to continue execution outside an iterated iteration by leaving a labeled statement.

For details, see *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148.

label_name

The *label_name* references the iteration statement with which the label is associated, and on which LEAVE is to be performed.

SQL_data_access

Specifies whether the procedure body accesses the database or contains SQL statements.

This clause is mandatory for all SQL procedures. See CREATE PROCEDURE (SQL Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for descriptions of the SQL data access options.

CONTAINS SQL

The procedure can execute SQL control statements.

The procedure neither reads nor modifies SQL data in the database. An example is a procedure whose body consists of just control statements local to the procedure.

MODIFIES SQL DATA

The procedure can execute all SQL statements that can validly be called from an SQL procedure.

This is the default option for SQL procedures that do not specify an SQL Data Access clause when the procedure is defined.

An example of such as statement is an UPDATE, INSERT or DELETE. This is the default

READS SQL DATA

The procedure cannot execute SQL statements that modify SQL data, but can execute statements that read SQL data.

An example is the FETCH statement.

DYNAMIC RESULT SETS

Number of dynamic result sets can be returned.

number_of_sets

The range of valid values for *number_of_sets* is 0 through 15, inclusive.

The default value for *number_of_sets* is 0.

privilege_option

Optional SQL security privilege set you assign to *procedure_name*.

See CREATE PROCEDURE (SQL Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for descriptions of the privilege options.

The keyword you specify for *privilege_option* determines the privileges that Vantage checks for the underlying objects specified in the request.

The following rules apply to all procedures.

- Objects referenced by any statement in the procedure need not have the WITH GRANT OPTION privilege. They require only the GRANT privilege on the referenced object.
- Vantage check CREATOR and OWNER privileges on DDL requests during execution.

CREATOR

Assign the privileges of the creator of the procedure regardless of its containing database or user.

DEFINER

Assign the privileges of the definer of the procedure. This is the default privilege option.

Vantage treats procedures as follows when the procedure is defined as a DEFINER whether explicitly or by default, for example, when the CREATOR of the procedure is not its OWNER:

- Vantage checks the privileges of the CREATOR during compilation of the procedure. The CREATOR must have the appropriate privilege for any object that a statement in the procedure references.
- Vantage checks the privileges of the OWNER, that is, the containing database or user for the procedure, during compilation. The OWNER must have the appropriate privileges for any object that is referenced by a statement in the procedure.
- During execution, Vantage checks the OWNER privileges for any object referenced by any statement in the procedure. Any access failures caused by not having the appropriate privileges return an error to the procedure, which it can handle if it is written to do so.
- Vantage grants the DROP and EXECUTE privileges to a UDF created in a database or user different from the CREATOR. An OWNER always has the implicit privilege to drop any object it owns. If the OWNER wants to execute the UDF, then it must grant the EXECUTE FUNCTION privilege to that function.

INVOKER

Assign the privileges of the user at the top of the current execution stack.

OWNER

Assign the privileges of the owner of the procedure, which are the privileges possessed by its containing database or user.

You cannot specify the OWNER option for this clause unless you have the explicitly granted CREATE OWNER PROCEDURE privilege to permit you to create an SQL procedure in a database or user other than your default database or user.

local_declaration

You can specify these options for a local declaration.

DECLARE

Keyword introducing a local variable declaration, cursor declaration, or condition handler declaration statement.

DECLARE is followed by a list of local variables, a cursor specification, or a list of handler declarations.

You can specify multiple local variable declarations, cursor declarations, or condition handler declarations for each procedure.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

variable_name

data_type

[*variable_name*]

[*database_name* | *user_name*] *table_name* %ROWTYPE

Name and data type of the local variable being declared.

See *SQL Fundamentals* for the rules for naming database objects.

Any number of local variables of the same data type can be specified as a comma-separated list. These variable names are replaced by data values during execution.

Local variables can have UDT and %ROWTYPE data types.

The %ROWTYPE attribute dynamically creates an implicit data type based on the structure of a base table or global temporary table row in *table_name*. The *table_name* variable cannot refer to a volatile table.

DEFAULT

Optional keyword for introducing a default value for the local variables.

If more than one local variable is specified along with a default value, that value applies to all of the local variables in the list.

literal

The literal must be compatible with the data type specified.

You can only specify a literal or NULL, not an expression.

NULL

Default value for the variables.

A variable is initialized to NULL if no default value is specified.

condition_name

Name for the declared condition that can be used to associate a symbolic condition name with a specific SQLSTATE value.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

sqlstate_code

SQLSTATE value assigned to *condition_name*.

***condition_name* CONDITION**

Name for the declared condition that can be used to associate a symbolic condition name with a specific SQLSTATE value.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

sqlstate_code

SQLSTATE value assigned to *condition_name*.

cursor_declaration

You can specify these options for a cursor declaration.

cursor_name

Name of the cursor.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

SCROLL

SCROLL allows the cursor to scroll forward to the next row or back to the first row of the response set.

NO SCROLL

When NO SCROLL is specified or when no option is specified, the cursor can only scroll forward to the next row. NO SCROLL is the default.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

CURSOR

Keyword qualifying the *cursor_name* as the cursor for the statement set to follow.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

WITHOUT RETURN

Specifies that the procedure does not return a result set.

WITHOUT RETURN is the default.

WITH RETURN ONLY

A result set cursor that is returned by the procedure, but that cursor cannot be fetched.

TO CALLER

The result set from the target procedure is to be returned to either of the following:

- the calling procedure.
- the calling client application.

This is the default.

TO CLIENT

The result set is to be returned to the client application even if called from a nested procedure.

If more than one SQL procedure specifies TO CLIENT, then the Teradata platform returns the results in the order opened.

FOR READ ONLY

The default option and is implicit.

FOR UPDATE

Makes the cursor updatable. Update or delete operations can be performed on the cursor.

PREPARE

An *SQL_statement_name* is to be set up to be executed dynamically.

The PREPARE statement is valid only for result set cursors.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

SQL_string

The SQL text that is to be executed dynamically.

SQL_string must be delimited by APOSTROPHE characters.

SQL_string_variable

Name of an SQL local variable that contains the SQL text string that is to be executed dynamically.

PREPARE

An *SQL_statement_name* is to be set up to be executed dynamically.

The PREPARE statement is valid only for result set cursors.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

SQL_string

The SQL text that is to be executed dynamically.

SQL_string must be delimited by APOSTROPHE characters.

SQL_string_variable

Name of an SQL local variable that contains the SQL text string that is to be executed dynamically.

cursor_specification

The SELECT statement specifying the column names or expressions from which to fetch data rows in the CURSOR clause.

You can specify these options for cursor specification statements.

SELECT

The columns or expressions specified by *column_name* form the cursor.

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

column_name

Name of the column set from which to fetch data.

AS

Optional introduction to *column_name_alias*.

alias_name

Alias for the table referenced by *table_name.column_name_alias* is also used to name expressions.

expression

Any valid SQL expression.

table_name

Name of a table or view from which rows are to be fetched.

search_condition

Optional search condition that must be satisfied by the row set to be fetched by the SELECT statement.

Other SELECT Clauses

Clauses such as ORDER BY, GROUP BY, and HAVING.

These clauses are valid only for read-only cursors. None is valid when specified with updatable cursors.

WITH ... BY clauses are not allowed in SQL procedure cursors.

condition_handler

You can specify these options for a condition handler.

CONTINUE**EXIT**

Type of condition handler being requested.

See *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 and *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details.

SQLSTATE

Keyword introducing an *sqlstate* value for which the exception or completion condition is to be handled.

You can specify any number of *sqlstate* values prefixed by SQLSTATE or SQLSTATE VALUE in a comma-separated list within each handler declaration.

VALUE

Optional keyword prefixing the *sqlstate* value.

sqlstate_code

Five-character string value that indicates a completion condition for the SQL statement. This is always delimited by APOSTROPHE characters.

A given *sql/state* value cannot be a part of more than one DECLARE HANDLER statements and cannot be repeated within a FOR clause.

SQLEXCEPTION

A keyword identifier for generic conditions.

You can specify these individually or in combination with one another, but you cannot specify any of them more than once within a given DECLARE HANDLER statement.

SQLEXCEPTION indicates a generic exception condition.

SQLWARNING

SQLWARNING indicates a generic completion condition.

NOT FOUND

NOT FOUND indicates a generic completion condition when no data is found.

condition_name

condition_name is a name that can be used to specify conditions for a handler to act on when the procedure definition specifies the condition name in a SIGNAL or RESIGNAL statement.

handler_action_statement

A single or compound statement that performs the handler action.

A single statement can be any one of the following:

- Any standard SQL statement, including dynamic SQL, supported by SQL procedures.
- Any SQL control statement.

The following are not valid as a single statement for *handler_action_statement*:

- Local variables
- Cursors
- Handler declarations

Usage Notes

Rules for SQL Procedure Privileges

Following are the rules for SQL procedure privileges and how they are checked:

- Dynamic and static SQL are governed by these rules.

Generally, the term *dynamic SQL* applies to any SQL statement whose complete text is not known until run time.

- The default privilege option for SQL procedures is DEFINER.
- When the creator of a procedure is also its immediate owner, all static DML and DCL statements within the procedure are valid.

The following table summarizes the rules for SQL procedure privilege assignment.

Note the following things about the column headings.

- For the column *Owner/Creator Relationship*, the word *owner* refers to the immediate owner.
- For the column *Privilege Checking Done at This Time: Compilation*, the appropriate CREATE and DROP privileges are also checked during compilation.
- For the column *Privilege Checking Done at This Time: Execution*, the EXECUTE PROCEDURE privilege is also checked at run time.

SQL SECURITY Privilege Option	Dynamic or Static SQL Statement?	Owner/Creator Relationship	Privilege Checking Done at This Time		Default Database or User
			Compilation	Execution	
DEFINER	Dynamic	O is not C	None	O and C	C
		O is C	None	O	C
	Static	O is not C	O and C	O and C	C
		O is C	O	O	C
INVOKER	Dynamic	O is not C	None	I	I
		O is C	None	I	I
	Static	O is not C	C	I	I
		O is C	C	I	I
OWNER	Dynamic	O is not C	COP	O	O
		O is C	None	O	O
	Static	O is not C	COP and O	O	O
		O is C	O	O	O
CREATOR	Dynamic	O is not C	None	C	C
		O is C	None	C	C
	Static	O is not C	C	C	C
		O is C	C	C	C

where:

Symbol or Text String ...	User Privileges Required to Submit a CREATE PROCEDURE or REPLACE PROCEDURE Request
None	None.
Error	One or more privileges it did not have at the time the CREATE PROCEDURE or REPLACE PROCEDURE request was compiled. Because the request aborts when this occurs, the cells for Privilege Check at Execution Time and Default Database are empty.
COP	CREATE OWNER PROCEDURE.
C	Privileges of the creator of the procedure.
I	Privileges of the invoker of the procedure.
O	Privileges of the owner of the procedure.
O and C	Privileges of the owner of the procedure and those of its creator.
O or C	Those of the owner of the procedure or those of its creator.
CREATOR	Those of the creator of the procedure regardless of its containing user or database.
DEFINER	Those currently defined. This is the default.
INVOKER	Those at the top of the current stack.
OWNER	Those of the immediate owner of the procedure (the user or database in which the procedure is contained).

This symbol or text string ...	Means that the immediate owner and creator of the procedure are ...
O is not C	different.
O is C	the same.

Using the first row for the INVOKER SQL SECURITY option as a model, you would interpret the table as follows: For the case when an SQL statement is invoked dynamically and the immediate owner of its containing procedure is not the same user as its creator, no privilege checking is done at the time the procedure is compiled, the privileges of the invoker of the procedure are checked at run time, and the default database or user that is used to implicitly qualify any unqualified object references within the SQL statements in the procedure body is that of the invoker.

The following examples show how the various SQL SECURITY options can be used:

- [Example: Specifying a SQL SECURITY—CREATOR](#)
- [Example: Specifying a SQL SECURITY—DEFINER](#)
- [Example: Specifying SQL SECURITY—INVOKER](#)
- [Example: Specifying a SQL SECURITY—OWNER](#)

Privilege Violations During Procedure Compilation

When privilege violations occur during procedure compilation, the table below lists the response.

Request Type	Privileges Checked	Response for Privilege Violations
DCL and DDL	CREATOR and OWNER	Warning message.
DML	CREATOR and OWNER	Error message for CREATOR. Warning message for OWNER.
Dynamic SQL	None	

Privilege Violations During Procedure Execution

When privilege violations occur during procedure execution, the table below lists the response.

Request Type	Privileges Checked	Response for Privilege Violations
DCL and DDL	CREATOR and OWNER	Error message.
DML	OWNER	Error message.
Dynamic SQL	CREATOR and OWNER	

Examples

Example: Creating a Procedure with Parameters of Different Data Types

The following example is for creating a valid procedure named *spParams* with six parameters of different data type combinations:

```
CREATE PROCEDURE sp_params (INOUT iop1 SMALLINT,
                           OUT   op1  DECIMAL(10,3),
                           IN    ip2  NUMERIC,
                           INOUT iop2 FLOAT,
                           OUT   op2  REAL,
                           IN    ip3  DOUBLE PRECISION)

BEGIN
  SET iop1=3;
  SET op1=5.2;
  SET iop2=10.2;
```



```
SET op2=iop2;
END;
```

Example: Specifying a SQL SECURITY—CREATOR

When you specify SQL SECURITY CREATOR, Vantage verifies and applies the privileges of the user who created the procedure.

The system uses the name of the creating (definer) user as the default qualifier for implicit qualification of any unqualified object references within the SQL statements in the procedure body.

The following example shows the CREATOR case for static SQL. In this example, *user_1* creates procedure *dyn_dml* in the SYSLIB database, and *user_2* calls SYSLIB.dyn_dml.

Vantage verifies the following privileges for this example.

AT this time ...	Vantage verifies the following privileges in the order indicated ...
compilation	<i>user_1</i> has the CREATE PROCEDURE privilege on the SYSLIB database. <i>user_1</i> has the INSERT privilege on the database object <i>user_1.t1</i> .
execution	<i>user_2</i> has the EXECUTE PROCEDURE privilege on the SQL procedure SYSLIB.dyn_dml. <i>user_1</i> has the INSERT privilege on the database object <i>user_1.t1</i> .

```
.LOGON user_1/user_1
.COMPILE FILE sp.sp1
/* sp.sp1 file
CREATE PROCEDURE SYSLIB.static_dml()
SQL SECURITY CREATOR
BEGIN
  INSERT INTO t1
  SELECT 1,1;
END;
/*
.LOGON user_2,user_2
CALL SYSLIB.static_dml();
```

For the dynamic SQL case, Vantage verifies the following privileges:

AT this time ...	Vantage verifies the following privileges in the order indicated ...
compilation	<i>user_1</i> has the CREATE PROCEDURE privilege on the SYSLIB database.
execution	<i>user_2</i> has the EXECUTE PROCEDURE privilege on the SQL procedure SYSLIB.dyn_dml. <i>user_1</i> has the INSERT privilege on the database object <i>user_1.t1</i> .

```

.LOGON user_1/user_1
.COMPILE FILE sp.sp1
/* sp.sp1 file
CREATE PROCEDURE SYSLIB.dyn_dml()
SQL SECURITY CREATOR
BEGIN
    CALL DBC."sysexecsql" ('INSERT INTO t1 (1,1);');
END;
/*
.LOGON user_2/user_2
CALL SYSLIB.dyn_dml();

```

Example: Specifying a SQL SECURITY—DEFINER

The DEFINER option is the SQL SECURITY clause default.

The following rules apply to the DEFINER option.

- Vantage checks the privileges of both the creator and the owner of the procedure at compilation and execution times.
- Static DDL and DML statements are valid in a procedure definition even if the creator and owner are not the same.

Vantage uses the name of the creating (definer) user as the default for implicit qualification of any unqualified object references within the procedure body.

The following example shows the DEFINER case for static SQL. In this example, *user_1* creates procedure *dyn_dml* in the SYSLIB database, and *user_2* calls SYSLIB.dyn_dml.

Vantage verifies the following privileges for this example.

AT this time ...	Vantage verifies the following privileges in the order indicated ...
compilation	<i>user_1</i> has the CREATE PROCEDURE privilege on the SYSLIB database. <i>user_1</i> has the INSERT privilege on <i>user_1</i> on the database object <i>user_1.t1</i> . The SYSLIB database has the INSERT privilege on the database object <i>user_1.t1</i> .
execution	<i>user_2</i> has the EXECUTE PROCEDURE privilege on the SQL procedure SYSLIB.dyn_dml. <i>user_1</i> has the INSERT privilege on the database object <i>user_1.t1</i> . The SYSLIB database has the INSERT privilege on the database object <i>user_1.t1</i> .

```

.LOGON user_1,user_1
.COMPILE FILE sp.sp1
/* sp.sp1 file
CREATE PROCEDURE SYSLIB.static_dml()
SQL SECURITY DEFINER

```

```

BEGIN
  INSERT INTO t1
  SELECT 1,1;
END;
/*
.LOGON user_2,user_2
CALL SYSLIB.static_dml();

```

For the dynamic SQL case, Vantage verifies the following privileges.

AT this time ...	Vantage verifies the following privileges in the order indicated ...
compilation	<i>user_1</i> has the CREATE PROCEDURE privilege on the SYSLIB database.
execution	<i>user_2</i> has the EXECUTE PROCEDURE privilege on the SQL procedure SYSLIB.dyn_dml. <i>user_1</i> has the INSERT privilege on the database object <i>user_1.t1</i> . The SYSLIB database has the INSERT privilege on the database object <i>user_1.t1</i> .

```

.LOGON user_1/user_1
.COMPILE FILE sp.sp1
/* sp.sp1 file
CREATE PROCEDURE SYSLIB.dyn_dml()
SQL SECURITY DEFINER
BEGIN
  CALL dbc."sysexecsql" ('INSERT INTO t1 (1,1);');
END;
/*
.LOGON user_2/user_2
CALL SYSLIB.dyn_dml();

```

Example: Specifying SQL SECURITY—INVOKER

The INVOKER logic uses the current top of the authorization stack to check for privileges, and the authorization identifier sitting on the top of the stack is used as the default qualifier for implicit qualification of any unqualified object references within the SQL statements in the procedure. When you specify SQL SECURITY INVOKER, Vantage uses the privileges on the top of the stack.

Static SQL

For static SQL, Vantage checks the privileges of the invoker during the creation of the procedure.

AT this time ...	Vantage verifies the following privileges in the order indicated ...
compilation	Creator has the CREATE PROCEDURE privilege on the SYSLIB database.

AT this time ...	Vantage verifies the following privileges in the order indicated ...
	Required privileges for creator on referenced objects but generates only warnings and allows stored procedure creation to succeed.
execution	EXECUTE PROCEDURE privilege for invoker. Required privileges for invoker on referenced objects and fails execution if privileges are missing.

For example, the DBC user creates the databases, tables, and user as follows:

```
CREATE DATABASE db1 AS PERM = 1e6;
CREATE DATABASE db2 AS PERM = 1e6;

CREATE TABLE db1.tab1(a INT, b INT);
CREATE TABLE db2.tab1(a INT, b INT);

CREATE DATABASE testdb1 AS PERM = 1e6;

CREATE USER User1 AS PERM = 1e6 PASSWORD = User1;

GRANT CREATE PROCEDURE ON testdb1 TO User1;
GRANT EXECUTE PROCEDURE ON testdb1 TO User1;
```

Use a text editor to create the stored procedure as follows:

```
sp1.sp1
-----
REPLACE PROCEDURE testdb1.sp1()
SQL SECURITY INVOKER
BEGIN
delete from db1.tab1;
insert into db1.tab1 select * from db2.tab1;
END;
```

Then, User1 compiles the stored procedure. The stored procedure creation is successful with warnings:

```
.compile file = sp1.sp1

*** Procedure has been created. 2 Errors/Warnings.
*** Warning: 5527 Stored Procedure Created with Warnings.
*** Total elapsed time was 1 second.

Warnings reported during compilation
-----
```

```
SPL5000:W(L4), E(3523):The user does not have DELETE access to db1.tab1.
SPL5000:W(L5), E(3523):The user does not have SELECT access to db2.tab1.
```

Use the SHOW PROCEDURE statement to verify the creation of the procedure:

```
SHOW PROCEDURE testdb1.sp1;

*** Text of DDL statement returned.
*** Total elapsed time was 1 second.

-----
REPLACE PROCEDURE testdb1.sp1()
SQL SECURITY INVOKER
BEGIN
delete from db1.tab1;
insert into db1.tab1 select * from db2.tab1;
END;
```

The privileges of the INVOKER are checked during execution and fails without the required privileges.

```
Call testdb1.sp1();
*** Failure 3523 SP1:The user does not have DELETE access to db1.tab1.
*** Total elapsed time was 1 second.
```

Dynamic SQL

For dynamic SQL, there are no privileges that can be checked during the creation of the procedure. Vantage generates a warning message only if the object does not exist. The system checks all privileges during the execution of the procedure.

The following example shows the INVOKER case for dynamic SQL. In this example, *user_1* creates procedure *dyn_dml* in the SYSLIB database, and *user_2* calls SYSLIB.dyn_dml.

Vantage verifies the following privileges for this example.

AT this time ...	Vantage verifies the following privileges in the order indicated ...
compilation	user_1 has the CREATE PROCEDURE privilege on the SYSLIB database.
execution	user_2 has the EXECUTE PROCEDURE privilege on the SQL procedure SYSLIB.dyn_dml. user_2 has the INSERT privilege on table user_2.t1.

```
.LOGON user_1,user_1
.COMPILE FILE sp.sp1
/* sp.sp1 file
```

```

CREATE PROCEDURE SYSLIB.dyn_dml()
SQL SECURITY INVOKER
BEGIN
    CALL dbc."sysexecsql" ('INSERT INTO t1 (1,1);');
END;
/*
.LOGON user_2,user_2
CALL SYSLIB.dyn_dml();

```

Example: Specifying a SQL SECURITY—OWNER

When you specify SQL SECURITY OWNER, Vantage verifies and applies the privileges of the owner of the procedure, which means those of its containing database or user.

The SQL SECURITY OWNER clause is not a valid option for SQL procedure creation unless you have been granted the CREATE OWNER PROCEDURE privilege to permit you to create an SQL procedure in another database. CREATE OWNER PROCEDURE is an explicit right that must be granted explicitly to a user or database.

Vantage uses the owner identifier as the default qualifier for implicit qualification of any unqualified object references within the SQL statements in the procedure.

The following example shows the OWNER case for static SQL. In this example, *user_1* creates the SQL procedure *dyn_dml* in the SYSLIB database, and *user_2* calls SYSLIB.dyn_dml.

Vantage verifies the following privileges for this static SQL example.

AT this time ...	Vantage verifies the following privileges in the order indicated ...
compilation	<i>user_1</i> has the CREATE OWNER PROCEDURE privilege on the SYSLIB database. The SYSLIB database has the INSERT privilege on table SYSLIB.t1.
execution	<i>user_2</i> has the EXECUTE PROCEDURE privilege on the SQL procedure SYSLIB.dyn_dml. The SYSLIB database has the INSERT privilege on table SYSLIB.t1.

```

.LOGON user_1,user_1
.COMPILE FILE sp.sp1
/* sp.sp1 file
CREATE PROCEDURE SYSLIB.static_dml()
SQL SECURITY OWNER
BEGIN
    INSERT INTO t1
    SELECT 1,1;
END;
/*

```

```
.LOGON user_2,user_2
CALL SYSLIB.static_dml;
```

For the dynamic SQL case, Vantage verifies the following privileges:

AT this time ...	Vantage verifies the following privileges in the order indicated ...
compilation	<i>user_1</i> has either the CREATE OWNER PROCEDURE privilege or the CREATE PROCEDURE privilege on the SYSLIB database.
execution	<i>user_2</i> has the EXECUTE PROCEDURE privilege on the SQL procedure SYSLIB.dyn_dml. The SYSLIB database has the INSERT privilege on table SYSLIB.t1.

```
.LOGON user_1,user_1
.COMPILE FILE sp.sp1
/* sp.sp1 file
CREATE PROCEDURE syslib.dyn_dml()
SQL SECURITY OWNER
BEGIN
    CALL dbc."sysexecsql" ('INSERT INTO t1 (1,1);');
END;
/*
.LOGON user_2,user_2
CALL SYSLIB.dyn_dml();
```

Example: Passing the Value of *n* for a TOP *n* Operator to an SQL Procedure

The following procedure passes the value for *n* to the TOP *n* operator in its SELECT request as the integer parameter *a*.

```
CREATE PROCEDURE fct (IN a INTEGER)
BEGIN
    DECLARE hc1 INTEGER;
    DECLARE hc2 INTEGER;
    FOR
        RecordPointer AS c_sptable CURSOR FOR
        SELECT TOP :a x1 AS c_c1, y1 AS c_c2
        FROM t1
    DO
        SET hc1 = RecordPointer.c_c1;
        SET hc2 = RecordPointer.c_c2;
        INSERT INTO t2 VALUES (:hc1,:hc2);
    END FOR ;
```

```
ROLLBACK;
END;
```

Example: Creating a Procedure with Local Variable, Cursor, and Conditional Handler Declarations

The following example shows the CREATE PROCEDURE statement for creating a procedure named *sp_sample3* with local variable declarations, cursor declarations, and condition handler declarations:

```
CREATE PROCEDURE sp_sample3(OUT p1 CHARACTER(80))
BEGIN
  DECLARE i INTEGER;
  DECLARE emp_cursor CURSOR WITHOUT RETURN FOR
    SELECT emp_name, salary FROM emp_details ORDER BY dept_code;
  DECLARE dept_cursor CURSOR WITHOUT RETURN FOR
    SELECT dept_name from department;
  DECLARE EXIT HANDLER
    FOR sqlstate '42000'
  BEGIN
    OPEN emp_cursor;
    SET p1='FAILED TO INSERT ROW';
  END;
  DECLARE CONTINUE HANDLER
    FOR SQLSTATE value '23505'
  BEGIN
    SET p1='FAILED TO INSERT ROW';
  END;
  DECLARE CONTINUE HANDLER
    FOR SQLEXCEPTION
  BEGIN
    SET p1='FAILED TO INSERT ROW';
  END;
  INSERT INTO table1 VALUES(1000,'aaa');
END;
```

Example: Creating SQL Procedures with Single Statement

The following are examples of procedures with a single statement that is not enclosed in a BEGIN END compound statement:

- A procedure with a single SET statement:


```
CREATE PROCEDURE sp_sample4a (IN in_param INTEGER,
                             OUT out_param INTEGER)
    SET out_param = in_param + 1;
```

- A procedure with a WHILE statement:

```
CREATE PROCEDURE sp_sample4b (IN in_param INTEGER,
                             OUT out_param INTEGER)

WHILE (in_param < 20)
    SET out_param = in_param + 1;
END WHILE;
```

Example: Using Nested Compound Statements in a Procedure

The following example shows nesting of compound statements.

The local variable and cursor name of the outer compound statement labeled *L1* are used in the inner compound statement labeled *L2*.

```
CREATE PROCEDURE sp_sample1(IN p_name CHAR(30), INOUT p_amt INTEGER)
L1: BEGIN
    DECLARE cvar1, v_amt INTEGER;
    DECLARE cursor1 CURSOR WITHOUT RETURN FOR
        SELECT c1 AS c_c1, c2 AS c_c2
        FROM temp;
    L2: BEGIN
        DECLARE v_name CHARACTER(30);
        DECLARE CONTINUE HANDLER FOR SQLSTATE '42000'
            BEGIN
                OPEN cursor1;
                INSERT INTO temp VALUES (v_name, v_amt);
            END;
        SET vName = p_name;
        -- tab1 table does not exist
        INSERT INTO tab1 VALUES (v_name, v_amt);
        FETCH cursor1 INTO cvar1, p_amt;
    END L2;
END L1;
```

Example: Splitting a CLOB Value in Half and Inserting the Pieces into Two Different Tables

The following example shows an SQL procedure used to split a CLOB in half and then insert the halves into two separate tables.

First the table definitions:

```
CREATE TABLE tabc1 (
  a1 INTEGER,
  b1 CLOB);
CREATE TABLE tabc2 (
  a2 INTEGER,
  b2 CLOB);
```

The following SQL code defines the procedure:

```
CREATE PROCEDURE clobsplitter(IN    a1 INTEGER,
                              IN    a2 INTEGER,
                              INOUT b  CLOB )
  BEGIN
    DECLARE localclob CLOB;
    DECLARE len INTEGER;
    SET len = CHARACTERS(b);
    SET localclob = SUBSTR(b, 1, len/2);
    INSERT tabc1 (:a1, localclob);
    INSERT tabc2 (:a2, SUBSTR(b, len/2 + 1));
    SET b = localclob || SUBSTR(b, len/2 + 1);
  END;
```

The stages in the process followed by this procedure are as follows:

1. The first half of CLOB *b* is assigned to local variable named *localclob*.
2. The contents of CLOB are inserted into table *tabc1*.
3. The second half of CLOB *b* is inserted into table *tabc2* without using a local LOB variable.
4. Local variable *localclob* is concatenated with the second half of *b* and assigned to *b*.
5. The contents of *b* are returned to the application.

If *clobsplitter* is called by another procedure, then *b* is passed to the other procedure.

Example: Converting a CLOB Containing XML Data into Rows for Insertion into a Table

The following example creates a procedure that converts a CLOB value containing XML sales data into a set of rows that are inserted into a *sales* table.

First the table definitions:

```
CREATE TABLE sales (
  partnum    INTEGER,
  qty sold   INTEGER,
  storecode  INTEGER,
  salesdate  DATE)
PRIMARY INDEX (partnum, qty sold, storecode, salesdate);
CREATE TABLE saleslog (
  storecode  INTEGER,
  salesdate  DATE,
  sales      CLOB,
  logdate    DATE,
  logtime    TIME)
PRIMARY INDEX (storecode, salesdate, logdate, logtime);
```

The following SQL code defines the procedure:

```
CREATE PROCEDURE stores_sales_procedure (storecode INTEGER,
                                         salesdate DATE,
                                         salesclob CLOB)
BEGIN
  INSERT INTO saleslog (:storecode, :salesdate, :salesclob,
                      CURRENT_DATE, CURRENT_TIME);

  INSERT INTO sales
    SELECT * FROM TABLE (xmlparser(:salesclob));
END;
```

The stages in the process followed by this procedure are as follows:

1. The sales CLOB is inserted into a log table named *saleslog*.
2. The CLOB is passed by means of the variable *salesclob* to a user-defined table function named *xmlparser* (see [CREATE FUNCTION and REPLACE FUNCTION \(Table Form\)](#)).
3. The system invokes table function *xmlparser* on every AMP.
4. The table function invokes *fnc_loblocal* (the definitions for *xmlparser* and *fnc_loblocal* are not shown for this example) to detect whether the CLOB is located on the same AMP on which the function is executing.

5. The function that has local access to the CLOB strips out the XML sales data and returns standard relational table rows to be inserted into *sales*.

Example: Creating an SQL Procedure with a Multistatement Request

The following procedure executes the statements in the multistatement request in parallel:

```
CREATE PROCEDURE salesadjust(IN item    INTEGER,
                             IN numsold INTEGER)
BEGIN
  DECLARE price DECIMAL(8,2);
  SELECT item_price INTO price
  FROM pricetbl
  WHERE item = pricetbl.item_no;
  BEGIN REQUEST
    UPDATE sales_summary
    SET total_sales = total_sales + price * numsold
    WHERE item = sales_summary.item_no
    ;UPDATE inventory
    SET item_no = item_no - numsold
    WHERE item = inventory.item_no;
  END REQUEST;
END;
```

The stages in the process followed by this procedure are as follows:

1. The price of an item is retrieved from *pricetbl*.
2. The total sales amount is computed by multiplying the total price by the number of items sold.
3. Two update requests are dispatched to the AMPs in parallel:

The first updates a *total_sales* summary table that keeps track of the cash made on all items sold.

The second adjusts *inventory* to reflect the remaining quantity stocked for that particular item.

Example: Creating a Procedure with Multistatement Request Using a Dynamic SQL Call Statement

Following is an example of a dynamic SQL call using DBC.SysExecSQL.

```
CREATE PROCEDURE salesadjust(IN item    INTEGER,
                             IN numsold INTEGER)
BEGIN
  DECLARE price DECIMAL(8,2);
  DECLARE update1, update2 VARCHAR(128);
```

```

SELECT item_price INTO price
FROM PriceTbl
WHERE item = pricetbl.item_no;
SET update1 = 'UPDATE sales_summary' ||
  'SET total_sales = total_sales + ' ||
  price ' || ' * ' || numsold ||
  'WHERE ' || item || ' = sales_summary.item_no;';
SET update2 = 'UPDATE inventory' ||
  'SET item_no = item_no - ' || numsold
  'WHERE ' || item || ' = inventory.item_no;';
BEGIN REQUEST
  CALL dbc.SysExecSQL(update1 || update2);
END REQUEST;
END;

```

Example: Setting the Transaction Query Band Using a Parameter

Both of the following procedures set the transaction query band using a parameter named *qbin*:

```

CREATE PROCEDURE setqbmsr(
  IN qbin VARCHAR(60))
BEGIN REQUEST
  SET QUERY_BAND = :qbin FOR TRANSACTION;
  INS abc(1,2);
END REQUEST;

CREATE PROCEDURE qbparm1 (
  IN par1 INTEGER,
  IN qbin VARCHAR(60),
  OUT qbout VARCHAR(60))
BEGIN REQUEST
  BEGIN TRANSACTION;
  SET QUERY_BAND = :qbin FOR TRANSACTION;
  SELECT GetQueryBand() INTO qbout;
  END TRANSACTION;
END REQUEST;

CALL qbparm1(10, 'kid=kate;', par3);
*** Procedure has been executed.
*** Total elapsed time was 1 second.
qbout

```

```
-----  
=T> kid=kate;
```

Example: Creating a Procedure with Dynamic SQL Using the SQL PREPARE Statement

The following CREATE PROCEDURE request creates a procedure that contains dynamic SQL using a PREPARE statement:

```
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 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;  
    FETCH NEXT FROM cstmt INTO item, sales;  
  END;
```

The PREPARE can also be written as follows using parameter markers as seen in the following procedure code fragment:

```
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;  
FETCH NEXT FROM cstmt INTO item, sales;
```

Example: Creating an SQL Procedure Using Condition and Iteration Statements

This procedure sets the background color of a rectangle to blue if its area is equal to that of the input rectangle and its background color is not blue. The rectangle and old background color are logged into a table.

```

CREATE PROCEDURE SP1(IN p1 rectangle)
BEGIN
  DECLARE var1 rectangle;           /* UDT local variable */
  DECLARE var2 VARCHAR(20);
  DECLARE var3 INTEGER;
  DECLARE var4 FLOAT;
  DECLARE rect_cursor CURSOR WITHOUT RETURN FOR
    SELECT rect_col, background_color, id
    FROM table1 FOR UPDATE;
  SET var4 = p1.area();              /* UDT expression */
/* DDL */
CREATE TABLE LogTab(
  id          INTEGER,
  Rect        rectangle,
  OldBGColor  VARCHAR(20));
L1: LOOP
  FETCH RectCursor INTO var1, var2, var3;  /* Fetch into UDT var */
  IF (SQLCODE <> 0) THEN
    LEAVE L1;
  END IF;
  IF (var1.area() = var4 AND var2 <> 'Blue') THEN
    BEGIN TRANSACTION;
    INSERT INTO LogTab
      VALUES (:var3, :var1, :var2);
    UPDATE Employee
      SET BackgroundColor = 'Blue'
      WHERE CURRENT OF RectCursor;
    END TRANSACTION;
  END IF;
END LOOP L1;
END;

```

Example: Creating SQL Procedures Using UDTs

The following examples and example excerpts show various applications of UDTs within SQL procedures:

Cursor Fetching:

```

OPEN cursor1;
LSUB1:
LOOP
  FETCH cursor1 INTO predefinedcol, distincttype, structuredtype;
  IF(SQLSTATE <> 0) THEN LEAVE LSUB1;

```

```

    END IF;
    SET rowcount = rowcount +1;
    INSERT INTO resultstable VALUES(:rowcount, :predefinedtype,      :distincttype,
    :structuredtype.attribute1(), 'FETCH');
END LOOP;
CLOSE Cursor1;

```

Declaring a Structured Type:

```

CREATE PROCEDURE sp1()
BEGIN
    DECLARE var1 INTEGER;
    DECARE var2  VARCHAR(300);
    DECLARE structuredtype STATEUDT;
    SET var1 = 92069;
    SET var2 = 'California";
    INSERT INTO T1 VALUES (var1, structuredtype.state(var2));

```

Using a Mutator Method in a SET Expression:

```

UPDATE resultstable SET structuredcol=:structuredtype.attribute1();

```

Create Procedure:

```

CREATE PROCEDURE ups02(IN pi udtint)
BEGIN
    FOR fcode AS b_code CURSOR WITHOUT RETURN FOR
        SELECT udt2 FROM tab2 WHERE col1 = :pi.method1()
    DO
        UPDATE tab2
        SET col2 = :fcode.udt2.method1()
        WHERE col1 = :pi.method1()
    ELSE
        INSERT INTO tab2(:pi, :fcode.udt2.method1());
    END FOR;
END;

```

Create Procedure:

```

CREATE PROCEDURE cas01 (OUT p1 insv_structured_date)
BEGIN
    DECLARE vudt insv_structured_date AS NEW insv_structured_date();

```



```
SET p1= NEW insv_structured_date(CAST('02/02/02' AS DATE));
END;
```

See *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148 for details about how UDTs interact with procedure control language features.

Example: Creating an SQL Procedure that Specifies Recursion

The following CREATE PROCEDURE definition specifies recursion.

```
CREATE PROCEDURE rqsp1(
  OUT rowcnt INTEGER)
BEGIN
  DECLARE target1 INTEGER DEFAULT 0;
  DECLARE target2 INTEGER DEFAULT 0;
  DECLARE target3 INTEGER DEFAULT 0;
  DECLARE finalcount INTEGER DEFAULT 0;
  DECLARE pscursor SCROLL CURSOR FOR
  WITH RECURSIVE RQ(x,y,depth) AS
  (
    SELECT a1, b1, 0

    FROM t1
    UNION ALL
    SELECT a1, y, depth+1

    FROM RQ,t1

    WHERE y = a1

    AND   depth < 10
  )
  SELECT *

  FROM RQ;
  OPEN pscursor;
  FETCH pscursor INTO target1, target2, target3;
  -- if there is no row from the cursor,
  -- SQLCODE=7632, SQLSTATE=02000
  WHILE (SQLCODE <> 7632)
  DO
    FETCH pscursor INTO target1, target2, target3;
    SET finalcount = finalcount+1;
```

```

        END WHILE;
        CLOSE pscursor;
        SET rowcnt = finalcount;
    END;

```

Example: Invoking an SQL UDF From an SQL Procedure

The following example invokes the SQL UDF *common_value_expression* several times.

```

CREATE PROCEDURE spAccount (
    IN  p2 INTEGER,
    OUT p1 CHARACTER(30))
L1: BEGIN
    DECLARE i INTEGER;
    DECLARE i1 INTEGER;
    DECLARE DeptCursor CURSOR FOR
        SELECT DeptName from Department
        WHERE test.common_value_expression(DeptNo, 0) = 25;
    DECLARE CONTINUE HANDLER FOR SQLSTATE VALUE '23505'
L2: BEGIN
        SET i1 = p2;
        SET p1='Failed To Insert Row';
    END L2;
L3: BEGIN
        INSERT INTO table_1
        VALUES(1, test.common_value_expression(1,2));
        IF SQLCODE <> 0 THEN LEAVE L1;
    END L3;
        INSERT INTO table_2
        VALUES(2, test.common_value_expression(2,3));
    END L1;

```

Example: Converting a CLOB Containing XML Data into Rows for Insertion into a Table

The following example creates a procedure that converts a CLOB value containing XML sales data into a set of rows that are inserted into a *sales* table.

First the table definitions:

```

CREATE TABLE sales (
    partnum    INTEGER,

```

```

    qty sold    INTEGER,
    storecode  INTEGER,
    salesdate  DATE)
PRIMARY INDEX (partnum, qty sold, storecode, salesdate);
CREATE TABLE saleslog (
    storecode  INTEGER,
    salesdate  DATE,
    sales      CLOB,
    logdate    DATE,
    logtime    TIME)
PRIMARY INDEX (storecode, salesdate, logdate, logtime);

```

The following SQL code defines the procedure:

```

CREATE PROCEDURE stores_sales_procedure (storecode INTEGER,
                                         salesdate DATE,
                                         salesclob CLOB)

BEGIN
    INSERT INTO saleslog (:storecode, :salesdate, :salesclob,
                          CURRENT_DATE, CURRENT_TIME);

    INSERT INTO sales
        SELECT * FROM TABLE (xmlparser(:salesclob));
END;

```

The stages in the process followed by this procedure are as follows:

1. The sales CLOB is inserted into a log table named *saleslog*.
2. The CLOB is passed by means of the variable *salesclob* to a user-defined table function named *xmlparser* (see [CREATE FUNCTION and REPLACE FUNCTION \(Table Form\)](#)).
3. The system invokes table function *xmlparser* on every AMP.
4. The table function invokes *fnc_loblocal* (the definitions for *xmlparser* and *fnc_loblocal* are not shown for this example) to detect whether the CLOB is located on the same AMP on which the function is executing.
5. The function that has local access to the CLOB strips out the XML sales data and returns standard relational table rows to be inserted into *sales*.

Example: Creating a Procedure with Parameters and Local Variables

The following example creates a valid SQL procedure named *sp_sample1* with the following parameters and local variables:

Variable Type	Variable Name
IN	ip

Variable Type	Variable Name
OUT	op
local variable	var1

Because the parameter type for *ip* is not specified, it defaults to IN. This procedure does not contain any condition handlers.

```
CREATE PROCEDURE sp_sample1(  ip INTEGER,
                             OUT op INTEGER)
BEGIN DECLARE var1 INTEGER;
  SELECT col1 INTO var1
    FROM tab1
    WHERE col2 = ip;
  SET op = var1 * 10;
END;
```

Example: Creating a Procedure that Performs a Consume Mode SELECT

The following example is similar to “Example: Creating a Procedure with Parameters and Local Variables” except that it performs a consume mode SELECT into a queue table. For information about the SELECT AND CONSUME statement, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

```
CREATE PROCEDURE spSampleQ(IN  ip INTEGER,
                           OUT op INTEGER)
BEGIN DECLARE qits    TIMESTAMP(6) DEFAULT CURRENT_TIMESTAMP(6),
             qsn      INTEGER GENERATED ALWAYS AS IDENTITY,
             qcol_1   INTEGER,
             qcol_2   INTEGER;
  SELECT AND CONSUME TOP 1 qits, qsn, qcol_1, qcol_2
    INTO c_qits, c_qsn, c_qcol_1, c_qcol_2
  FROM qtable;
END;
```

Note that you must specify an AND CONSUME TOP 1 phrase in your SELECT statement to consume rows from a queue table. The AND CONSUME keywords indicate that the request is a consume mode request, while TOP 1 indicates that the oldest row from the queue table is to be retrieved.

Example: Creating a Procedure that Consumes a Queue Table

The following procedure consumes a shopping cart queue table row that returns the three variables *sp_ordernum*, *sp_product*, and *sp_quantity*:

```
CREATE PROCEDURE consumecart(OUT sp_ordernum CHARACTER(15),
                             OUT sp_product  CHARACTER(30),
                             OUT sp_quantity INTEGER)
BEGIN
  SELECT AND CONSUME TOP 1 ordernum, product, quantity INTO
    :sp_ordernum, :sp_product, :sp_quantity FROM shoppingcart;
END;
```

The system retrieves the queue table row at the top of the FIFO queue from a single AMP and deletes it from the *shoppingcart* table.

The row was consumed first because it had been in the queue for the longest duration as determined by its QITS value being the oldest of all rows in the queue. The system builds the response row as specified in the expression list and returns it to the requestor in variables.

Related Information

- *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146
- *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148
- [DROP MACRO](#)
- [RENAME MACRO](#)
- [SET QUERY_BAND](#)
- [SHOW object](#)

Also see [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#) and *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about how to create external SQL procedures.

ALTER PROCEDURE (SQL Form)

Recompiles an existing SQL procedure and allows changes in the following compile time attributes of the procedure.

- SPL option
- WARNING option
- AT TIME ZONE option

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

To alter an SQL procedure, you must have either ALTER PROCEDURE or DROP PROCEDURE privilege on that procedure or on the database containing the procedure.

There are no privileges granted automatically.

ALTER PROCEDURE Syntax (SQL Form)

```
ALTER PROCEDURE [ database_name. | user_name. ] procedure_name
  [ LANGUAGE SQL ]
  COMPILE
  [ WITH { [NO] SPL | [NO] WARNING }[,...] ]
  [ AT TIME ZONE { LOCAL | [ sign ] 'quotestring' } ] ] [;]
```

ALTER PROCEDURE Syntax Elements (SQL Form)

database_name

The name of a qualifying database containing the SQL procedure to be altered. If *database_name* is not specified, the default database for the current session is assumed.

user_name

The name of a qualifying database containing the SQL procedure to be altered. If *user_name* is not specified, the default user for the current session is assumed.

procedure_name

The name of the procedure to be recompiled.

The SQL procedure you specify must already exist, must have been created with the SPL compile-time option, and cannot be an external procedure. You can recompile only one SQL procedure at a time.

The following optional specifications are stored in the data dictionary with the SQL procedure definition. You can list the options in any order. Each option can be specified at most once.

LANGUAGE SQL

The procedure body is written in the SQL language. This clause is optional.

Note:

SQL DCL and DDL statements for administering row-level security are not allowed in a stored procedure.

[NO] SPL

The source text of the SQL procedure should [not] be stored in the dictionary. You can alter the SPL option for a procedure to NO SPL.

[NO] WARNING

Compilation warnings are [not] returned during alteration of the SQL procedure.
You can alter the WARNING option for a procedure to NO WARNING.

AT TIME ZONE

Specifies that the specification that follows sets the time zone for the SQL procedure.
You can only specify AT TIME ZONE if you precede it with either the COMPILE option or the COMPILE ONLY option. You cannot specify AT TIME ZONE by itself or following the EXECUTE PROTECTED or EXECUTE NOT PROTECTED options.

LOCAL

Sets the time zone for the procedure to the system default.

[sign] 'quotestring'

Sets the procedure time zone offset to \pm *quotestring*. The default is +.
Vantage implicitly converts the specified expression, as needed and if allowed, to a time zone displacement or time zone string. See *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for a complete list of the implicit data type conversions Vantage performs on \pm *quotestring* data types.

Examples: ALTER PROCEDURE (SQL form)**SQL Procedure for Use in Examples**

The examples in this section refer to the SQL procedure *spAP2*, shown below:

```
CREATE PROCEDURE spAP2(IN InParam INTEGER,
                      OUT OutParam INTEGER)
BEGIN
  DECLARE Var1 INTEGER DEFAULT 10;
  SET OutParam = InParam + Var1 + 1;
END;
```

The examples assume that the SQL procedure has been compiled as follows:

```
BTEQ> .COMPILE FILE = testsp.spAP2
```

The SPL option is not specified, but the source text of the SQL procedure *spAP2* is stored in the database because SPL is the default.

Example: Compiling an SQL Procedure with the AT TIME ZONE Option

This example illustrates the AT TIME ZONE option with an SQL-language procedure named *spa_tz*.

```
ALTER PROCEDURE spa_tz
  COMPILE AT TIME ZONE LOCAL;
```

This option is only valid when it is specified following the COMPILE or COMPILE ONLY options.

For example, the following request is not valid because the AT TIME ZONE specification precedes the COMPILE specification in the request.

```
ALTER PROCEDURE spa_tz
  LANGUAGE SQL
  AT TIME ZONE COMPILE 'gmt';
```

The same request, when the options are specified in the correct sequence, is valid and alters the procedure as requested.

```
ALTER PROCEDURE spa_tz
  LANGUAGE SQL
  COMPILE AT TIME ZONE 'gmt';
```

Example: Impact of Session Mode on an ALTER PROCEDURE Statement

This example illustrates the behavior of ALTER PROCEDURE when the session mode is changed.

The procedure used to do this is as follows:

1. Compile the procedure *spAP3* in a Teradata session mode session.

```
BTEQ> .COMPILE FILE spAP3.sp1
```

2. Change the session mode.

```
.LOGOFF
.SET SESSION TRANS ANSI
.LOGON testsp, password
```

3. Execute ALTER PROCEDURE.

```
ALTER PROCEDURE ap3
  COMPILE;
```

The request fails with the following message.


```
*** Failure 5510 Invalid session mode for procedure execution.
Statement# 1, Info =0
```

Because the execution of ALTER PROCEDURE failed, the existing version of the procedure *spAP3* is retained.

Example: Recompiling an SQL Procedure with an ALTER PROCEDURE Statement, Failure Case

In this example, you execute an ALTER PROCEDURE request on a procedure originally compiled with NO SPL option. ALTER PROCEDURE fails because the source text of the procedure is not available for recompiling. The procedure was originally compiled with NO SPL option; as a result, the source text has not been stored in the database.

1. Compile the procedure *spAP2*.

```
.COMPILE FILE spAP2.sp1 WITH NOSPL
```

2. Verify its attributes.

```
HELP PROCEDURE spAP2 ATTR;
```

Vantage returns the following report.

Transaction Semantics TERADATA	Character Set ASCII
Platform LINUX	Collation ASCII
Default Character DataType UNICODE	Version Number 11
SPL Text N	Print Mode N
Default Database testsp	

3. Perform the ALTER PROCEDURE request.

```
ALTER PROCEDURE spAP2
COMPILE;
```

Vantage returns the following report.

```
ALTER PROCEDURE spAP2
COMPILE;
*** Failure 5535 No SPL source text available for
stored procedure 'spAP2'.
```

Because the execution of the ALTER PROCEDURE request fails, the existing version of the procedure *spAP2* is retained.

Example: Recompiling an SQL Procedure with an ALTER PROCEDURE Statement, Successful Case

This example illustrates the use of ALTER PROCEDURE to recompile an SQL procedure created in Teradata Database 13.0 with changed options. The example shows that the original character set of a procedure is not changed by ALTER PROCEDURE.

1. Verify the current attributes of the procedure using HELP PROCEDURE.

```
HELP PROCEDURE spAP2 ATTR;
```

Vantage returns the following report.

Transaction Semantics TERADATA	Character Set EBCDIC
Platform LINUX	Collation ASCII
Default Character DataType UNICODE	Version Number 09
SPL Text Y	Warning Option Y
Default Database testsp	

The Version Number 09 indicates that the procedure was created in Teradata Database 13.0.

2. Using BTEQ, change the session character set from EBCDIC to ASCII.

```
.SET SESSION charset 'ASCII'
```

3. Execute an ALTER PROCEDURE request that specifies the NO SPL and NO WARNING options.

```
ALTER PROCEDURE spAP2
  COMPILE WITH NO SPL, NO WARNING;
```

4. Check the procedure attributes again, after completion of the alteration.

Transaction Semantics TERADATA	Character Set EBCDIC
Platform LINUX	Collation ASCII
Default Character DataType UNICODE	Version Number 11
SPL Text N	Warning Option N
Default Database testsp	

Note that the compile time options are changed as specified, but the character set of the procedure has not changed. Version Number 11 indicates that the procedure has been upgraded to Teradata Database 14.0.

Example: Suppressing Compilation Warnings with ALTER PROCEDURE

This example shows how compilation warnings can be suppressed using the NO WARNING option of the ALTER PROCEDURE statement by using the following procedure.

1. Create a new procedure *spAP3*.

2. Compile the procedure *spAP3*:

```

REPLACE PROCEDURE testsp.spAP3 ()
BEGIN
  DECLARE var1 INTEGER DEFAULT 07;
  SELECT ErrorCode INTO :var1
  FROM dbc.errormsgs
  WHERE ErrorCode = 5526;
  SET var1 = var1 + 1;
END;
BTEQ> .COMPILE FILE spAP3.sp1

```

Vantage returns the following report.

```

*** Procedure has been created 4 Errors/Warnings.
*** Warning: 5527 Stored Procedure Created with Warnings.
*** Total elapsed time was 2 seconds.
.COMPILE FILE spAP3.sp1
      $
*** SQL Warning 5802 Statement is not ANSI.
Warnings reported during compilation
-----
SPL5000:W(L1), W(5802):Statement is not ANSI.
SPL5000:W(L3), W(5802):Statement is not ANSI.
SPL5000:W(L5), W(5802):Statement is not ANSI.
SPL5000:W(L6), W(5802):Statement is not ANSI.
-----

```

3. Recompile the procedure with NO WARNING option.

```

ALTER PROCEDURE spAP3
COMPILE WITH NO WARNING;

```

When this request is successfully performed, the procedure *spAP3* is recompiled, but no compilation warnings are displayed.

ALTER PROCEDURE (External Form)

Recompiles an existing external procedure and allows changes in the following compile time attributes of the procedure.

- Generate a new library for the recompiled procedure or not.
- Toggle the protection mode between protected and unprotected states.
- Change the creation time zone.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

To alter an external procedure, you must have the ALTER EXTERNAL PROCEDURE privilege on that procedure or on the database containing the procedure.

There are no privileges granted automatically.

ALTER PROCEDURE Syntax (External Form)

```
ALTER PROCEDURE [ database_name. | user_name. ] procedure_name
  LANGUAGE { C | CPP | JAVA }
  { COMPILE [ONLY] [ AT TIME ZONE { LOCAL | [ sign ] 'quotestring' } ] |
    EXECUTE [NOT] PROTECTED
  } [;]
```

ALTER PROCEDURE Syntax Elements (External Form)

procedure_name

The name of the external SQL procedure to be altered. You can alter only one SQL procedure at a time.

database_name

The name of a qualifying database containing the external SQL procedure to be altered. If *database_name* is not specified, the default database for the session is assumed.

user_name

The name of a qualifying user containing the external SQL procedure to be altered. If *user_name* is not specified, the default database for the session is assumed.

LANGUAGE

Programming language in which the external procedure is written.

C

The external procedure is written in the C programming language.

CPP

The external procedure is written in the C++ programming language.

JAVA

The external SQL procedure is written in the Java programming language.

COMPILE

The external procedure is to be recompiled.

Vantage recompiles the procedure and generates a new library for it.

For Java external procedures, the JAR file referenced in the EXTERNAL NAME clause is redistributed to all affected nodes on the system. This is useful for the case of a missing JAR file on a given node.

ONLY

Vantage recompiles the procedure but does not generate a new library.

You cannot specify this for Java external procedures. Vantage does not distribute a new dynamic linked library to database nodes.

When you load an external procedure onto another platform of a different type, the system marks it as not valid, and it must be recompiled. If there are many external procedures in one database, it saves time to specify the ONLY option for all recompilations to avoid having to generate and distribute a new library, until the last one is compiled in that database leaving off the ONLY option.

AT TIME ZONE

The specification that follows sets the time zone for the procedure. You can only specify AT TIME ZONE if you precede it with either the COMPILE option or the COMPILE ONLY option.

You cannot specify AT TIME ZONE by itself or following the EXECUTE PROTECTED or EXECUTE NOT PROTECTED options.

LOCAL

Set the time zone for the procedure to the system default.

sign 'quotestring'

Set the procedure time zone offset to \pm 'quotestring'.

The specification of a sign is optional. The default is +.

Vantage implicitly converts the specified expression, as needed and if allowed, to a time zone displacement or time zone string.

See SET TIME ZONE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for a complete list of the implicit data type conversions Vantage performs on \pm 'quotestring' data types.

EXECUTE PROTECTED

Change the execution mode for the specified external procedure from unprotected mode to protected mode.

The EXECUTE PROTECTED option causes the external procedure to execute as a separate process. For details, see ALTER PROCEDURE (External Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

An external SQL procedure linked with CLlv2 can only execute in protected mode.

A Java external SQL procedure can only execute in protected mode.

NOT

Change the execution mode for the specified external procedure from protected mode to unprotected mode. The EXECUTE NOT PROTECTED option causes the external procedure to execute directly on the Vantage.

For details, see ALTER PROCEDURE (External Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

You cannot specify this option for Java external procedures.

Usage Notes

Invocation Restrictions

Valid for external procedures only.

Not valid inside an SQL procedure body.

ALTER PROCEDURE Examples (External Form)

Example: Recompiling an External Procedure

This example recompiles the C-language external procedure named my_xsp.

```
ALTER PROCEDURE my_xsp LANGUAGE C COMPILE;
```

Example: Using the AT TIME ZONE Option with an External Procedure

This example illustrates the AT TIME ZONE option with a C-language external procedure named spa_tz.

```
ALTER PROCEDURE spa_tz
  COMPILE AT TIME ZONE LOCAL;
```

The AT TIME ZONE option is only valid when you specify it following either the COMPILE option or the COMPILE ONLY option.

For example, the following request is not valid because the AT TIME ZONE specification precedes the COMPILE specification in the request.

```
ALTER PROCEDURE spa_tz
  LANGUAGE C
  AT TIME ZONE COMPILE 'gmt';
```

The same request, when the options are specified in the correct sequence, is valid and alters the procedure as requested.

```
ALTER PROCEDURE spa_tz
  LANGUAGE C
  COMPILE AT TIME ZONE 'gmt';
```

Example: Altering the Protection Mode of an External Procedure

This example changes the protection mode of the C-language external procedure named my_xsp from protected to not protected.

```
ALTER PROCEDURE my_xsp LANGUAGE C EXECUTE NOT PROTECTED;
```

RENAME PROCEDURE

Renames an existing SQL procedure.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have DROP privileges on the procedure to be renamed and the appropriate CREATE privileges on its containing database or user.

RENAME PROCEDURE Syntax

```
RENAME PROCEDURE [ database_name. | user_name. ] old_procedure_name
  { TO | AS } [ database_name. | user_name. ] new_procedure_name [;]
```

RENAME PROCEDURE Syntax Elements

old_procedure_name

Existing name for the procedure.

database_name

Optional name of the containing database for the procedure to be renamed if other than the current database.

You cannot use this statement to change the database qualifier for the procedure.

user_name

Optional name of the containing user for the procedure to be renamed if other than the current user.

You cannot use this statement to change the user qualifier for the procedure.

new_procedure_name

New name for the procedure.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, method, or Java stored procedure object even if the new object name contains only single-byte characters. If you attempt to do so, Vantage aborts the request and returns an error to the requestor. Instead, use a multibyte session character set.

database_name

Optional name of the containing database for the renamed SQL procedure if other than the current database.

user_name

Optional name of the containing user for the renamed SQL procedure if other than the current user.

Example: Renaming a Procedure

The following request renames the *get_region_xsp* procedure to *emp*.

```
RENAME PROCEDURE get_region_xsp TO emp;
```

Related Information

- [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#)
- [CREATE PROCEDURE and REPLACE PROCEDURE \(SQL Form\)](#)
- [DROP PROCEDURE](#)

DROP PROCEDURE

Drops the definition for the specified procedure from the Data Dictionary and from the containing database or user.

When you drop a procedure, the system frees the disk space used by the dropped procedure and any fallback copy, removes any explicit access privileges on the object, and removes the metadata for the dropped object from the data dictionary.

ANSI Compliance

DROP PROCEDURE is ANSI SQL:2011-compliant.

Required Privileges

You must have the appropriate DROP privilege on the specified procedure.

DROP PROCEDURE Syntax

```
DROP PROCEDURE [ database_name. | user_name. ] procedure_name [;]
```

DROP PROCEDURE Syntax Elements

database_name

Name of the containing database for the procedure to be dropped.

This specification is required only if the procedure to be dropped is contained in a different database than the current database.

user_name

Name of the containing user for the procedure to be dropped.

This specification is required only if the procedure to be dropped is contained in a different database than the current database.

procedure_name

Name of the procedure to drop.

Example: Dropping a Procedure

The following request drops a procedure named *get_region_xsp*.

```
DROP PROCEDURE get_region_xsp;
```

Related Information

- *Teradata Vantage™ - SQL External Routine Programming*, B035-1147

HELP PROCEDURE

Displays the attribute and format parameters for each parameter of a procedure or just the creation time attributes for the specified procedure.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have one of the following access privileges to perform HELP PROCEDURE:

- At least one procedure-specific privilege on the database containing the procedure.
- At least one privilege on the specified procedure.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against a specified procedure.

HELP PROCEDURE Syntax

```
HELP PROCEDURE [ database_name. ] procedure_name [ ATTRIBUTES | ATTR | ATTRS ] [;]
```

HELP PROCEDURE Syntax Elements

database_name

Name of the containing database for the procedure if different from the current database.

If a database is not specified, the current default database is assumed.

user_name

Name of the containing user for the procedure if different from the current user.

If a user is not specified, the current default database is assumed.

procedure_name

Name of the procedure for which to display the attributes and format of each parameter.

ATTRIBUTES

ATTRS

ATTR

Keyword indicating that the procedure creation-time attributes from the column SPObjCodeRows in DBC.TVM are to be displayed rather than the attribute and format information for the procedure parameters.

Examples

Example: Reporting UDT Parameters

The following example shows a possible report returned by HELP PROCEDURE when the data type of a parameter in the specified procedure is a UDT.

```
HELP PROCEDURE udtblobset;
  Parameter Name b
                Type UT
                Comment ?
                Nullable Y
                Format ?
                Title ?
                Max Length ?
  Decimal Total Digits ?
  Decimal Fractional Digits ?
                Range Low ?
                Range High ?
                UpperCase N
                Table/View? P
                Default Value ?
                Char Type ?
                Parameter Type 0
```

```

IdCol Type ?
UDT Name CRPV_BLOB

```

If the procedure had been defined with an input parameter with the `VARIANT_TYPE` data type, then the UDT Name in the output would have been reported as `VARIANT_TYPE`.

Example: Reporting ARRAY Parameters for a Procedure

The following example shows a possible report returned by `HELP PROCEDURE` when the data type of a parameter in the specified procedure is an `ARRAY`. Suppose you create a one-dimensional `ARRAY` named `structudt_ary`.

```

Parameter Name b
Type A1
Comment ?
Nullable Y
Format ?
Title ?
Max Length ?
Decimal Total Digits ?
Decimal Fractional Digits ?
Range Low ?
Range High ?
UpperCase N
Table/View? P
Default Value ?
Char Type ?
Parameter Type 0
IdCol Type ?
UDT Name structudt_ary

```

Example: HELP PROCEDURE Attributes Report

You have created the procedure named `sp1new` using the time zone string `AMERICA PACIFIC`. The `HELP PROCEDURE` output for `sp1new` looks like this.

```

HELP PROCEDURE pa.sp1new ATTRIBUTES;
Transaction Semantics TERADATA
Print Mode N
Platform LINUX
Character Set ASCII
Default Character Data Type LATIN

```

```

Collation ASCII
SPL Text Y
Version Number 09
Default Database PA
Warning Option Y
Creation Timezone -07:00
Creation Timezone String AMERICA PACIFIC
Unicode Pass Through S

```

Suppose you then alter procedure *sp1new*, recompiling it at time zone GMT '01:00'.

```
ALTER PROCEDURE pa.sp1new COMPILE AT TIME ZONE '01:00';
```

The HELP PROCEDURE output for *sp1new* now looks like this. Because you specified the time zone using GMT terminology rather than a time zone string, Vantage does not report a time zone string.

```

HELP PROCEDURE pa.sp1new ATTRIBUTES;
Transaction Semantics TERADATA
Print Mode N
Platform LINUX
Character Set ASCII
Default Character Data Type LATIN
Collation ASCII
SPL Y
Version Number 09
Default Database PA
Warning Option Y
Creation Timezone 01:00
Creation Timezone String
Unicode Pass Through S

```

Example: HELP PROCEDURE with Unicode Pass Through Set

Stored procedures inherit the Unicode Pass Through attribute of the session in which they are created:

Setting	Unicode Pass Through
S	On for the session
F	Off for the session.

The following output shows that Unicode Pass Through was set to on for the session that created the procedure.

```
HELP PROCEDURE sp1 ATTRIBUTES;
```

```
*** Help information returned. One row.
```

Transaction Semantics	TERADATA
Print Mode	N
Platform	LINUX
Character Set	ASCII
Default Character DataType	LATIN
Collation	ASCII
SPL Text	Y
Version Number	11
Default Database	USER2
Warning Option	Y
Creation Timezone	00:00
Creation Timezone String	
Unicode Pass Through	S

Related Information

- *Teradata Vantage™ - SQL External Routine Programming, B035-1147*

Macro Statements

CREATE MACRO and REPLACE MACRO

Defines a set of statements that are frequently used or that perform a complex operation. The statements in the macro body are submitted when the macro is invoked by a subsequent EXECUTE statement.

REPLACE MACRO redefines an existing macro. If the specified macro does not exist, REPLACE MACRO creates a new macro with that name.

ANSI Compliance

CREATE MACRO and REPLACE MACRO are Teradata extensions to the ANSI SQL:2011 standard.

Required Privileges: CREATE MACRO

You must have the CREATE MACRO privilege on the containing database or user in which the macro is to be created. The creator of a macro is automatically granted the DROP MACRO and EXECUTE privileges WITH GRANT OPTION on the macro.

The user creating a macro must have the privileges for all statements it performs.

Once a macro has been created, its immediate owner is the database or user in which it is contained, not the user who created it. The immediately owning database or user must have all the appropriate privileges for executing the macro, including WITH GRANT OPTION.

Access to data via a macro is controlled by the privileges of its immediate owner, not by the privileges of its creator. This can be a security issue. See *Teradata Vantage™ - SQL Data Control Language*, B035-1149 for details.

The user who performs a macro need not be aware of the tables affected by its performance.

Required Privileges: REPLACE MACRO

You must have the DROP MACRO privilege on an existing macro or its containing database or user to replace it.

You must have the CREATE MACRO privilege on the macro or its containing database or user if it does not already exist.

The user replacing a macro must have the privileges for all statements it performs.

Once a macro has been replaced, its immediate owner is the database in which it exists, not the user who replaced it. The immediately owning database must have all the appropriate privileges for executing the macro, including WITH GRANT OPTION.

Privileges Granted Automatically

The following privileges are granted automatically to the creator of a macro:

- DROP MACRO
- EXECUTE
- GRANT

CREATE MACRO and REPLACE MACRO Syntax

```
{ CREATE MACRO | CM | REPLACE MACRO } [ database_name. ] macro_name
  [ ( macro_parameter [,...] ) ]
  AS ( [ USING using_modifier ]
      [ LOCKING locking_modifier ][...]
      SQL_statement ;
      ) [;]
```

macro_parameter

```
parameter_name type_declaration [ type_attribute ]...
```

CREATE MACRO and REPLACE MACRO Syntax Elements

database_name

The name of the containing database for *macro_name* if something other than the current database.

macro_name

The name of the new macro. If a fully qualified name is not specified, the default database or user is used.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

The name of a macro must be unique within its containing database.

using_modifier

One or more variable parameter names. A value is substituted for each parameter name when the request is processed. Also see the information about the USING request modifier in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146. This is an optional phrase.

USING does not work in a macro when the macro is accessed using BTEQ. When running BTEQ, specify USING in the EXECUTE request.

Including USING in a macro might require special programming. See your Teradata Field support engineer or the Teradata Support Center for help.

locking modifier

A lock on a database, table, view or row hash.

The specified lock overrides the default usage lock placed in response to a request.

Also see the information about the LOCKING request modifier in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

SQL_statement

Specifies an SQL request. Every request in the macro body must be terminated by a SEMICOLON character.

Parameter names referenced in the macro body must be prefaced by the COLON (:) character. The macro body can include EXECUTE requests to invoke other macros.

Note:

SQL DCL and DDL statements for administration of row level security are not allowed in a macro.

parameter_name

The name of a parameter that is replaced with a value during macro execution. UDT columns and TOP *n* operators are valid parameters.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Parameters are restricted to data values. You cannot parameterize database object names.

When referenced in the macro body, you must prefix a parameter name with the COLON (:) character.

type_declaration

A data definition or default definition for a parameter.

If you do not assign a default value, you must specify a value for the parameter at EXECUTE time.

UDTs are valid data types.

You cannot specify a character server data set of KANJI1 for CHARACTER, VARCHAR, LONGVARCHAR, GRAPHIC, VARGRAPHIC, or CLOB data. If you attempt to do so, Vantage aborts the request and returns an error to the requestor.

For a list of data types see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

type_attribute

The data type attributes, such as NOT NULL, UPPERCASE, or TITLE.

The following column attributes are never valid with macro parameters:

- CHECK constraints
- COMPRESS phrase

For a list of data type attributes see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

CREATE MACRO and REPLACE MACRO Examples

Example: INSERT Operation Followed by a SELECT Verification Operation

The following request creates a macro that first inserts a row for a new employee in the *employee* table, then performs a SELECT request to verify that the information was entered correctly:

```
CREATE MACRO new_employee (
  number    INTEGER,
  name      VARCHAR(12),
  dept      INTEGER    DEFAULT 900,
  position  VARCHAR(12)
  sex       CHARACTER,
  dob       DATE        FORMAT 'MMbDDbYYYY',
  edlev     BYTEINT ) AS
(ININSERT INTO employee (empno,name,deptno,jobtitle,sex,dob,edlev)
VALUES (:number, :name, :dept, :position, :sex, :dob, :edlev);
-- The following select verifies the insert
SELECT *
FROM employee
WHERE empno = :number; );
```

Use the -- comment construct to include comments in the macro. Text appearing after -- and up to the end of the line is not performed.

If this macro is performed in ANSI session mode, the INSERT has not been committed. This is also true in Teradata session mode when the macro is performed as part of an explicit transaction.

Example: INSERT Operation Followed By an UPDATE Operation

This example creates a macro that also inserts a row for a new employee in the *employee* table, then performs an UPDATE request rather than a SELECT request. The UPDATE request changes the *department* table by incrementing the employee count in the row containing a department number that matches the value of the *:dept* parameter.

```
CREATE MACRO new_employee_2 (
  (number    INTEGER,
   name      VARCHAR(12),
   dept      INTEGER    DEFAULT 900,
   position  VARCHAR(12),
   sex       CHARACTER,
   dob       DATE       FORMAT 'MMbDDbYYYY',
   edlev     BYTEINT)  AS
  (INSERT INTO employee (empno,name,deptno,jobtitle,sex,dob,edlev)
   VALUES (:number, :name, :dept, :position, :sex, :dob, :edlev) ;
   UPDATE department
   SET empcount=empcount+1
   WHERE deptno = :dept; );
```

If this macro is performed in ANSI session mode, the INSERT and UPDATE requests have not been committed. This is also true in Teradata session mode when the macro is performed as part of an explicit transaction.

Example: Using REPLACE MACRO

The following statement replaces a macro.

```
REPLACE MACRO new_employee(name VARCHAR(12) NOT NULL,
  street CHARACTER(30),
  city   CHARACTER(20),
  number INTEGER NOT NULL,
  dept   SMALLINT DEFAULT 999) AS
  (INSERT INTO employee (name, street, city, empno, deptno)
   VALUES (:name, :street, :city, :number, :dept);
  UPDATE department
  SET empcount = empcount + 1
  WHERE deptno = :dept ;);
```

Example: Macro Support for UDT Parameters

In this example, the *p2* parameter of the macro is a VARCHAR, which is passed through to the macro. The conversion is performed within the macro itself using the NEW constructor expression. See *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

```
CREATE MACRO m2 (p1 integer, p2 VARCHAR(100))
AS (INSERT t1(:p1, NEW structured_type(:p2)));
USING (a INTEGER, b VARCHAR(100))
EXEC m2(:a, :b);
```

Example: Macro Support for UDT Parameters with Triggers

Suppose you have a macro named *ins_mail_list* that inserts the first name, last name, and home address of new customers into a table named *mailing_addresses*. The value of the customer home address is defined with a structured UDT named *address*.

The macro is defined as follows:

```
CREATE MACRO insert_mail_list (first VARCHAR(15), last VARCHAR(15),
                             addr address)
AS (INSERT INTO mailing_addresses VALUES (:first, :last, :addr));
```

The following trigger adds new California customers to a mailing list whenever a new customer row is inserted into the *customer* table. The insert into the *mailing_addresses* table is done using the triggered SQL action statement, the macro *insert_mail_list*, which accepts the UDT parameter *address*:

```
CREATE TRIGGER ca_mailing_list
AFTER INSERT ON customer
REFERENCING NEW AS newrow
FOR EACH ROW
WHEN (newrow.address.state() = 'CA')
EXEC insert_mail_list(newrow.name.last(), newrow.name.first(),
                     newrow.address);
```

Example: Passing a TOP *n* Value Into a Macro as a Parameter

This example passes the integer value of *n* for the TOP *n* operator into a macro as a parameter named *p*.

```
CREATE MACRO m (p INTEGER)
AS (SELECT TOP :p x1
    FROM t1);
```

Example: Setting the Transaction Query Band Using a Parameter

The following macro sets the transaction query band using the parameter *qbin*:

```
CREATE MACRO qbmac (p1 INTEGER, p2 INTEGER, qbin VARCHAR(200))
AS (SET QUERY_BAND = :qbin FOR TRANSACTION
    SELECT GetQueryBand());
*** Macro has been created.
```

```

*** Total elapsed time was 1 second.
EXEC qbmac (5,10,'music=classical;musician=david_tudor;');
*** Set QUERY_BAND accepted.
*** Total elapsed time was 1 second.
*** Query completed. One row found. One column returned.
GetQueryBand()
-----
=T> music=classical;musician=david_tudor;

```

Example: Invoking an SQL UDF From a Macro Definition

The following example invokes the SQL UDF *common_value_expression* within a SELECT request encapsulated within the definition of the macro *m1*.

```

CREATE MACRO m1 (a1 INTEGER, b1 INTEGER, c1 INTEGER)
AS (SELECT test.common_value_expression(:a1 = :b1)
    FROM t1
    WHERE t1.a1 = :c1; );

```

Example: Macro Support for UDT Parameters using the NEW Constructor Invocation Before Macro Invocation

The following example demonstrates macro support for UDT parameters.

The *p2* parameter of the macro is a structured UDT.

The conversion from VARCHAR to the structured UDT is done using the NEW constructor invocation expression, which is performed before the macro is actually invoked.

```

CREATE MACRO m1 (p1 INTEGER, p2 structured_type)
AS (INSERT t1(:p1, :p2));
USING (a INTEGER, b VARCHAR(100))
EXEC m1(:a, NEW structured_type(:b));

```

Example: Specifying an ABORT or ROLLBACK Condition

You can include a condition for halting execution of a macro by incorporating an ABORT or ROLLBACK request into its definition. If the specified condition is encountered during execution, the macro is aborted. The transaction in process is concluded, locks on the tables are released, changes made to data are backed out, and any spooled output is deleted.

As an example, to restrict the *new_employee* macro from being used to add employees to the Executive Office, department 300, incorporate an ABORT request into the macro specification, as follows:

```

CREATE MACRO personnel.new_employee
(number      (SMALLINT      FORMAT '9(5)'),
name        (VARCHAR(12)),

```

```

dept      (SMALLINT      FORMAT '999'),
position  (VARCHAR(12)),
birthdate (DATE          FORMAT 'MMbDDbYYYY'),
sex       (CHARACTER(1))
education (BYTEINT)) AS
(
(ABORT 'Department 300 not valid'
WHERE :dept = 300;
INSERT INTO employee (empno,name,deptno,jobtitle,dob,sex,edlev)
VALUES (:number,:name,:dept,:position,:birthdate,:sex,:education);    );

```

Specify the text of an optional error message following the ABORT keyword and enclose it in APOSTROPHE characters. This message displays on the terminal screen if the macro is aborted for the specified condition. In this example, the abort condition, `:dept = 300`, is specified in the WHERE clause of the ABORT request.

Example: DELETE With an ABORT Condition

This example shows a DELETE request, and then decrements the employee count for the *department* table. The ABORT request terminates macro execution if the row for the employee being deleted is not present in the *employee* table.

The example shows a macro designed for use in ANSI mode, and for which the user wants to commit if the delete and update operations are successful. Note that the requests in the body of the macro are entered as one multistatement request. Therefore, if the WHERE condition of the ROLLBACK statement is met, the entire request is aborted and the value in *empcount* is protected.

```

CREATE MACRO delete_employee
(num  SMALLINT      FORMAT '9(5)',
dname VARCHAR(12),
dept  SMALLINT      FORMAT '999') AS
(ABORT 'Name does not exist'
WHERE :num NOT IN (SELECT empno
                    FROM employee
                    WHERE name = :dname);

DELETE FROM employee
WHERE name = :dname;

UPDATE department
SET empcount = empcount - 1
WHERE deptno = :dept;
COMMIT WORK; );

```

Related Information

See EXEC in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for information about how to execute a macro.

If an SQL procedure would better meet your needs for a specific application than a macro, see the following references:

- [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#)
- [CREATE PROCEDURE and REPLACE PROCEDURE \(SQL Form\)](#)
- *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148

RENAME MACRO

Renames an existing macro.

An EXCLUSIVE lock is placed on the macro being renamed.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have DROP privileges on the macro to be renamed and the appropriate CREATE privileges on its containing database.

RENAME MACRO Syntax

```
RENAME MACRO [ database_name_1. | user_name_1. ] old_macro_name
  { TO | AS } [ database_name_2. | user_name_2. ] new_macro_name [;]
```

RENAME MACRO Syntax Elements

database_name_1

Optional name of the containing database for the macro to be renamed if other than the current database.

You cannot use this statement to change the database qualifier for the macro.

user_name_1

Optional name of the containing user for the macro to be renamed if other than the current user.

You cannot use this statement to change the user qualifier for the macro.

old_macro_name

Existing name for the macro.

database_name_2

Optional name of the containing database for the renamed macro if other than the current database.

user_name_2

Optional name of the containing user for the renamed macro if other than the current user.

new_macro_name

New name for the macro.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

RENAME MACRO Example

The following request renames the *new_empl* macro as *new_employee*.

```
RENAME MACRO new_empl TO new_employee;
```

Related Information

- [CREATE MACRO and REPLACE MACRO](#)
- [DROP MACRO](#)

DROP MACRO

Drops the definition for the specified macro from the dictionary.

The system frees the disk space used by the dropped macro and any fallback copy, removes any explicit access privileges on the object, and removes the metadata for the dropped object from the data dictionary.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the appropriate DROP privilege on the specified macro.

DROP MACRO Syntax

```
DROP MACRO [ database_name. | user_name. ] macro_name [;]
```

DROP MACRO Syntax Elements

database_name

Name of the containing database for the macro to be dropped.

This specification is required only if the macro to be dropped is contained in a different database than the current database.

user_name

Name of the containing user for the macro to be dropped.

This specification is required only if the macro to be dropped is contained in a different user than the current user.

macro_name

Name of the macro to be dropped.

Related Information

- CREATE MACRO and REPLACE MACRO in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184

HELP MACRO

Displays the attributes for the specified macro.

If a macro does not contain any parameters, a message is returned to indicate that parameters have not been defined.

HELP MACRO can also be used to obtain a comment that explains how to use the macro.

If you perform HELP MACRO for a macro that has semantic errors in its definition, the system does not return an error message.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

To execute this statement, you must have either of the following privileges:

- Ownership of the macro.
- At least one privilege on the requested object.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against a specified macro.

HELP MACRO Syntax

```
HELP MACRO [ database_name. | user_name. ] macro_name [;]
```

HELP MACRO Syntax Elements

database_name

Containing database for *macro_name*, if other than the current database.

user_name

Containing user for *macro_name*, if other than the current user.

macro_name

Macro for which to display help.

Example: HELP MACRO With UDTs

The following example shows HELP MACRO output for a macro that uses UDT parameters.

```
HELP MACRO SYSUDTLB.Example;
      Parameter Name P1
              Type I
              Comment ?
              Nullable N
              Format -(10)9
              Title ?
      Max Length 4
      Decimal Total Digits ?
      Decimal Fractional Digits ?
              Range Low ?
              Range High ?
              Uppercase ?
      Default Value ?
              Char Type ?
              IdCol Type ?
              UDT Name ?
```

```

Parameter Name RETURN0
Type UT
Comment ?
Nullable N
Format ?
Title ?
Max Length ?
Decimal Total Digits ?
Decimal Fractional Digits ?
Range Low ?
Range High ?
Uppercase ?
Default Value ?
Char Type ?
IdCol Type ?
UDT Name ?

```

Example: HELP MACRO

The following request returns information about the *new_emp* macro.

```

HELP MACRO new_emp;
Parameter Name  Type  Comment
-----
Name            CV    Employee name, last name first; required
Number          I     Employee number; required
Dept            I2    Department number; required

```

Related Information

- [CREATE RECURSIVE VIEW and REPLACE RECURSIVE VIEW](#)
- [CREATE TABLE and CREATE TABLE AS](#)
- [CREATE VIEW and REPLACE VIEW](#)
- [SHOW object](#)

User-Defined Function Statements

CREATE FUNCTION and REPLACE FUNCTION (SQL Form)

Creates or replaces an SQL UDF.

ANSI Compliance

CREATE FUNCTION (SQL Form):

This statement is a Teradata extension to the ANSI SQL:2011 standard.

REPLACE FUNCTION (SQL Form):

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

The privileges required to perform CREATE FUNCTION and REPLACE FUNCTION (SQL Form) differ as follows:

- You must have explicit CREATE FUNCTION privileges on the database, including SYSUDTLIB for SQL UDFs associated with UDTs, in which the function is to be contained to perform the CREATE FUNCTION request.

The system does not grant the CREATE FUNCTION privilege automatically when you create a database or user: you must grant the privilege explicitly.

CREATE FUNCTION is not granted implicitly on the databases and functions owned by a database or user unless the owner also has an explicit WITH GRANT OPTION privilege defined for itself.

- You must have explicit DROP FUNCTION privileges on the function or on the database in which the function is contained to perform the REPLACE FUNCTION (SQL Form) statement on an existing function. You do not need the CREATE FUNCTION privilege to replace a function.
- You must have explicit CREATE FUNCTION privileges on the database in which the function is to be contained to perform the REPLACE FUNCTION (SQL Form) statement to create a new function.

If a UDT is specified as an input parameter or the SQL function result, the current user must have one of the following privileges:

- UDTUSAGE on the SYSUDTLIB database.
- UDTUSAGE on the specified UDT.

Privileges Granted Automatically

The following privileges are granted automatically to the creator of an SQL function:

- DROP FUNCTION
- EXECUTE FUNCTION

CREATE FUNCTION and REPLACE FUNCTION Syntax (SQL Form)

```
{ CREATE | REPLACE } FUNCTION [ database_name_1. | user_name_2. ] function_name
  ( parameter_specification [, ...] ) RETURNS return_data_type
  language_and_access_specification
  [ function_attribute [...] ]
  [ SQL SECURITY DEFINER ] COLLATION INVOKER INLINE TYPE 1
  RETURN return_expression [;]
```

Note:

You can specify *language_and_access_specification* and *function_attribute* [...] in the reverse order.

parameter_specification

```
parameter_name parameter_data_type
```

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [( fractional_seconds_precision)] [WITH TIME
ZONE] |

  INTERVAL YEAR [( precision)] [TO MONTH] |

  INTERVAL MONTH [( precision)] |

  INTERVAL DAY [( precision)]
    [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL HOUR [(precision)]
    [TO { MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL MINUTE [(precision)] [ TO SECOND
[(fractional_seconds_precision)] ] |
```

```

INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) |
PERIOD (DATE) |
PERIOD ({ TIME | TIMESTAMP } [(precision)] [ WITH TIME ZONE ]) |
REAL |
DOUBLE PRECISION |
FLOAT [(integer)] |
NUMBER [( { integer | *} [, integer ]...)] |
{ DECIMAL | NUMERIC } [(integer [, integer ]...)] |
{ CHAR | BYTE | GRAPHIC } [(integer)] |
{ VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [(integer)] |
LONG VARCHAR |
LONG VARGRAPHIC |
{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
(integer [ G | K | M ]) |
[SYSUDTLIB.] { XML | XMLTYPE } [(integer [ G | K | M ])] [ INLINE
LENGTH integer ] |
[SYSUDTLIB.] JSON [(integer [ K | M ])] [ INLINE LENGTH integer ]
[ CHARACTER SET { UNICODE | LATIN } ] |
[SYSUDTLIB.] ST_GEOMETRY [(integer [ K | M ])] [ INLINE LENGTH
integer ] |
[SYSUDTLIB.] DATASET [(integer [ K | M ])] [ INLINE LENGTH integer ]
storage_format |
[SYSUDTLIB.] { UDT_name | MBR | ARRAY_name | VARRAY_name }
}

```

language_and_access_specification

```
{ [ language_clause ] SQL_data_access |
  SQL_data_access [ language_clause ]
}
```

function_attribute

```
{ SPECIFIC [ database_name_2. | user_name_2. ] specific_function_name |
  [NOT] DETERMINISTIC |
  CALLED ON NULL INPUT |
  RETURNS NULL ON NULL INPUT
}
```

storage_format

```
STORAGE FORMAT { Avro | CSV [ CHARACTER SET { UNICODE | LATIN } ] }
[ WITH SCHEMA [ database. ] schema_name ]
```

CREATE FUNCTION and REPLACE FUNCTION Syntax Elements (SQL Form)

database_name_1***user_name_1***

An optional database or user name specified if the function is to be created or replaced in a non-default database or user.

If you use the SYSLIB database as your UDF depository, you must modify the size of its permanent space and grant appropriate privileges on it because it is created with 0 permanent space and without access privileges.

Users must have the EXECUTE FUNCTION privilege on any UDF they run or on the database containing the function.

If you do not specify a database name, then the system creates or replaces the function within the current database.

function_name

Calling name for the SQL function.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

This clause is mandatory for all UDFs.

If you use *function_name* to identify the function, take care to follow the identifier naming conventions of the programming language in which it is written.

You cannot give a UDF the same name as an existing Vantage-supplied function (also known as an intrinsic function) unless you enclose the name in QUOTATION MARK characters. For example, “TRIM” (). Using the names of intrinsic functions for UDFs is a poor programming practice and should be avoided.

A UDT and a UDF without parameters that is stored in SYSUDTLIB cannot have the same name.

Parameter data types and number of parameters are used to distinguish among different functions within the same class that have the same *function_name*.

parameter_name

A parenthetical comma-separated list of data types, including UDTs, and parameter names for the variables to be passed to the SQL function. The data types are required to differentiate between functions with the same name.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

The maximum number of parameters an SQL UDF accepts is 128.

You must specify opening and closing parentheses even if no parameters are to be passed to the function.

Parameter names must be unique within an SQL UDF definition.

If you specify one parameter name, then you must specify names for all the parameters passed to the function.

parameter_data_type

The data type associated with each parameter is the type of the parameter or returned value. All Vantage data types except VARIANT_TYPE and TD_ANYTYPE are valid. Character data can also specify a CHARACTER SET clause.

Although you can specify the CHARACTER SET for a parameter, the CHARACTER SET of the caller of the SQL UDF argument is what Vantage uses for SQL UDF processing.

You cannot specify a character server data set of KANJI1. Otherwise, Vantage returns an error to the requestor.

For data types that take a length or size specification, like BYTE, CHARACTER, DECIMAL, VARCHAR, and so on, the size of the parameter indicates the largest number of bytes that can be passed.

return_data_type

Data type for the value returned by the SQL function.

This clause is mandatory for all SQL UDFs.

You cannot specify a character server data set of KANJI1. Otherwise, Vantage returns an error to the requestor.

You cannot specify a RETURNS data type of TD_ANYTYPE for an SQL UDF.

The function is responsible for providing the returned data with the correct type.

The result type has a dictionary entry in *DBC.TVFields* under the name RETURN0[*n*], where *n* is a sequence of digits appended to RETURN0 rows to make each value unique, ensuring that no user-defined parameter names are duplicated. The value of *n* is incremented until it no longer duplicates a parameter name.

The *n* subscript is not used if there is no parameter name of RETURN0.

language_clause

A code that represents the programming language in which the SQL function is written.

The only valid language for writing SQL UDFs is SQL, so you must specify either LANGUAGE SQL or nothing.

The default is LANGUAGE SQL.

SQL_data_access

Whether the SQL function body accesses the database or contains SQL statements.

This is a mandatory attribute for all SQL UDFs and it must be specified as CONTAINS SQL.

database_name_2***user_name_2***

An optional database or user name for the specific function.

specific_function_name

Specific name for the function.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Unlike *function_name*, the specific name for an SQL function, method, or UDT must be unique within its database to avoid name clashes. This name is stored in *DBC.TVM* as the name of the UDF database object.

This clause is mandatory for overloaded *function_names*, but otherwise optional and can only be specified once per SQL function definition.

DETERMINISTIC

The function returns identical results for identical inputs.

DETERMINISTIC can be specified only once per function definition.

NOT DETERMINISTIC

The function might not return identical results for identical inputs.

For example, if the SQL function calls a random number generator as part of its processing, then the results of an SQL function call cannot be known in advance of making the call and the function is NOT DETERMINISTIC.

NOT DETERMINISTIC can be specified only once per function definition.

The default is NOT DETERMINISTIC.

CALLED ON NULL INPUT

Specifies that the SQL function is always evaluated whether parameters are null at the time the function is to be called or not.

This clause is optional and can only be specified once per SQL function definition.

The default is CALLED ON NULL INPUT.

RETURNS NULL ON NULL INPUT

If any parameters are null at the time the SQL function is to be called, a null result is returned without evaluating the function.

This clause is optional and can only be specified once per SQL function definition.

The default is CALLED ON NULL INPUT.

SQL SECURITY DEFINER

Vantage checks the privileges of the creator and owner of the function for objects it references when it is invoked.

If either the immediate owner or the creator do not have the privileges required to access the database objects specified within the UDF definition at the time the function is invoked, Vantage returns an error to the requestor.

If the owner of the function does not exist at the time privileges are checked, Vantage returns an error to the requestor.

SQL SECURITY DEFINER is the default for an SQL UDF and is the only valid entry.

You can only specify this clause if you also specify LANGUAGE SQL.

COLLATION INVOKER

The function uses the default collation setting of the session it is called from rather than the default collation for its creator.

This clause is mandatory.

INLINE TYPE 1

The function is to be executed inline from an SQL request rather than as an independent function.

This clause is mandatory.

The TYPE 1 option indicates that the SQL UDF executes based on the environmental settings of the system. A system that has different settings, might produce different results.

When an SQL UDF is created using the inline option, you can still specify how it should work at the time it is created, depending on the environment in effect for the creator when the function is created and the environment for the caller when it is invoked. If the two environments differ, then the outcome of calling the function can still differ.

return_expression

The SQL statement the SQL UDF is to execute.

You cannot specify a FORMAT attribute for a RETURN expression data type.

The only supported SQL procedure statement is RETURN. See [RETURN Statement](#).

RETURN Statement

Returns a value from an SQL UDF.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

None.

Privileges Granted Automatically

None.

RETURN Statement Syntax

```
RETURN value_expression
```

RETURN Statement Syntax Elements

value_expression

The SQL expression whose value the containing SQL UDF is to return.

The result can also be null.

Usage Notes

- RETURN is an SQL control statement that can only be executed from within an SQL UDF.
- You cannot specify more than one RETURN statement per SQL UDF.
- The data type of *value_expression* must match the data type specified by the RETURNS clause of the SQL UDF definition or it must be capable of being cast implicitly to the data type specified in the RETURNS clause.

The specified data types can be any valid Teradata type.

- *value_expression* cannot contain references to tables.
- *value_expression* cannot contain scalar subqueries such as those made by DML statements.
- *value_expression* can contain references to parameters, constants, SQL UDFs, external UDFs, and methods.
- *value_expression* cannot be a conditional expression or have a Boolean return type.
- *value_expression* can contain arithmetic values and functions, string functions, DateTime functions, and SQL operators that define a scalar result.
- *value_expression* cannot contain aggregate or ordered analytic functions.
- If you specify a reference to an SQL UDF in the RETURN statement, it cannot be any of the following types of reference:

- Self-references
- Circular references
- Forward references

A forward reference is a case where an SQL UDF references another SQL UDF that does not yet exist.

RETURN Statement Examples

Example 1: Simple SQL UDF Definition Using a CASE Expression in its RETURN Statement

The following CREATE FUNCTION request performs an addition operation if the first argument submitted to *calc* is the literal 'A', a subtraction operation if the first argument is 'S', a multiplication operation if the first argument is 'M', and a division operation if the first argument is anything else.

```

CREATE FUNCTION calc (func CHARACTER(1), A INTEGER, B INTEGER)
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN CASE
    WHEN func = 'A'
    THEN A + B
    WHEN func = 'S'
    THEN A - B
    WHEN func = 'M'
    THEN A * B
    ELSE A / B
END;

```

The following SELECT request returns a result value of 5.

```
SELECT calc('A', 2, 3);
```

The following SELECT request returns a result value of 6.

```
SELECT calc('M', 2, 3);
```

Example 2: Defining an SQL UDF With a CASE Expression in its RETURN Statement

Similar to “Example 1: Simple SQL UDF Definition Using a CASE Expression in its RETURN Statement,” this CREATE FUNCTION request specifies a CASE expression in its RETURN statement.

```

CREATE FUNCTION test.case_expr (a INTEGER, b INTEGER)
RETURNS CHARACTER(50)
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
SPECIFIC test.case_expr
CALLED ON NULL INPUT
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN CASE
    WHEN a > 1

```

```

        THEN 'a is greater than 1'
      WHEN b > 1
        THEN 'b is greater than 1'
      ELSE 'a and b are not greater than 1'
    END;

```

Examples

Example: Defining an SQL UDF With a LOB Parameter

The following CREATE FUNCTION request specifies a CLOB as a parameter in its definition.

```

CREATE FUNCTION test.myudf (a CLOB, b INTEGER, c INTEGER,
                           d CHARACTER (100))
  RETURNS CHAR(50)
  LANGUAGE SQL
  DETERMINISTIC
  CONTAINS SQL
  SPECIFIC test.myudf
  CALLED ON NULL INPUT
  SQL SECURITY DEFINER
  COLLATION INVOKER
  INLINE TYPE 1
  RETURN d || (SUBSTRING(a FROM b FOR c));

```

Example: Defining an SQL UDF With a Function in its RETURN Statement

The following CREATE FUNCTION request specifies a CHAR2HEXINT function in its RETURN statement.

```

CREATE FUNCTION test.myudf (a CHARACTER(100))
  RETURNS CHARACTER(400)
  LANGUAGE SQL
  DETERMINISTIC
  CONTAINS SQL
  SPECIFIC test.myudf
  CALLED ON NULL INPUT
  SQL SECURITY DEFINER
  COLLATION INVOKER
  INLINE TYPE 1
  RETURN CHAR2HEXINT(a);

```

Example: Defining an SQL UDF With a DateTime Expression in its RETURN Statement

The following CREATE FUNCTION request specifies a DateTime expression in its RETURN statement.

```
CREATE FUNCTION test.myudf ()
  RETURNS DATE
  LANGUAGE SQL
  DETERMINISTIC
  CONTAINS SQL
  SPECIFIC test.myudf
  CALLED ON NULL INPUT
  SQL SECURITY DEFINER
  COLLATION INVOKER
  INLINE TYPE 1
  RETURN CURRENT_DATE + 1;
```

Example: Defining an SQL UDF With a Period Function in its RETURN Statement

The following CREATE FUNCTION request specifies a bound Period function (END) and a proximity function (PRIOR) in its RETURN statement.

```
CREATE FUNCTION test.myudf (a PERIOD(TIME))
  RETURNS TIME
  LANGUAGE SQL
  DETERMINISTIC
  CONTAINS SQL
  SPECIFIC test.myudf
  CALLED ON NULL INPUT
  SQL SECURITY DEFINER
  COLLATION INVOKER
  INLINE TYPE 1
  RETURN PRIOR(END(a));
```

Example: Defining an SQL UDF With a Method in its RETURN Statement

This example specifies the system-generated method *zip* in its RETURN statement.

First you must create the structured UDT *address*.

```
CREATE TYPE address
AS (street VARCHAR(20), zip CHARACTER(5))
NOT FINAL;
```

Now you can create the SQL UDF *zipcode*, which specifies the system-generated method *zip* in its RETURN statement.

```
CREATE FUNCTION test.zipcode (a address)
RETURNS CHARACTER(5)
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
SPECIFIC test.zipcode
CALLED ON NULL INPUT
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN a.zip();
```

Example: Defining an SQL UDF With an External UDF in its RETURN Statement

The following CREATE FUNCTION request invokes the external UDF *my_external_udf* from its RETURN statement.

```
CREATE FUNCTION test.udf (a INTEGER)
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
SPECIFIC test.udf
CALLED ON NULL INPUT
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN a + SYSLIB.my_external_udf();
```


Example: Defining an SQL UDF With a Geospatial Data Type in Its RETURNS Clause and RETURN Statement

The following CREATE FUNCTION request uses the NEW construct in its RETURN statement and specifies the geospatial data type ST_GEOMETRY in its RETURNS clause and its RETURN statement.

```
CREATE FUNCTION test.geospatial_type_UDF()
  RETURNS ST_GEOMETRY
  LANGUAGE SQL
  DETERMINISTIC
  CONTAINS SQL
  SPECIFIC test.geospatial_type_UDF
  CALLED ON NULL INPUT
  SQL SECURITY DEFINER
  COLLATION INVOKER
  INLINE TYPE 1
  RETURN NEW ST_GEOMETRY('POINT (1,2)');
```

Example: Creating a Function with Geospatial Input Arguments

The following CREATE FUNCTION request has input parameters of the ST_GEOMETRY type, specifies the method *st_within* in its RETURN statement, and returns an INTEGER value from the invocation of *st_within*.

```
CREATE FUNCTION test.geospatial_type_udf
  (a ST_GEOMETRY, b ST_GEOMETRY)
  RETURNS INTEGER
  LANGUAGE SQL
  DETERMINISTIC
  CONTAINS SQL
  SPECIFIC test.geospatial_type_UDF
  CALLED ON NULL INPUT
  SQL SECURITY DEFINER
  COLLATION INVOKER
  INLINE TYPE 1
  RETURN a.st_within(b);
```

Example: Invoking an SQL UDF as an Argument to an External UDF

This example invokes the SQL UDF *common_value_expression* as an argument to the external UDF *my_ext_udf*.

```
SELECT my_ext_udf(test.common_value_expression(t1.a1, t1.b1))
FROM t1;
```

Related Information

- [SHOW object](#)

CREATE FUNCTION and REPLACE FUNCTION (Table Form)

Creates a table function definition.

ANSI Compliance

CREATE FUNCTION (Table Form) is compliant with the ANSI SQL:2011 standard.

Required Privileges

You must have explicit CREATE FUNCTION privileges on the database in which the table function is to be contained to perform the CREATE FUNCTION statement.

The system does not grant the CREATE FUNCTION privilege automatically when you create a database or user: you must grant that privilege explicitly.

CREATE FUNCTION is not granted implicitly on the databases and functions owned by a database or user unless the owner also has an explicit WITH GRANT OPTION privilege defined for itself.

Privileges Granted Automatically

The following privileges are granted automatically to the creator of a table function:

- DROP FUNCTION
- EXECUTE FUNCTION

CREATE FUNCTION and REPLACE FUNCTION Syntax (Table Form)

```
{ CREATE | REPLACE } FUNCTION [ database_name_1. | user_name_1. ] function_name
( parameter_specification [...] ) RETURNS TABLE table_specification
language_and_access_specification
```

```

function_attribute [...]
[ USING [ GLOP SET ] GLOP_set_name ]
EXTERNAL
    [ NAME { external_function_name | 'code_specification [delimiter...]'
| 'JAR_ID_specification' } ]
[ PARAMETER STYLE { SQL | JAVA | SQLTABLE } ]
[ EXTERNAL SECURITY { DEFINER [ authorization_name ] | INVOKER } ]
[ EXECUTE MAP = map_name [ COLOCATE USING colocation_name ] ] [;]

```

Note:

You can specify *language_and_access_specification* and *function_attribute* [...] in the reverse order.

parameter_specification

```
{ [ parameter_name ] data_type | , }
```

table_specification

```

{ ( column_specification [,...] ) |
    VARYING { COLUMNS ( maximum_output_columns ) |
        USING FUNCTION [ database_name_2. | user_name_2. ]
        [ function_name ]
    }
}

```

language_and_access_specification

```

{ language_clause SQL_data_access |
  external_data_access
}

```

Note:

You can specify *language_clause* and *SQL_data_access* in the reverse order.

function_attribute

```
{ SPECIFIC [ database_name_3. | user_name_3. ] specific_function_name |
  PARAMETER STYLE { SQL | JAVA } |
  [NOT] DETERMINISTIC |
  CALLED ON NULL INPUT
}
```

code_specification

```
{ F delimiter function_entry_name |
  { S | C } path_specification
}
```

JAR_ID_specification

```
JAR_ID:java_class_name.method_name
  [ ( java_parameter_class [,...] ) returns java_parameter_class ]
```

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [( fractional_seconds_precision)] [WITH TIME
ZONE] |

  INTERVAL YEAR [( precision)] [TO MONTH] |

  INTERVAL MONTH [( precision)] |

  INTERVAL DAY [( precision)]
  [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL HOUR [(precision)]
  [TO { MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL MINUTE [(precision)] [ TO SECOND
[(fractional_seconds_precision)] ] |

  INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) ] |
```

```

PERIOD (DATE) |

PERIOD ({ TIME | TIMESTAMP } [(precision)] [ WITH TIME ZONE ]) |

REAL |

DOUBLE PRECISION |

FLOAT [(integer)] |

NUMBER [( { integer | *} [, integer ]...)] |

{ DECIMAL | NUMERIC } [(integer [, integer ]...)] |

{ CHAR | BYTE | GRAPHIC } [(integer)] |

{ VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [(integer)] |

LONG VARCHAR |

LONG VARGRAPHIC |

{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
(integer [ G | K | M ]) |

[SYSUDTLIB.] { XML | XMLTYPE } [(integer [ G | K | M ])]
[ INLINE LENGTH integer ] |

[SYSUDTLIB.] JSON [(integer [ K | M ])] [ INLINE LENGTH integer ]
[ CHARACTER SET { UNICODE | LATIN } ] |

[SYSUDTLIB.] ST_GEOMETRY [(integer [ K | M ])] [ INLINE LENGTH
integer ] |

[SYSUDTLIB.] DATASET [(integer [ K | M ])]
[ INLINE LENGTH integer ] storage_format |

[SYSUDTLIB.] { UDT_name | MBR | ARRAY_name | VARRAY_name }
}

```

column_specification

```
column_name column_data_type
```

path_specification

```
{ I delimiter name_on_server delimiter include_name |
  L delimiter library_name |
  O delimiter name_on_server delimiter object_name |
  P delimiter package_name |
  S delimiter name_on_server delimiter source_name |
  NS delimiter source_file delimiter include_file
}
```

java_parameter_class

```
{ primitive [] [ [] ] | object [ [] ] }
```

Note:

You must type the colored or bold braces.

storage_format

```
STORAGE FORMAT { Avro | CSV [ CHARACTER SET { UNICODE | LATIN } ] }
[ WITH SCHEMA [ database. ] schema_name ]
```

CREATE FUNCTION and REPLACE FUNCTION Syntax Elements (Table Form)

database_name_1**user_name_1**

Optional database or user name specified if the function is to be created or replaced in a non-default database or user.

If you use the recommended SYSLIB database as your UDF depository, you must modify the size of its permanent space and grant appropriate privileges on it because it is created with 0 permanent space and without access privileges. See *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

Users must have the EXECUTE FUNCTION privilege on any UDF they run from SYSLIB.

If you do not specify a database name, then the system creates or replaces the function within the current database or user.

function_name

Calling name for the function.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

This is a mandatory attribute for all table UDFs.

If you use *function_name* to identify the function, take care to follow the identifier naming conventions of the programming language in which it is written.

You cannot give a table UDF the same name as an existing Vantage-supplied function (also known as an intrinsic function) unless you enclose the name in QUOTATION MARK characters. For example, "TRIM"(). Using the names of intrinsic functions for UDFs is a poor programming practice and should be avoided.

A UDT and a table UDF without parameters that is stored in SYSUDTLIB cannot have the same name.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, method, or Java stored procedure object even if the new object name contains only single-byte characters. Otherwise, Vantage returns an error to the requestor. Instead, use a multibyte session character set.

function_name must match the spelling and case of the C, C++, or Java function name exactly if you do not specify a *specific_function_name* or *external_function_name*. The same suggestion applies to the naming conventions of the Java language for Java table functions. This rule applies only to the definition of the function, not to calling it.

SQL supports function name overloading within the same function class, so *function_name* need not be unique within its class; however, you cannot give the same name to both a scalar and an aggregate function within the same database or user.

Parameter data types and number of parameters are used to distinguish among different functions within the same class that have the same *function_name*.

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for further information about function overloading.

parameter_name

Parameter names must be unique within a table UDF definition.

The maximum number of parameters a UDF accepts is 128.

You must specify opening and closing parentheses even if no parameters are to be passed to the function.

data_type

Vantage supports the system-defined parameter type `VARIANT_TYPE` for input parameters in a table function.

Note the following rules and restrictions for specifying the system-defined parameter type `VARIANT_TYPE` for an input parameter in a table function.

- You can specify `VARIANT_TYPE` as the data type for a parameter whose external routine is written in C or C++.
- You cannot specify `VARIANT_TYPE` as the data type for a parameter whose external routine is written in Java.

Note that while UDFs support a maximum of 128 parameters, each `VARIANT_TYPE` input parameter supports a maximum of another 128 parameters. Because you can declare a maximum of 8 UDF input parameters to have the `VARIANT_TYPE` data type, the actual number of UDF input parameters Vantage supports when you specify the maximum number of `VARIANT_TYPE` parameters is 1,144.

You can specify `TD_ANYTYPE` as the data type for any parameter of a table function. The system-defined data type `TD_ANYTYPE` can assume any system-defined data type. Its attributes are determined when the function is executed.

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for more information about the `TD_ANYTYPE` parameter data type and see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for more information about how to code C and C++ routines to take advantage of `TD_ANYTYPE`.

The following rules and restrictions apply to specifying the system-defined parameter type `TD_ANYTYPE` for an input or result parameter.

- You can specify `TD_ANYTYPE` as the data type for an input parameter in a scalar, aggregate, or table function written in C, C++, or Java.
- You can specify `TD_ANYTYPE` as the data type for a result parameter in a scalar or aggregate function written in C, C++, or Java.
- You *cannot* specify `TD_ANYTYPE` as the data type for a result parameter in a table function.

You cannot specify a character server data set of KANJI1. Otherwise, Vantage returns an error to the requestor.

BLOB and CLOB parameter data types must be represented by a locator, in contrast with RETURN TABLE clause LOB column data types, which must not be represented by locators. For a description of locators, see the information about the USING request modifier in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Vantage does not support in-memory LOB parameters: an AS LOCATOR phrase must be specified for each LOB parameter.

Note, however, that whenever a LOB that requires data type conversion is passed to a table UDF, the LOB must be materialized for the conversion to take place.

If you specify one parameter name, then you must specify names for all the parameters passed to the function.

If you do not specify parameter names, the system assigns unique names to them in the form *P1*, *P2*, ..., *P n*. These names are used in the COMMENT statement, displayed in the report produced by the HELP FUNCTION statement, and appear in the text of error messages. See [COMMENT \(Comment Placing Form\)](#) and [HELP ONLINE](#).

The data type associated with each parameter is the type of the parameter or returned value. All Vantage data types are valid. Character data can also specify a CHARACTER SET clause.

For data types that take a length or size specification, like BYTE, CHARACTER, DECIMAL, VARCHAR, and so on, the size of the parameter indicates the largest number of bytes that can be passed.

column_name

Name of a column in the set of rows to be returned by the function. You must define at least one column name and its data type for any table function definition.

For information about naming database objects, see *SQL Fundamentals*.

The maximum number of columns you can specify per table function is 2,048.

If one of the specified columns has a LOB data type, then you must define at least one other non-LOB column.

The complete set of column names and their accompanying data types defines the structure of the table the function returns when it is called.

column_data_type

A data type for each column name you specify. For a list of data types, see [Data Types Syntax](#).

If the type is one of the CHARACTER family, then you can also specify a CHARACTER SET attribute.

You cannot specify a RETURNS clause data type of TD_ANYTYPE.

You cannot specify a character server data set of KANJI1. Otherwise, Vantage returns an error to the requestor.

If the data type for a returned column is either BLOB or CLOB, then you cannot specify it with an AS LOCATOR phrase. Such a specification returns an error to the requestor.

This is in direct contrast with the specification of LOB parameter data types, which must be specified with an AS LOCATOR phrase.

You cannot specify any other attributes for a column other than its data type and a CHARACTER SET clause for character data types.

maximum_output_columns

Number of output columns to be returned by the function is not known before it is invoked, so limit them to a maximum of *maximum_output_columns* columns.

The upper limit is 2,048 columns per table function.

There is no default value.

database_name_2

user_name_2

Optional database or user specification.

function_name

Accepts one table or table expression as input and produces one table.

Specifying an explicit function name is optional.

language_clause

A code that represents the programming language in which the external function is written:

Code	Meaning
C	The external function is written in C, even if the external function is in object form.
CPP	The external function is written in C++, even if the external function is in object form.
JAVA	The external function is written in Java.
SAS	The external function is written in SAS, which must use the SQLTABLE parameter style.

This is a mandatory attribute for all UDFs.

The valid languages for writing external UDFs are C, C++, and Java.

If the external function object is not written in C, C++, or Java, it must be compatible with C, C++, or Java object code.

See *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

SQL_data_access

Specifies whether the external function body accesses the database or contains SQL statements.

This clause is mandatory for all table UDFs, and it must be specified as NO SQL.

function_attribute**database_name_3****user_name_3**

Optional database or user name.

specific_function_name

Specific name for the function.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, method, or Java stored procedure object even if the new object name contains only single-byte characters. Otherwise, Vantage returns an error to the requestor. Instead, use a multibyte session character set.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

This clause is mandatory for overloaded function names, but is otherwise optional and can only be specified once per function definition.

If you use *specific_function_name* to identify the C or C++ function name, take care to follow the identifier naming conventions of the C or C++ languages. The same suggestion applies to the naming conventions of the Java language for Java functions.

The *specific_function_name* must be unique within its database or user to avoid name clashes, unlike *function_name*.

This name is stored in *DBC.TVM* as the name of the UDF database object.

PARAMETER STYLE

The parameter passing convention to be used when passing parameters to the table function.

The specified parameter style must match the parameter passing convention of the external function.

This clause is optional for SQL table functions and can only be specified once per function definition. It is mandatory for Java table functions.

If you do not specify a parameter style at this point, you can specify one with the external body reference.

You cannot specify parameter styles more than once in the same CREATE/REPLACE FUNCTION request.

You cannot use a table function to enforce row-level security for a security constraint.

For more information about UDF parameter styles, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

SQL

Uses indicator variables to pass arguments.

As a result, you can always pass nulls as inputs and return nulls in results.

SQL is the default parameter style.

JAVA

Mandatory for all Java table functions.

If the Java function must accept null arguments, then the EXTERNAL NAME clause must include the list of parameters and specify data types that map to Java objects.

DETERMINISTIC

Specifies whether the function returns identical results for identical inputs.

This clause is optional and can only be specified once per function definition.

NOT DETERMINISTIC

For example, if the function calls a random number generator as part of its processing, then the results of a function call cannot be known in advance of making the call and the function is NOT DETERMINISTIC.

The default is NOT DETERMINISTIC.

CALLED ON NULL INPUT

The function is always evaluated whether parameters are null at the time the function is to be called or not.

This clause is optional and can only be specified once per function definition.

GLOP_set_name

Name of the GLOP set this table function is associated with.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

The specified GLOP set does not have to exist at the time the table function is created.

You can specify this clause anywhere between the RETURNS clause and the EXTERNAL clause.

EXTERNAL

Introduction to the external function body reference clause.

This clause is mandatory for all table UDFs.

This clause can specify four different things:

- The keyword EXTERNAL only.
- The keywords EXTERNAL NAME plus an external function name (with optional Parameter Style specification).

This is a mandatory attribute for all UDFs.

- The keywords EXTERNAL NAME plus a set of external string literals.
- The keywords EXTERNAL NAME plus a Java JAR ID specification.

external_function_name

Entry point for the function object, up to 30 characters.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Case is significant and must match the C, C++, or Java function name.

external_string_literal

A string that specifies the location of source and object components needed to build the table function.

For more information, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

code_specification

Single character code. Depending on the initial code in the sequence, the string specifies either the table function object name for the UDF or an encoded name or path for the components needed to create the table function.

F

Function object. The string that follows is the entry point name of the C or C++ table function object.

```
F!function_entry_point_name
```

C

Client. The source or object code for the table function is stored on the client.

S

Server. The source or object code for the table function is stored on the server.

path_specification

Following is a list of the path specifications for the external table function. The options may be repeated as many times as needed with the exception of the package option. You can specify the following file types as external string literals.

I

Include file (.h).

```
I!name_on_server!include_name
```

L

Library name for a nonstandard library files needed by the UDF.

```
L!library_name
```

O

Object file.

```
O!name_on_server!object_name
```

P

Package name. You cannot use the package option with any other options except F, the C function name option.

```
P!package_name
```

S

Source file.

```
S!name_on_server!source_name
```

NS

No source file. Source files and include files are not stored in the function table. This option only affects how source code is processed in the creation of a new function and applies to all source code specified in the external string literal.

```
NS!source_file!include_file
```

delimiter

Specify a delimiter character, such as !. You must use the same delimiter throughout the string specification.

name_on_server

Name the file on the server. Include files must have the same name specified in the include statement in the C source, without the extension.

file_pathname

Location (path) and name of the source, include file, object, or library. Because packages and libraries must be preinstalled, you must use the server option (S). Path specifications can use forward slashes (/) or backward slashes (\) regardless of whether the function is being created on a Unix or Windows platform.

JAR_ID_specification

The following variables apply to Java table functions only.

JAR_ID

The registered name of the JAR file associated with this function.

java_class_name

The name of the Java class contained within the JAR that contains the Java method to be executed.

method_name

The name of the method executed when the UDF is executed.

See SQL Fundamentals for the rules for naming database objects.

java_parameter_class***primitive***

A primitive parameter class as one of the following:

- byte

- double
- int
- long
- short

object

An object parameter class definition in the format:

java.pkg.class

EXTERNAL SECURITY

Keywords introducing the external security clause.

This clause is mandatory for external UDFs that perform operating system I/O operations.

If you do not specify an external security clause, but the UDF being defined performs OS I/O, then the results of that I/O are unpredictable. The most likely outcome is crashing the database, and perhaps crashing the entire system.

See [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#) for information about creating authorizations for external routines.

DEFINER

Specifies that the UDF runs in the client user context of the associated security authorization object created for this purpose, which is contained within the same database as the table function.

- If you specify an authorization name, you must define an authorization object with that name before you can invoke the table function.
- If you do not specify an authorization name, you must define a default DEFINER authorization object.

The default authorization object must be defined before a user can run the table function.

Vantage reports a warning if the specified authorization name does not exist at the time the UDF is created, stating that no authorization name exists.

If you then attempt to execute the table function, the request aborts and Vantage returns an error to the requestor.

authorization_name

Optional authorization name.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

The specified authorization object must already be defined or the system reports an error. For further information, see [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#).

INVOKER

Specifies that the table function runs in the OS user context with the associated default authorization object that exists for this purpose.

See [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#).

EXECUTE MAP

Specify a contiguous or sparse map and, optionally, colocation name for table operator execution. The table operator executes only on the AMPs in the map.

You can override this map by specifying the EXECUTE MAP clause in the FROM clause of the SELECT statement that executes the table operator. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

You must have been granted the specified map.

map_name

Name of contiguous or sparse map.

You cannot specify TD_DataDictionaryMap or TD_GlobalMap.

colocation_name

Name for collocating the function on the same AMPs with other functions, tables, join indexes, or hash indexes.

You can only specify this option for a sparse map. For a contiguous map, the *colocation_name* is set to NULL.

If you do not specify a colocation name, the name defaults to *database_function*, where *database* is the name of the database or user followed by an underscore (_) and *function* is the name of the user defined function. If *database* exceeds 63 characters, *database* is truncated to 63 characters. If *function* exceeds 64 characters, *function* is truncated to 64 characters.

Usage Notes

User, Database, or Profile **DEFAULT MAP OVERRIDE ON ERROR** Option

If the map you specify is not valid for any reason, (such as it does not exist, has not been granted to you, or is not in the same secure zone), Vantage either substitutes a default map or returns an error, subject to the values of the DEFAULT MAP settings for your PROFILE, USER, or DATABASE.

See the CREATE USER [DEFAULT MAP](#) option, CREATE DATABASE [DEFAULT MAP](#) option, or CREATE PROFILE [DEFAULT MAP](#) option.

Map Used by the Function (Table Form)

The map is one of the following as listed in order of precedence:

- Map, if specified by the EXECUTE MAP option in the FROM clause of the SELECT statement that executes the table operator. For more information, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- Map, if specified by the EXECUTE MAP option as part of the CREATE FUNCTION statement.
- User's default contiguous map.

Secure Zones and Sparse Maps

For a sparse map, you must be in the same secure zone as the sparse map.

CREATE FUNCTION Examples

Example: Creating a Function from Input Arguments

This example takes variable input arguments based on input from a previous correlated subquery in the FROM clause of a SELECT request, that is not shown. The function definition inputs XML text, which it then parses to extract a customer ID, store number, and the set of items purchased. The output produces one row per item purchased, with the columns being based on existing columns in the table defined in the function definition.

```
CREATE FUNCTION xml_extract( xml_text VARCHAR(64000))
  RETURNS TABLE (cust_id INTEGER,
                  store   INTEGER,
                  item    INTEGER)

LANGUAGE C
NO SQL
EXTERNAL;
```

Example: Creating and Using an XML Function

Assume you have the following table function definition:

```
CREATE FUNCTION xml_extract( xml_text LOCATOR CLOB)
  RETURNS TABLE (cust_id INTEGER,
                  store   INTEGER,
                  item    INTEGER)
```

```
LANGUAGE C
NO SQL
EXTERNAL;
```

This function extracts all items from the CLOB *xml_text* that have been ordered from a particular store by a particular customer by means of a web application. It then produces one result row for each item ordered.

The XML data is already in a database table with the following table definition.

```
CREATE TABLE xml_tbl (
  store_no INTEGER,
  refnum    INTEGER,
  xml_store_text CLOB)
UNIQUE PRIMARY INDEX (store_no, refnum);
```

The assumptions underlie the analysis of the XML data by means of the *xml_extract* table function.

- Table *xml_tbl* contains the following columns.
 - A store number column named *store_no*.
 - A reference number column named *refnum*.
 - An XML text column named *xml_store_text*.
- The *xml_store_text* column contains the XML-formatted text for customers who ordered web-based items from the store.
- Each XML text data column contains information for the customer who placed the order as well as the items that customer ordered.

The purpose of the table function is to extract all items the customer ordered from the XML document. One XML row is created for each order placed by the online web-based system. Because the XML text could consist of several items, a table function is the natural approach to extracting the data, with one row being extracted for each item ordered. If there were 10 items in the XML text, then the table function would return a 10-row table.

The following SELECT request shows a possible application for the *xml_extract* table function:

```
SELECT l.customer_id l.store, l.item,
FROM (SELECT xml_store_text
      FROM xml_tbl AS x
      WHERE store_no = 25), TABLE(xml_extract(x.xml_text_store))
      AS l (cust_id,store,item);
```

This SELECT request produces one row for each item bought by all customers from store 25. Its first derived table produces the XML test field from all rows with a store number of 25. The second derived table is the result of evaluating table function *xml_extract*.

The logical process followed by the SELECT operation is as follows:

1. Create the first derived table from the first subquery in the FROM clause with the table correlation name *x*.
2. Evaluate the table function.

The function has an input argument that references column *x.xml_text_store* from the derived table. The database must invoke the table function repeatedly in a loop for each row produced by table *x*. The loop ends when the table function returns with the “no more data” message.

The process followed by the loop is as follows:

- a. Read a row for table *x* where *store_no* = 25.
- b. Determine whether such a row is found.

IF a matching row is ...	THEN ...
found	continue processing.
not found	stop processing.

- c. Call the table function *xml_extract* (*x.xml_text_store*).
- d. Determine whether there is more data to process.

IF the call ...	THEN ...
does not return with a SQLSTATE code of '02000' (no more data)	continue processing.
returns with a SQLSTATE code of '02000'	process complete.

- e. Write the new row produced by the *xml_extract* table function.
- f. Go to Stage c.

3. Return results to requestor.

Vantage performs the following process when solving this problem:

1. The system passes the *x.xml_text_store* result produced by the derived table SELECT request as input to the table function *xml_extract*.
2. The system invokes the table function repeatedly for the same row.
Each time the table function is called, it produces one row containing the *cust_id*, *store*, and *item* columns.
3. When the table function has exhausted everything from that *xml_text_store* CLOB, it returns with an SQLSTATE of “no data” ('02000').
4. This “no more data” result causes the system to read the next row produced by the first derived table.
5. The system repeats the process for another customer, generating more rows for the next *xml_text_store*.

Example: Constant Reference Table Function

This example shows a constant reference table function equijoined to another table.

Note the following things about this example:

- The table function is sent to all AMPs with the same constant data; therefore, it is a constant mode table function (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147).
- The table function *tudf1* produce all the rows selected.

After that, the WHERE condition selects a subset from those produced. Only the qualifying rows are returned to the requestor.

```
SELECT *
FROM tbl_1, TABLE(tudf1(28.8, 109)) AS tf2
WHERE tbl_1.c1 = tf2.c1;
```

Example: Variable Reference Table Function

This example shows a variable reference table function equijoined to another table.

Note the following things about this example:

- The table function is sent to all AMPs and is passed *tbl.c1* from all rows in *tbl1*; therefore it is a variable mode table function (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147).
- The designer of this table function added a third argument to demonstrate that the function can provide some intelligence by not producing any rows unless the condition “equal” is met. For example, if the input argument *tbl_1.c1* and the row for value *tf2.c1* are not equal, the function does not produce any rows. This code eliminates the need for a subsequent step to filter rows produced by the table function resulting from the WHERE clause condition.

```
SELECT *
FROM tbl_1, TABLE(tudf1(28.8, tbl_1.c1, 'equal')) AS tf2
WHERE tbl_1.c1 = tf2.c1;
```

Example: Variable Reference Table Function Called from a Derived Table

This example shows a variable reference table function (see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147) equijoined to a derived table.

The derived table *dt1* is created first as a subset of *tbl1*, then a column *dt1.c1* for each row is passed to the table function. The resultant rows of the table function are combined with the selected rows of the derived table by means of the WHERE condition.

The relevant function definition is as follows:

```
CREATE FUNCTION tudf1 (FLOAT, INTEGER, INTEGER)
  RETURN TABLE (
    c1 DECIMAL,
    c2 INTEGER,
    c3 INTEGER)
  ... ;
```

The SELECT request that uses this table function is as follows.

```
SELECT dt1.c1, tf2.c2, tf2.c3
FROM (SELECT c1, c2
      FROM tbl1
      WHERE c1 < 500) AS dt1,
      TABLE (tudf1(45.6, 193, dt1.c1)) AS tf2 (nc1, nc2, nc3)
WHERE dt1.c1 = tf2.nc1);
```

Example: Table Function With a UDT Parameter that Returns a Column With a UDT Data Type

This example creates a table UDF with a UDT parameter that returns a distinct UDT column named *udtc4*.

First create a new distinct data type named *TABLEINT*:

```
CREATE TYPE TABLEINT AS INTEGER FINAL;
```

Now create a new table function named *fnc_tbf001udt* that declares the input parameter *p2* with a data type of *TABLEINT*:

```
CREATE FUNCTION fnc_tbf001udt(
  p1 INTEGER,
  p2 TABLEINT)
  RETURNS TABLE (c1    INTEGER,
                  c2    INTEGER,
                  c3    VARCHAR(3),
                  udtc4 TABLEINT)
  LANGUAGE C
  NO SQL
  PARAMETER STYLE SQL
  EXTERNAL NAME 'CS!fnc_tbf001udt!fnc_tbf001udt.c';
```

Now use the table function in a SELECT request to return results in the form of a table:

```
SELECT *
FROM TABLE(fnc_tbf001udt(1, 1)) AS t1
WHERE t1.c2 IN (0,1);
```

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about how to code external functions similar to *fnc_tbf001udt*.

Example: Different LOB Specifications for Parameter and RETURNS TABLE Clauses

The following example indicates the different ways you must specify LOB data types for function parameters and table columns, respectively. Notice how the parameter type specifications require the specification of an AS LOCATOR phrase, while the table column specifications prohibit the use of an AS LOCATOR phrase.

```
CREATE FUNCTION lobtf_concat3 (
  NumRows INTEGER,
  A      BLOB AS LOCATOR,
  B      VARBYTE(64000),
  C      BLOB AS LOCATOR)
RETURNS TABLE (ampnum  INTEGER,
a_out    BLOB(10),
b_out    VARBYTE(10),
c_out    BLOB(10),
myresult BLOB(30))
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
  EXTERNAL NAME 'SS!lobtf_concat3!/home/i18n/hsf/tf/c/
lobtf_concat3.c';
```

Example: Dynamic Result Row Specification

The following example defines a table function named *extract_store_data* that has variable output columns.

The SELECT request following the function definition uses *extract_store_data* to parse the input text string and extract store sales data into the *store_data* table.

```
CREATE FUNCTION extract_store_data (
  text      VARCHAR(32000),
  from_store INTEGER)
```

```

RETURNS TABLE VARYING COLUMNS (10)
SPECIFIC extract_store_data
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
NOT DETERMINISTIC
EXTERNAL NAME 'SS!extract_store_data!extract_store_data.c';
INSERT INTO store_data
SELECT *
FROM (TABLE(extract_store_data('...', 1000)
RETURNS store_data) AS store_sales;

```

Example: Java Table UDF

The following example shows two different ways to create a definition for a table function. The functions behave identically.

Note the void return type and usage of return parameter types for the Java method signature in this example, which is different from that of scalar and aggregate UDFs. The difference is necessary because of the possibility of multiple returned column values from the table UDF.

```

CREATE FUNCTION mytableudf (
    p1 INTEGER )
RETURNS TABLE (c1 INTEGER,
                c2 INTEGER)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.mytableudf';
public static void mytableudf(int i, int[] ret1,
                             int[] ret2) throws SQLException

CREATE FUNCTION mytableudf (
    p1 INTEGER )
RETURNS TABLE (c1 INTEGER,
                c2 INTEGER)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.mytableudf(int,    int[],int[])';
public static void mytableudf(int i, int[] ret1,
                             int[] ret2) throws SQLException

```


Example: Table Operator Function Written in SAS

This table operator function is written in SAS, which uses the SQLTABLE parameter style.

```
CREATE FUNCTION sas_transform (
  RETURNS TABLE VARYING USING FUNCTION
  LANGUAGE SAS
  PARAMETER STYLE SQLTABLE
  DETERMINISTIC
  RETURN 'optional SAS code text'
  EXTERNAL NAME 'sas_transform_udf:sas_transform_UDF.GetSASCodeText';
```

Example: Ordering Input Parameters to a Table Function

The following simple example illustrates the use of the HASH BY and LOCAL ORDER BY clauses for a table UDF. Table function *add2int* takes two integer values as input and returns both of them and their sum.

Query Q1 selects all columns from *add2int*, which requests its input, *dt*, to be hashed by *dt.y1* and value-ordered on each AMP by *dt.x1*. Note that the specified hashing and local ordering might not be relevant to the *add2int* function and is only used for illustration.

The expected outcome of Q1 is that *dt* is first spooled and then hashed by *y1* among the AMPs. On each AMP, the rows are sorted by the value of *x1*. The final hashed and sorted spool is used as the input to *add2int*.

```
CREATE TABLE t1 (
  a1 INTEGER,
  b1 INTEGER);
CREATE TABLE t2 (
  a2 INTEGER,
  b2 INTEGER);
REPLACE FUNCTION add2int (
  a INTEGER,
  b INTEGER)
RETURNS TABLE (
  addend1 INTEGER,
  addend2 INTEGER,
  mysum INTEGER)
SPECIFIC add2int
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
NOT DETERMINISTIC
```

```

    CALLED ON NULL INPUT
    EXTERNAL NAME 'CS!add3int!add2int.c';
/* Query Q1 */
    WITH dt(x1,y1) AS (SELECT a1,b1
                        FROM t1)

    SELECT *
    FROM TABLE (add2int(dt.x1,dt.y1)
    HASH BY y1
    LOCAL ORDER BY x1) AS tf;

```

Example: Using a Table Function to Pass Data Between Two Vantage Systems

If you are running a dual active Vantage system, or if you have a second Vantage system for a development or test machine, there might be times when you would need to pass data between the two platforms. For example, it might be useful to query the dictionary tables from one system so they can be correlated with the other. This example illustrates how you might do that.

The following SELECT request uses a table function named *tdat* that executes a query on a remote Teradata system and then returns the answer set to the requesting system. In this example, the UDF returns all the database names in the dictionary tables of the other system.

```

SELECT *
FROM table(rdg.tdat(2,1,'adw1/rdg,rdg', 'SELECT databasename
FROM          dbc.databasesV')));

```

This SELECT request is passed as a fixed input argument of the table function from the requesting system and is executed on the target system, as is the logon string for that system.

The DDL to create the table function is as follows:

```

CREATE FUNCTION rdg.tdat
  (rowc      INTEGER,
   InLineNum INTEGER,
   logonstr  VARCHAR(50) CHARACTER SET LATIN,
   sqlRqst   VARCHAR(512) CHARACTER SET LATIN)
RETURNS TABLE (
  ampId      INTEGER,
  cnt        INTEGER,
  OutLineNum INTEGER,
  str1       VARCHAR(256) CHARACTER SET LATIN,
  .
  .
  .

```

```

        str20      VARCHAR(256) CHARACTER SET LATIN)
SPECIFIC tdat
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
NOT DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'SS:tdat:/home/rdg/tdat/Tdat.c:SL:cliv2';

```

By creating a view across two Teradata systems you can compare dictionary content across both platforms and compare detailed data such as table space or privileges.

The following view compares the rows that appear in the *DBC.Tables* view of each system.

```

CREATE VIEW allTables AS
  SELECT 'Local System' AS system, databasename, tablename, version,
        tablekind, protectionType, JournalFlag,
CreatorName,          requesttext(VARCHAR(100))
  FROM DBC.tables
UNION
  SELECT 'Remote System', str1 (CHAR(30)), str2 (CHAR(30)),
        str3 (INTEGER), str4 (CHAR(1)), str5 (CHAR(1)),
        str6 (CHAR(2)), str7 (CHAR(30)), str8 (VARCHAR(100))
  FROM table(rdg.tdat(2,1,'adw1/rdg,rdg', 'SELECT databasename,
tablename, version, tablekind, protectionType,
        JournalFlag, CreatorName, requesttext(VARCHAR(100))
  FROM DBC.tables')) T;

```

A sampling of data returned from the remotely performed SELECT query, when ordered by tablename (for easy cross comparison, looks like this:

System	DatabaseName	TableName	Version	TableKind
-----	-----	-----	-----	-----
Remote System	test	a	1	T
Local System	DBC	AccessRights	1	T
Remote System	DBC	AccessRights	1	T
Remote System	DBC	AllSpaceV	1	V
Local System	DBC	AllSpaceV	1	V
Local System	rdg	allamp	1	T
Remote System	test	allamp	1	T

Related Information

- [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#) for information about creating external authorization objects.
- [CREATE FUNCTION \(External Form\) and REPLACE FUNCTION \(External Form\)](#) for usage notes about the UDF clauses that are not fully explained here.
- [CREATE GLOBAL TEMPORARY TRACE TABLE](#).
- [DROP AUTHORIZATION](#)
- HASH BY and LOCAL ORDER BY clauses in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about coding external table functions.

CREATE FUNCTION and REPLACE FUNCTION (External Form)

Compiles and installs an external UDF and creates or replaces the SQL function definition used to invoke that UDF.

Note:

Customers using Vantage delivered as-a-service cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

ANSI Compliance

CREATE FUNCTION is ANSI SQL:2011-compliant with extensions.

REPLACE FUNCTION is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

The privileges required to perform CREATE FUNCTION and REPLACE FUNCTION differ:

- You must have explicit CREATE FUNCTION privileges on the database, including SYSUDTLIB for UDFs associated with UDTs, in which the function is to be contained to perform the CREATE FUNCTION request.

The system does not grant the CREATE FUNCTION privilege automatically when you create a database or user: you must grant that privilege explicitly.

CREATE FUNCTION is not granted implicitly on the databases and functions owned by a database or user unless the owner also has an explicit WITH GRANT OPTION privilege defined for itself.

- You must have explicit DROP FUNCTION privileges on the function or on the database in which the function is contained to perform the REPLACE FUNCTION statement on an existing function. You do not need the CREATE FUNCTION privilege to replace a function.

- You must have explicit CREATE FUNCTION privileges on the database in which the function is to be contained to perform the REPLACE FUNCTION statement to create a new function.
- If you are not the creator of a function, you must have the EXECUTE FUNCTION privilege on any UDF that you run from the SYSLIB database.

The creator of a function is granted EXECUTE FUNCTION on that UDF automatically.

If a UDT is specified as an input parameter or the function result, the current user must have one of the following privileges:

- UDTUSAGE on the SYSUDTLIB database.
- UDTUSAGE on the specified UDT.

If the function is to be used to enforce the security policy for a table row, but the purpose of the function is not to delete, insert, select, or update table rows, you must have the appropriate OVERRIDE DELETE CONSTRAINT, OVERRIDE INSERT CONSTRAINT, OVERRIDE SELECT CONSTRAINT, or OVERRIDE UPDATE CONSTRAINT privilege to execute it. The request that invokes the function must also include the values to be assigned to the constraint columns of the target table rows.

Privileges Granted Automatically

The following privileges are granted automatically to the creator of an external function:

- DROP FUNCTION
- EXECUTE FUNCTION

CREATE FUNCTION and REPLACE FUNCTION Syntax (External Form)

```
{ CREATE | REPLACE } FUNCTION [ database_name_1. | user_name_1. ] function_name
( parameter_specification [...] ) RETURNS return_data_type [ CAST FROM data_type ]
language_and_access_specification
function_attribute [...]
[ USING GLOP SET GLOP_set_name ]
EXTERNAL [ NAME
    { external_function_name | 'code_specification [delimiter...]'
  | 'JAR_ID_specification' } ]
[ PARAMETER STYLE { SQL | TD_GENERAL | JAVA } ]
[ FOR { COMPRESS | DECOMPRESS } ]
[ EXTERNAL SECURITY { DEFINER [ authorization_name ] | INVOKER } ] [;]
```

Note:

You can specify *language_and_access_specification* and *function_attribute* [...] in the reverse order.

parameter_specification

```
{ [ parameter_name ] data_type | , }
```

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [( fractional_seconds_precision )] [WITH TIME
ZONE] |

  INTERVAL YEAR [( precision )] [TO MONTH] |

  INTERVAL MONTH [( precision )] |

  INTERVAL DAY [( precision )]
    [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL HOUR [(precision)]
    [TO { MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL MINUTE [(precision)] [ TO SECOND
[(fractional_seconds_precision)] ] |

  INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) ] |

  PERIOD (DATE) |

  PERIOD ({ TIME | TIMESTAMP } [(precision)] [ WITH TIME ZONE ]) |

  REAL |

  DOUBLE PRECISION |

  FLOAT [(integer)] |

  NUMBER [( { integer | *} [, integer ]... ) ] |

  { DECIMAL | NUMERIC } [(integer [, integer ]... ) ] |

  { CHAR | BYTE | GRAPHIC } [(integer)] |

  { VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [(integer)] |
```

```

LONG VARCHAR |

LONG VARGRAPHIC |

{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
  (integer [ G | K | M ]) |

[SYSUDTLIB.] { XML | XMLTYPE } [(integer [ G | K | M ])]
  [ INLINE LENGTH integer ] |

[SYSUDTLIB.] JSON [(integer [ K | M ])] [ INLINE LENGTH integer ]
  [ CHARACTER SET { UNICODE | LATIN } ] |

[SYSUDTLIB.] ST_GEOMETRY [(integer [ K | M ])] [ INLINE LENGTH
integer ] |

[SYSUDTLIB.] DATASET [(integer [ K | M ])]
  [ INLINE LENGTH integer ] storage_format |

[SYSUDTLIB.] { UDT_name | MBR | ARRAY_name | VARRAY_name }
}

```

language_and_access_specification

```

{ language_clause SQL_data_access |
  external_data_access
}

```

Note:

You can specify *language_clause* and *SQL_data_access* in the reverse order.

function_attribute

```

{ SPECIFIC [ database_name_2. ] specific_function_name |
  CLASS { AGGREGATE | AG } [ ( interim_size ) ] |
  PARAMETER STYLE { SQL | TD_GENERAL | JAVA } |
  [NOT] DETERMINISTIC |
  CALLED ON NULL INPUT |

```

```

    RETURNS NULL ON NULL INPUT
}

```

code_specification

```

{ F delimiter function_entry_name |
  D |
  { S | C } path_specification
}

```

JAR_ID_specification

```

JAR_ID:java_class_name.method_name

```

```

[ ( java_parameter_class [,...] ) returns java_parameter_class ]

```

storage_format

```

STORAGE FORMAT { Avro | CSV [ CHARACTER SET { UNICODE | LATIN } ] }
[ WITH SCHEMA [ database_name. ] schema_name ]

```

path_specification

```

{ I delimiter name_on_server delimiter include_name |
  L delimiter library_name |
  O delimiter name_on_server delimiter object_name |
  P delimiter package_name |
  S delimiter name_on_server delimiter source_name |
  NS delimiter source_file delimiter include_file
}

```

java_parameter_class

```

{ primitive [ ] [ [ ] ] | object [ [ ] ] }

```

Note:

You must type the colored or bold braces.

CREATE FUNCTION and REPLACE FUNCTION Syntax Elements (External Form)

database_name_1

user_name_1

An optional database or user name specified if the function is to be created or replaced in a non-default database or user.

If you use the recommended SYSLIB database as your UDF depository, you must modify the size of its permanent space and grant appropriate privileges on it because it is created with 0 permanent space and without access privileges.

All row-level security policy functions must reside in the SYSLIB database.

An external UDF used as a cast, ordering, or transform routine for a UDT must be created in the SYSUDTLIB database.

If you do not specify a database name, then the system creates or replaces the function within the current database.

function_name

The calling name for the function.

This clause is mandatory for all UDFs.

A function name can be a maximum of 30 characters in length.

If you use *function_name* to identify the function, take care to follow the identifier naming conventions of the programming language in which it is written.

You cannot give a UDF the same name as an existing Vantage-supplied function (also known as an intrinsic function) unless you enclose the name in QUOTATION MARK(U+0022) characters. For example, "TRIM"(). Using the names of intrinsic functions for UDFs is a poor programming practice and should be avoided.

A UDT and a UDF without parameters that is stored in SYSUDTLIB cannot have the same name.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, method, or Java stored procedure object even if the new object name contains only single-byte characters. Otherwise, an error is returned to the requestor. Instead, use a multibyte session character set.

function_name must match the spelling and case of the C, C++, or Java function name exactly if you do not specify a *specific_function_name* or *external_function_name*. This rule applies only to the definition of the function, not to calling it.

With the exception of row-level security policy UDFs and XML domain-specific functions with an input parameter that specifies the TD_VALIST data type, SQL supports function name overloading within the same function class, so *function_name* need not be unique within its class. However, you cannot give the same name to both a scalar and an aggregate function within the same database or user.

Row-level security enforcement UDFs do not support function name overloading. Because of the parameter types required by Vantage for input to the UDF, the same number of parameters and the same data types for the parameters are required for each UDF that executes the security policy for a specific statement action. The only difference that can exist between the parameters for UDFs that execute a security policy is that the parameters of different UDFs can either include or omit null indicators, depending on whether the constraint allows nulls.

Parameter data types and number of parameters are used to distinguish among different functions within the same class that have the same *function_name*.

For further information about function overloading, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

COMPRESS

This UDF is to be used for algorithmic compression of data.

The same UDF cannot be used to compress and decompress data.

DECOMPRESS

Specifies that the UDF is to be used for decompression of data that was algorithmically compressed using the UDF named *compress_UDF_name*.

The same UDF cannot be used to decompress and compress data.

GLOP_set_name

Name of the GLOP set this function is associated with.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

It is not mandatory that the specified GLOP set exist at the time the function is created.

You can specify this clause anywhere between the RETURNS clause and the EXTERNAL clause.

EXTERNAL

Introduction to the mandatory external function body reference clause.

This clause is mandatory for all UDFs.

This clause can specify four different things:

- The keyword **EXTERNAL** only.
- The keywords **EXTERNAL NAME** plus an external function name (with optional Parameter Style specification).

This is a mandatory attribute for all UDFs.

- The keywords **EXTERNAL NAME** plus a set of external string literals.
- The keywords **EXTERNAL NAME** plus a Java JAR ID specification.

external_function_name

Entry point for the function object.

For details about object name length limits, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Case is significant and must match the C, C++, or Java function name.

EXTERNAL SECURITY

Keywords introducing the external security clause.

This clause is recommended for UDFs that perform operating system I/O operations because it permits you to specify a particular OS user under whom the function runs. Otherwise, a protected mode UDF runs under the generic user *tdatuser*.

Also see [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#).

DEFINER

The UDF runs in the client user context of the associated security authorization object created for this purpose, which is contained within the same database as the procedure.

- If you specify an authorization name, you must define an authorization object with that name before you can invoke the procedure.
- If you do not specify an authorization name, you must define a default **DEFINER** authorization object.

The default authorization object must be defined before a user can run the procedure.

Vantage reports a warning if the specified authorization name does not exist at the time the procedure is created, stating that no authorization name exists.

If you then attempt to execute the procedure, the request aborts and the system returns an error to the requestor.

authorization_name

An optional identifier for this DEFINER.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

See [CREATE AUTHORIZATION](#) and [REPLACE AUTHORIZATION](#).

INVOKER

The function runs using the INVOKER authorization associated with the logged on user who is running the function.

See [CREATE AUTHORIZATION](#) and [REPLACE AUTHORIZATION](#).

parameter_specification

Optional parameter names and locators for the variables to be passed to the function. A function that is used to compress or decompress a UDT column can have only one input parameter. The data type of the parameter can be any of the supported UDT data types.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

You must specify opening and closing parentheses even if no parameters are to be passed to the function.

The maximum number of parameters a UDF accepts is 128.

The parameter data type of an algorithmic compression function used to compress a UDT column must match the RETURNS data type of its decompression function exactly. The type of an algorithmic compression function must also match the data type of the UDT column being compressed.

UDFs with a parameter type of UDT must reside in SYSUDTLIB rather than in SYSLIB.

parameter name

Parameter names must be unique within a UDF definition. If you specify one parameter name, then you must specify names for all the parameters passed to the function. You cannot use the keyword SELF to name UDF parameters. If you do not specify parameter names, Vantage assigns unique names to them in the form P1, P2, ..., P *n*. These names are used in the COMMENT statement and displayed in the report produced by the HELP FUNCTION statement, and appear in the text of error messages. See [COMMENT \(Comment Placing Form\)](#) and [HELP FUNCTION](#).

data_type

A parenthetical comma-separated list of data types, including UDTs. The data types are required to differentiate between overloaded functions with the same name.

The data type associated with each parameter is the type of the parameter or returned value. All Vantage data types are valid. For data types that take a length or size specification, like BYTE, CHARACTER, DECIMAL, VARCHAR, and so on, the size of the parameter indicates the largest number of bytes that can be passed. Character data can also specify a CHARACTER SET clause.

Note:

You cannot specify a character server data set of KANJI1. Otherwise, the system returns an error to the requestor.

BLOB and CLOB types must be represented by a locator. For a description of locators, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146. Vantage does not support in-memory LOB parameters: an AS LOCATOR phrase must be specified for each LOB parameter and return value.

Note:

Whenever a LOB that requires data type conversion is passed to a UDF, the LOB must be materialized for the conversion to take place.

return_data_type

The data type for the value returned by the external function.

This clause is mandatory for all UDFs.

You can specify the system-defined parameter type TD_ANYTYPE as the data type for a RETURNS clause. TD_ANYTYPE can assume any system-defined data type. Its attributes are determined when the function is executed. For more information about the TD_ANYTYPE data type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143. For information about writing function code that takes advantage of the TD_ANYTYPE data type, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

The RETURNS data type of an algorithmic function used to compress a UDT column must be VARBYTE(*n*).

The RETURNS data type of an algorithmic function used to decompress a UDT column must match the UDT parameter data type of its compression function and the data type of the UDT column exactly.

The return length *n* of the RETURNS VARBYTE(*n*) data type of an algorithmic decompression function must match the length of the compression function VARBYTE(*n*) parameter exactly.

You cannot specify a character server data set of KANJI1. Otherwise, Vantage returns an error to the requestor.

The function is responsible for providing the returned data with the correct type. If the return type is difficult for the function to create, you should also specify a CAST FROM clause so the system can perform the

appropriate data type conversion. For more information about using CAST expressions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

The result type has a dictionary entry in *DBC.TVFields* under the name RETURN0[*n*], where *n* is a sequence of digits appended to RETURN0 rows to make each value unique, ensuring that no user-defined parameter names are duplicated. The value of *n* is incremented until it no longer duplicates a parameter name.

The RETURNS data type for a row-level security policy UDF has the following possible values.

- If the policy implements INSERT or UPDATE actions, the data type must be the same as that specified by the constraint definition. See [CREATE CONSTRAINT](#).

The value for the parameter is whatever was inserted as a new row or was updated in an existing row.

- If the policy implements DELETE or SELECT actions, the data type for the return parameter must be CHARACTER(1).

The value for the parameter is either T or F.

Do not specify the subscript indicated by *n* if there is no parameter name of RETURN0.

Teradata Unity uses a CLIV2 parcel to send request-specific context information as part of the request to enable Vantage to replace the result of functions referenced in the request with predefined values. Vantage makes this context information available in the RETURN expression for a UDF.

However, UDFs can generate and use their own arbitrary nondeterministic values that Vantage does not have knowledge of. Therefore, Vantage cannot guarantee that SQL referencing such UDFs has a consistent result.

CAST FROM *return_data_type*

The result type returned by the external function that is to be converted to the type specified by the RETURNS clause.

Example:

```
...RETURNS DECIMAL(9,5) CAST FROM FLOAT...
```

Whenever a LOB that requires data type conversion is passed to an external UDF, the LOB must first be materialized for the conversion to take place.

The value for *data_type* can be a UDT.

You cannot specify a character server data set attribute of KANJI1. Otherwise, Vantage returns an error to the requestor.

language_clause

A code that represents the programming language in which the external function is written.

This is a mandatory attribute for all UDFs.

The valid languages for writing external UDFs are C, C++, and Java.

For C and C++ functions, *language_clause* must be specified as LANGUAGE C or LANGUAGE CPP even if the external function is supplied in object form.

If the external function object is not written in C, C++, or Java, it must be compatible with C, C++, or Java object code.

LANGUAGE

Keyword to introduce the programming language.

C

The external UDF is written in C.

CPP

The external UDF is written in C++.

JAVA

The external UDF is written in Java.

Row-level security constraints cannot be written in Java.

SQL_data_access

Specifies whether the external function body accesses the database or contains SQL statements.

This is a mandatory attribute for all external UDFs and it must be specified as NO SQL.

function_attribute

Specific name for the function. This clause is mandatory for overloaded function names, but otherwise is optional and can only be specified once per function definition.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, method, or Java stored procedure object even if the new object name contains only single-byte characters. Otherwise, the system returns an error to the requestor. Instead, use a multibyte session character set.

database_name_2

An optional database name.

specific_function_name

If you use *specific_function_name* to identify the C or C++ function name, take care to follow the identifier naming conventions of the C or C++ languages. The same suggestion applies to the naming conventions of the Java language for Java functions.

Unlike *function_name*, the specific name for a function, method, or UDT must be unique within its database to avoid name clashes. This name is stored in DBC.TVM as the name of the UDF database object.

CLASS

Class of the function being defined.

Do not specify a keyword if the function class is scalar.

All row-level security policy UDFs must be scalar functions.

AGGREGATE**AG**

The function class is aggregate.

Do not specify this clause for scalar functions.

This clause is optional and can only be specified once per function definition.

interim_size

The size of the aggregate cache allocated for an aggregate UDF.

The minimum value is 1 byte.

The maximum value is 64,000 bytes.

The default value is 64 bytes.

PARAMETER STYLE

Parameter passing convention to be used when passing parameters to the function.

The specified parameter style must match the parameter passing convention of the external function.

If you do not specify a parameter style at this point, you can specify one with the external body reference.

You cannot specify parameter styles more than once in the same CREATE/REPLACE FUNCTION request.

This clause is optional and can only be specified once per function definition.

For more information about UDF parameter styles, see CREATE FUNCTION (External Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

SQL

Uses indicator variables to pass arguments.

As a result, you can always pass nulls as inputs and return nulls in results.

SQL is the default parameter style.

TD_GENERAL

Uses parameters to pass arguments.

Can neither be passed nor return nulls.

JAVA

Mandatory for all Java functions.

DETERMINISTIC

Specifies that the function returns identical results for identical inputs.

DETERMINISTIC and NOT DETERMINISTIC are optional and can only be specified once per function definition.

NOT DETERMINISTIC

Specifies that the function does not always return identical results for identical inputs.

For example, if the function calls a random number generator as part of its processing, then the results of a function call cannot be known in advance of making the call and the function is NOT DETERMINISTIC.

The default is NOT DETERMINISTIC.

CALLED ON NULL INPUT

The function is always evaluated whether parameters are null at the time the function is to be called or not.

If the PARAMETER STYLE for the function is TD_GENERAL, nulls generate an exception condition.

This clause is optional and can only be specified once per function definition.

The default is CALLED ON NULL INPUT.

RETURNS NULL ON NULL INPUT

If any parameters are null at the time the function is to be called, a null result is returned without evaluating the function.

You cannot specify this option for aggregate functions.

This clause is optional and can only be specified once per function definition.

The default is CALLED ON NULL INPUT.

code_specification

Single character code. Depending on the initial code in the sequence, the string specifies either the external function object name for the UDF or an encoded name or path for the components needed to create the external function.

F

Function object. The string that follows is the entry point name of the C or C++ external function object.

```
F!function_entry_point_name
```

D

Enables symbolic debugging for the UDF, which shows source code and displays variables by name. Without this option, UDFs can only be debugged at the machine instruction level. You should always specify this option for debugging purposes when UDFs are being tested. This option adds -g to the C compiler command line. See [SET SESSION DEBUG FUNCTION](#) and the information about C/C++ command-line debugging for UDFs in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

The D option applies only to C and C++ code, not to Java UDFs.

Note:

You should not use this option when installing debugged UDFs on production system because this option increases the size of the UDF library.

S

Server. The source or object code for the external function is stored on the server.

C

Client. The source or object code for the external function is stored on the client.

path_specification

Following is a list of the path specifications for the external function. You can repeat options as necessary with the exception of the package option. You can specify the following file types as external string literals.

I

Include file (.h).

```
I!name_on_server!include_name
```

L

Library name for a nonstandard library files needed by the UDF.

```
L!library_name
```

O

Object file.

```
O!name_on_server!object_name
```

P

Package name. You cannot use the package option with any other options except F, the C function name option.

```
P!package_name
```

S

Source file.

```
S!name_on_server!source_name
```

NS

No source file. Source files and include files are not stored in the function table. This option only affects how source code is processed in the creation of a new function and applies to all source code specified in the external string literal.

```
NS!source_file!include_file
```

delimiter

Specify a delimiter character, such as !. You must use the same delimiter throughout the string specification.

name_on_server

Name the file on the server. Include files must have the same name specified in the include statement in the C source, without the extension.

JAR_ID_specification

The following variables apply to Java functions only.

JAR_ID

The registered name of the JAR file associated with this function.

java_class_name

The name of the Java class contained within the JAR that contains the Java method to be executed.

method_name

The name of the method executed when the UDF is executed.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

primitive

A primitive parameter class as one of the following:

- byte
- double
- int
- long
- short

object

An object parameter class definition in the format:

java.pkg.class

Examples

Example: Creating a UDF C Function

This request creates a user-defined C function on a Linux system. The UDF looks for the C source code file named `find_text.c` on the client system in the current directory of the logged on user. The function name must be `find_text`, which is its entry point name.

The example creates a function that searches for a user-specified test pattern within a user-specified character string. The maximum length search string and pattern string the function allows is 500 bytes. The result is a character value of either T or F, representing true (the test pattern is found in the searched string) or false (the test pattern is not found in the searched string).

```
CREATE FUNCTION find_text (
  searched_string VARCHAR(500),
  pattern          VARCHAR(500))
RETURNS CHARACTER
LANGUAGE C
NO SQL
EXTERNAL
PARAMETER STYLE SQL;

*** Function has been created.
*** Warning: 5607 Check output for possible warnings encountered in compiling
XSP/UDF.
*** Total elapsed time was 5 seconds.

Check output for possible compilation warnings.
-----
/usr/bin/cc -Xc -I /tpasw/etc -c -o find_text.o find_text.c
Teradata High Performance C Compiler R3.0c
(c) Copyright 1994-98, Teradata Corporation
(c) Copyright 1987-98, MetaWare Incorporated
/usr/bin/cc -Xc -I /tpasw/etc -c -o pre_another.o pre_another.c
Teradata High Performance C Compiler R3.0c
(c) Copyright 1994-98, Teradata Corporation
(c) Copyright 1987-98, MetaWare Incorporated
/usr/bin/cc -G -Xc -I /tpasw/etc -o libudf_03ee_11.so
find_text.o pre_find_text.o pre_really_long_function_name.o
long_name.o pre_udf_scalar_substr.o substr.o pre_Find_Text.o
pattern.o pre_char2hexint.o char2hexint.o -ludf -lmw -lm
```

The following SELECT request that searches for a specific text description in documents having a date not less than a specified age shows how the function *find_text* might be used.

```

USING (age DATE, look_for VARCHAR(500))
SELECT doc_number, text
FROM documents
WHERE (:age <= doc_copyright
AND   (find_text(text, :look_for) = 'T'));

```

Example: Parsing an XML Document Using a Scalar UDF

One example where scalar UDFs are useful is scanning an XML document and returning specified content, after that document has been stored inside the database. The following example is of an external UDF that uses XPath, which is a set of syntax rules that allow you to navigate an XML document. XPath, which has a function similar to substring, uses path expressions to identify nodes in an XML document.

Depending on your requirements, the XML document could be stored as a CLOB (Character Large Object) or as a VARCHAR column. The former is illustrated in the graphic below, while the following prototype uses the latter.

In this example, the XML document is stored inside Vantage as one VARCHAR column, *XMLOrder*. The base table, *OrderLog*, contains only two columns, *PONum* and the VARCHAR column. Here is the XML document:

```

<?xml version="1.0"?>
<ROOT>
<ORDER>
  <DATE>8/22/2004</DATE>
  <PO_NUMBER>101</PO_NUMBER>
  <BILLTO>Mike</BILLTO>
  <ITEMS>
    <ITEM>
      <PARTNUM>101</PARTNUM>
      <DESC>Partners Conference Ticket</DESC>
      <USPRICE>1200.00</USPRICE>
    </ITEM>
    <ITEM>
      <PARTNUM>147</PARTNUM>
      <DESC>V2R5.1 UDF Programming</DESC>
      <USPRICE>28.95</USPRICE>
    </ITEM>
  </ITEMS>
</ORDER>
</ROOT>

```

The DDL for the orderlog table looks like this:

```
CREATE SET TABLE orderlog, NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT (
  PONum    INTEGER NOT NULL,
  XMLOrder VARCHAR(63,000) )
UNIQUE PRIMARY INDEX ( PONum );
```

The following SELECT request references the *XpathValue* UDF that uses XPath to extract element and attribute content from the XML document. The arguments passed within the SELECT request, such as the BILLTO name, are then used by XPath to search each document in the table. When a document having that specific billing name is identified, then the associated PO number and date are returned as output arguments.

```
SELECT XpathValue(O.xmlOrder, '//ORDER/PO_NUMBER/*') AS PO_Number,
       XpathValue(O.xmlOrder, '//ORDER/DATE/*') AS theDate
FROM OrderLog O
WHERE XpathValue(O.xmlOrder, '//ORDER/BILLTO/*') = 'Mike';
```

And the output of the query that uses *XPathValue* UDF looks like this:

PO_Number	TheDate
101	8/22/2004

Example: Using a UDF to Write a Message to an External Queue

This example creates a C scalar UDF named *write_mq* that writes a message to an external queue. The function calls an MQ access module, identical to the access module a TPump job or other utility might use when processing from a queue.

The SQL is a simple SELECT request that contains nothing in the select list but the scalar UDF and the arguments it expects. When this request is performed, it produces the message, 'Hello World,' which is placed on an MQ queue on the client:

```
SELECT
WriteMQ('queue.manager.1', 'QUEUE1', 'CHANNEL1/TCP/153.64.119.177',
'Hello World');
```

The parameters passed are the queue manager (*qmgr*), queue name (*qnm*), client communication channel (*channel*), and the message itself (*vcmsg*), respectively.

The following is the DDL used to create the function.

```
CREATE FUNCTION write_mq(
  qmgr    VARCHAR(256),
  qnm     VARCHAR(256),
  channel VARCHAR(256),
  vcmmsg  VARCHAR(32000))
RETURNS INTEGER
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'F:emruwmq:SI:cmqc:/usr/include/cmqc.h:SL:mqic
:SL:mqmcs:SS:emruwmq:/home/rmh/projects/emruwmq/emruwmq.c';
```

Example: Retrieving Rows from the Data Dictionary and Writing Them to an External Queue on a Client System

Now consider a somewhat broader use of the UDF defined in [Example: Using a UDF to Write a Message to an External Queue](#). In this example, you retrieve dictionary information from within Vantage and write it to an external MQ queue on a client system.

To do this, you invoke the *write_mq* UDF for each row found in *DBC.Tables* that meets the requirements specified in the SELECT request WHERE clause. The SELECT request concatenates the database and table names, creating a VARCHAR string that becomes the input argument to the UDF, which then writes the concatenated *dbname.tablename* text as a message to the MQ queue.

```
SELECT COUNT(*) AS sent_msgs
FROM
(SELECT write_mq
 ('queue.manager.1','QUEUE1','CHANNEL1/TCP/153.64.119.177',
  Trim(database)||'.'||trim(TableName)) AS c1
 FROM DBC.Tables
 WHERE TableKind = 'T')T;
```

What this example illustrates is the ease of sending an entire result set of an arbitrary SQL request to a queue that is completely outside Vantage.

Example: Creating and Using an EXTERNAL SECURITY Clause in a UDF

Suppose you have defined the following external authorization object:


```
CREATE AUTHORIZATION DEFINER sales
USER 'salesdept'
PASSWORD 'secret';
```

You now create the following C UDF:

```
CREATE FUNCTION sales_collect (
    store_number INTEGER,
    item_no      INTEGER)
RETURNS INTEGER
LANGUAGE C
NO SQL
EXTERNAL 'cs!salecol!salecollector.c'
PARAMETER STYLE SQL
EXTERNAL SECURITY DEFINER sales;
```

This function collects data associated with sales for a given store for a given item number (*item_no*) and returns the number of items sold. One possible, albeit contrived, scenario would have the function communicating with the store via a network interface. The function is created using the DEFINER context. This means that when a user logs onto its database account and executes an SQL request that invokes this function, it uses the logon ID associated with the sales authorization object, not the user that is executing the invoking SQL request. In this example, OS user *salesdept* has access to the external data the function reads to obtain the information it needs.

Example: Creating and Using a UDT Input Parameter Data Type in Scalar UDF Definition

This example creates a C aggregate UDF defined with a UDT input parameter and then uses it to select a UDT column from a table.

First create a new distinct type named *varchar_udt*:

```
CREATE TYPE varchar_udt AS VARCHAR(20000) FINAL;
```

Now create a new function named *udf_agch002002udt* that uses the input parameter *parameter_1* with the distinct UDT data type *varchar_udt*:

```
CREATE FUNCTION udf_agch002002udt (
    parameter_1 varchar_udt)
RETURNS varchar_udt
CLASS AGGREGATE (20000)
LANGUAGE C
NO SQL
```

```
EXTERNAL NAME    'CS!udf_agch002002udt!udf_agch002002udt.c'
PARAMETER STYLE SQL;
```

Suppose you have created the following table:

```
CREATE TABLE aggr_data_table (
  a INTEGER,
  b VARCHARUDT);
```

You then insert one row into *aggr_data_table*:

```
INSERT INTO aggr_data_table VALUES (1, 'george');
```

You can now select the UDT column from *aggr_data_table* using the aggregate UDF *udf_agch002002udt*:

```
SELECT udf_agch002002udt(aggr_data_table.b)
FROM aggr_data_table;
```

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about how to code external functions like *udf_agch002002udt*.

Example: Creating Functions that Algorithmically Compress LOB-Related Data

The CREATE FUNCTION requests in this example are written to algorithmically compress various kinds of LOB-related data.

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about how to write the C, C++, or Java code for the external functions these function definitions reference to compress LOB-related data.

You can find the functions written to decompress the data algorithmically compressed by the functions in these examples in [Example: Creating Functions that Decompress Algorithmically Compressed LOB-Related Data](#).

This example creates a UDF named *clob_compress* to algorithmically compress CLOB columns.

```
CREATE FUNCTION clob_compress (a CLOB AS LOCATOR)
RETURNS BLOB AS LOCATOR
SPECIFIC clob_compress
LANGUAGE C
NO SQL
FOR COMPRESS
PARAMETER STYLE TD_GENERAL
```

```

RETURNS NULL ON NULL INPUT
DETERMINISTIC
EXTERNAL NAME 'cs!clobcompress!clobcompress.c';

```

This example creates a UDF named `d_clob_compress` to algorithmically compress distinct LOB-based UDT columns.

```

CREATE FUNCTION d_clob_compress (a DCLOB)
RETURNS BLOB AS LOCATOR
SPECIFIC d_clob_compress
LANGUAGE C
NO SQL
FOR COMPRESS
PARAMETER STYLE TD_GENERAL RETURNS NULL ON NULL INPUT
DETERMINISTIC
EXTERNAL NAME 'cs!dclobcompress!dclobcompress.c';

```

Example: Creating Functions that Decompress Algorithmically Compressed LOB-Related Data

The functions in these examples are designed to decompress LOB-based data that has been algorithmically compressed by the examples in “Example: Creating Functions that Algorithmically Compress LOB-Related Data.”

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about how to write the C, C++, or Java code for the external functions these function definitions reference to decompress LOB-related data.

This example creates a UDF named `clob_decompress` to decompress CLOB columns that were algorithmically compressed using the function `clob_compress` from [Example: Creating Functions that Algorithmically Compress LOB-Related Data](#).

```

CREATE FUNCTION clob_decompress (a BLOB AS LOCATOR)
RETURNS CLOB AS LOCATOR
SPECIFIC clob_decompress
LANGUAGE C
NO SQL
FOR DECOMPRESS
PARAMETER STYLE TD_GENERAL
RETURNS NULL ON NULL INPUT
DETERMINISTIC
EXTERNAL NAME 'cs!clobdecompress!clobdecompress.c';

```

This example creates a UDF named *d_clob_decompress* to decompress distinct LOB-based UDT columns that were algorithmically compressed using the function *d_clob_compress* from [Example: Creating Functions that Algorithmically Compress LOB-Related Data](#).

```
CREATE FUNCTION d_clob_decompress (a BLOB AS LOCATOR)
RETURNS DCLOB
SPECIFIC d_clob_decompress
LANGUAGE C
NO SQL
FOR DECOMPRESS
PARAMETER STYLE TD_GENERAL RETURNS NULL ON NULL INPUT
DETERMINISTIC
EXTERNAL NAME 'cs!dclodbdecompress!dclodbdecompress.c';
```

This example creates a UDF named *s_clob_decompress* to decompress distinct LOB-based UDT columns that were algorithmically compressed using the function *s_clob_compress* from [Example: Creating Functions that Algorithmically Compress LOB-Related Data](#).

```
CREATE FUNCTION s_clob_decompress (a BLOB AS LOCATOR)
RETURNS SCLOB
SPECIFIC s_clob_decompress
LANGUAGE C
NO SQL
FOR DECOMPRESS
PARAMETER STYLE TD_GENERAL RETURNS NULL ON NULL INPUT
DETERMINISTIC
EXTERNAL NAME 'cs!sclobdecompress!sclobdecompress.c';
```

Example: Creating and Using a VARIANT_TYPE Input Parameter UDT Data Type in a UDF Definition

This example creates a C aggregate UDF defined with an input parameter of type VARIANT_TYPE. First create a new distinct data type named *integer_udt*.

```
CREATE TYPE integer_udt AS INTEGER FINAL;
```

Now create a new aggregate function named *udf_agch002002dynudt* that uses the input parameter *parameter_1* with a dynamic UDT type of VARIANT_TYPE.

```
CREATE FUNCTION udf_agch002002dynudt (
  parameter_1 VARIANT_TYPE)
RETURNS integer_udt CLASS AGGREGATE(4)
```

```
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!udf_agch002002dynudt!udf_agch002002dynudt.c'
PARAMETER STYLE SQL;
```

You can then use `udf_agch002002dynudt` in a SELECT request with the NEW VARIANT_TYPE constructor expression as follows:

```
SELECT udf_agch002002dynudt(NEW VARIANT_TYPE (tbl1.a AS a,
      (tbl1.b + tbl1.c) AS b))
FROM Tb11;
```

This SELECT request creates a dynamic UDT with two attributes named *a* and *b*.

See CREATE FUNCTION in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for an explanation of how writing UDFs using VARIANT_TYPE can reduce or eliminate the overhead of creating multiple UDFs to handle function name overloading.

See *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210 and *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for information about the NEW VARIANT_TYPE constructor expression.

Example: Using the TD_ANYTYPE Parameter and RETURNS Clause Data Type in a UDF Definition

This function accepts the following data types as valid input parameters: CHARACTER, VARCHAR, and CLOB. See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for the C code for the *ascii* UDF.

The function then returns the numeric representation of the first character in the *string_expr* parameter according to the ASCII character encoding scheme.

The RETURNS clause type supported by this function is any of the following types: BYTEINT, SMALLINT, and INTEGER.

You can use this function to determine the numeric representation of CHARACTER, VARCHAR or CLOB data without knowing anything about the server character set for the data. The function can also return data requesting it as any of the following types: BYTEINT, SMALLINT or INTEGER.

```
CREATE FUNCTION ascii (
  string_expr TD_ANYTYPE)
RETURNS TD_ANYTYPE
LANGUAGE C
NO SQL
SPECIFIC ascii
```

```
EXTERNAL NAME 'CS!ascii!ascii.c'
PARAMETER STYLE SQL;
```

Because the TD_ANYTYPE parameter type can accept any supported data type, it is not necessary to create separate functions like *ascii_char_latin()*, *ascii_char_unicode()*, *ascii_varchar_latin()*, *ascii_varchar_unicode()*, *ascii_clob_latin()*, *ascii_clob_unicode()*, and so forth to handle multiple parameter data types.

Example: Creating a Java Scalar Function

The following request creates a Java scalar UDF named *judf_cref012003*:

```
CREATE FUNCTION judf_cref012003 (
    par1 BIGINT)
RETURNS BIGINT
CAST FROM INTEGER
SPECIFIC judf_cref012003
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
NOT DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.mybigint(long)
              returns int';
```

Example: Creating a Java Aggregate Function

The following request creates a Java aggregate UDF named *std_dev*:

```
CREATE FUNCTION std_dev(
    x FLOAT)
RETURNS FLOAT
CLASS AGGREGATE(79)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.std_dev(
              com.teradata.fnc.Phase,
              com.teradata.fnc.Context[],
              double)
              returns java.lang.Double';
```

For aggregate UDFs, you must specify two extra object type parameters as part of the Java method: *com.teradata.fnc.Phase* , which specifies the current aggregate phase, and *com.teradata.fnc.Context*, which enables you to get or set the context during the execution. For details about these two parameters, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

You can store the intermediate result of aggregate UDFs using either a byte-based or an object-based approach. The byte-based approach consumes less memory and provides better performance, but you must encode and decode the byte array stored in the intermediate storage yourself rather than the system doing the encoding and decoding for you automatically. The object-based method is more user-friendly because the encoding and decoding is done by Java object serialization, but it also affects performance negatively.

The object-based approach for Java aggregate functions requires you to specify a number of bytes with the CLASS AGGREGATE phrase whenever the number of bytes required by intermediate aggregate storage exceeds the default value of 64.

The specified CLASS AGGREGATE value is the maximum number of bytes that the intermediate aggregate storage can occupy in memory. Always set the value as small as possible, because its negative affect on the performance of the function is positively correlated with the class aggregate size.

For Java aggregate functions, the intermediate result is stored as an object that is serialized to a byte array. To keep the size of the intermediate storage small, it is a good practice to avoid using inner classes and to choose short names for the class and its data members. You should also calculate the number of bytes that is required to store the intermediate result for the object, then specify that number in the CLASS AGGREGATE clause. The following code fragment shows how to calculate the serialized size of an object in bytes:

```
public static int getSize(Object obj){
    int size=0;
    try{
        ByteArrayOutputStream barr = new ByteArrayOutputStream();
        ObjectOutput s = new ObjectOutputStream(barr);
        s.writeObject(obj);
        s.close();
        size=barr.toByteArray().length;
        System.out.println("obj="+obj+",size="+size);
    }catch(IOException e){
        e.printStackTrace();
    }
    return size;
}
```

Example: Using a One-Dimensional ARRAY in a Parameter Definition

This example uses a one-dimensional ARRAY type with a CHARACTER element type that will be used as the data type for the single parameter *a1* for the function *my_array_udf*.

The first step is to create an appropriate one-dimensional ARRAY type. The first CREATE TYPE request uses Oracle-compatible syntax to define the ARRAY type *phonenumbers_ary*.

```
CREATE TYPE phonenumbers_ary
AS VARRAY(5) OF CHAR(10);
```

The second CREATE TYPE request uses Teradata-ANSI style syntax to define the same type.

```
CREATE TYPE phonenumbers_ary
AS CHAR(10) ARRAY[5];
```

The following CREATE FUNCTION request creates the function *my_array_udf* using an SQL parameter style and uses the one-dimensional ARRAY type *phonenumbers_ary* as the data type of its single parameter, *a1*.

```
CREATE FUNCTION my_array_udf(
    a1 phonenumbers_ary)
RETURNS VARCHAR(100)
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!my_array_udf!my_array_udf.c';
void my_array_udf (
    ARRAY_HANDLE    *ary_handle,
    VARCHAR_LATIN   *result,
    int             *indicator_ary,
    int             *indicator_result,
    char            sqlstate[6],
    SQL_TEXT        extname[129],
    SQL_TEXT        specific_name[129],
    SQL_TEXT        error_message[257])
{
    /* body function */
}
```

The following CREATE FUNCTION request creates the same function, but uses the TD_GENERAL parameter style.


```

CREATE FUNCTION my_array_udf (
    a1 phonenumbers_ary)
RETURNS VARCHAR(100)
NO SQL
PARAMETER STYLE TD_GENERAL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!my_array_udf!my_array_udf.c';
void my_array_udf (
    ARRAY_HANDLE *ary_handle,
    VARCHAR_LATIN *result,
    char          sqlstate[6])
{ /* body function */
}

```

Example: Using a Multidimensional ARRAY in a Parameter Definition

This example uses a multidimensional ARRAY type to define an INTEGER element type.

The first step is to create the an appropriate multidimensional ARRAY type. The first CREATE TYPE request uses Oracle-compatible syntax to define the ARRAY type *phonenumbers_ary*.

```

CREATE TYPE 3d_array
AS VARRAY (1:5)(1:7)(1:20) OF INTEGER;

```

The second CREATE TYPE request uses Teradata-ANSI style syntax to define the same type.

```

CREATE TYPE 3d_array
AS INTEGER ARRAY[1:5][1:7][1:20];

```

The following CREATE FUNCTION request creates the function *array_udf* using an SQL parameter style and uses the multidimensional ARRAY type *3d_array* as the data type of its single parameter, *a1*.

```

CREATE FUNCTION array_udf(
    a1 3d_array)
RETURNS VARCHAR(100)
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!array_udf!array_udf.c!F!my_array_udf';
void my_array_udf (
    ARRAY_HANDLE *ary_handle,

```

```

    VARCHAR_LATIN  *result,
    int            *indicator_ary,
    int            *indicator_result,
    char           sqlstate[6],
    SQL_TEXT       extname[129],
    SQL_TEXT       specific_name[129],
    SQL_TEXT       error_message[257])
{
/* body function */
}

```

The following CREATE FUNCTION request creates the same function, but uses the TD_GENERAL parameter style.

```

CREATE FUNCTION array_udf (
    a1 3d_array)
RETURNS VARCHAR(100)
NO SQL
PARAMETER STYLE TD_GENERAL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!array_udf!array_udf.c!F!my_array_udf';
void my_array_udf (
    ARRAY_HANDLE *ary_handle,
    VARCHAR_LATIN *result,
    char          sqlstate[6])
{ /* body function */
}

```

Example: Creating a Row-Level Security Function

This example creates several row-level security functions to update table constraint column values. Note the following about these functions.

- The parameter name CURRENT_SESSION defines the source data for the parameter as the current session value for the constraint that is associated with the UDF.
- The parameter name INPUT_ROW defines the source data for the parameter as the value in the corresponding constraint column of the row that is the object of the invoking request.
- The data types assigned to these constraints for the various example functions enable you to determine whether the DML operation being performed applies to a classification constraint or a compartment constraint.

The *insert_level* function inserts a row using the level value for the session (as determined from the parameter name CURRENT_SESSION) as input and returns the level to be assigned to the new row.

Because the `CURRENT_SESSION` parameter is defined with a `SMALLINT` data type, you know that the level value to be inserted concerns a single value of a hierarchical classification constraint.

```
CREATE FUNCTION SYSLIB.insert_level (
    CURRENT_SESSION SMALLINT)
RETURNS SMALLINT
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
EXTERNAL NAME 'CS!insertlevel!c:\cctests\insertlevel.c';
```

The *update_level* function updates a row using the level value for the session (as determined from the parameter name `CURRENT_SESSION`) as input and the level value from the target row (as determined from the `INPUT_ROW` parameter). It returns the level to be assigned to the updated row. Because both the `CURRENT_SESSION` and `INPUT_ROW` parameters are defined with a `SMALLINT` data type, the level value to be updated concerns a single value of a hierarchical classification constraint.

```
CREATE FUNCTION SYSLIB.update_level (
    CURRENT_SESSION SMALLINT,
    INPUT_ROW        SMALLINT)
RETURNS SMALLINT
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
EXTERNAL NAME 'CS!updatelevel!c:\cctests\updatelevel.c';
```

The *delete_level* function deletes a row using the level value from the target row (as determined from the parameter name `INPUT_ROW`) as input. Because the `INPUT_ROW` parameter is defined with a `SMALLINT` data type, the row to be deleted concerns a single value of a hierarchical classification constraint.

The associated external function for this UDF returns either the character T to indicate that the specified deletion can be done or the character F to indicate that the specified deletion cannot be done.

```
CREATE FUNCTION SYSLIB.delete_level (
    INPUT_ROW SMALLINT)
RETURNS CHARACTER
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
EXTERNAL NAME 'CS!deletelevel!c:\cctests\deletelevel.c';
```

The *read_level* function selects a row using the level value from the session (as determined from the parameter name `CURRENT_SESSION`) and the level value from the target row (as determined from

the INPUT_ROW parameter) as input. Because both the CURRENT_SESSION parameter and the INPUT_ROW parameter are defined with a SMALLINT data type, the row to be read concerns a single value of a hierarchical classification constraint.

The associated external function for this UDF returns either the character T to indicate that the select operation can be done or the character F to indicate that the select operation cannot be done.

```
CREATE FUNCTION SYSLIB.read_level (
    CURRENT_SESSION SMALLINT,
    INPUT_ROW        SMALLINT)
RETURNS CHARACTER
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
EXTERNAL NAME 'CS!readlevel!c:\cctests\readlevel.c';
```

The function inserts a row using the category value for the session (as determined from the CURRENT_SESSION parameter) as input. Because the CURRENT_SESSION parameter is defined with a BYTE(8) data type, you know that the row to be inserted concerns a non-hierarchical compartment constraint.

The associated external function for this UDF returns the category to be assigned to the new row.

```
CREATE FUNCTION SYSLIB.insert_category (
    CURRENT_SESSION BYTE(8))
RETURNS BYTE(8)
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'CS!insert_category!c:\cctests\insert_category.c';
```

The *update_category* function updates a row using the category value for the session (as determined from the CURRENT_SESSION parameter) and the category value from the target row (as determined from the INPUT_ROW parameter) as input. Because both the CURRENT_SESSION parameter and the INPUT_ROW parameter are defined with the BYTE(8) data type, the row to be updated concerns a non-hierarchical compartment constraint.

The associated external function for this UDF returns the category to be assigned to the updated row.

```
CREATE FUNCTION SYSLIB.UpdateCategory (
    CURRENT_SESSION BYTE(8),
    INPUT_ROW        BYTE(8))
RETURNS BYTE(8)
LANGUAGE C
NO SQL
```

```

PARAMETER STYLE SQL
EXTERNAL NAME 'CS!updatecategory!c:\cctests\updatecategory.c';

```

The function to delete a row has as input the category value from the target row (as determined from the INPUT_ROW parameter) and returns either the character T to indicate that the specified deletion can be done or the letter F to indicate that the specified deletion cannot be done.

```

CREATE FUNCTION SYSLIB.DeleteCategory (
    INPUT_ROW BYTE(8))
RETURNS CHARACTER
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'CS!deletecategory!c:\cctests\deletecategory.c';

```

The function to select a row has as input the category value from the session (as determined from the CURRENT_SESSION parameter) and the category value from the target row (as determined from the INPUT_ROW parameter) and has as output the character T to indicate that the select can be done or the letter F to indicate it cannot be done.

```

CREATE FUNCTION SYSLIB.ReadCategory (
    CURRENT_SESSION BYTE(8),
    INPUT_ROW        BYTE(8))
RETURNS CHARACTER
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'CS!readcategory!c:\cctests\readcategory.c';

```

Example: Creating a Simple Constraint UDF to Enforce No Read Up Row-Level Security

This example creates a simple constraint definition and its associated UDF to implement a simple No Read Up security policy using sensitivity labels.

Note the following about this function:

- The SQL create text for the function defines two parameters to handle the data for the constraint.
 - *UserClearance*
 - *RowClassification*
- The input to the *UserClearance* parameter is data regarding the classification level for the user who wants to read the row.

- The input to the *RowClassification* parameter is data regarding the classification level for the row to be read.
- Both of the parameters the function defines have a SMALLINT data type, which confirms that they relate to a single-level hierarchical classification constraint.

The name of this single-level hierarchical classification constraint is *ReadClassification*.

- The very simple external function compares the values for *UserClearance* and *RowClassification*.
If the value for *UserClearance* is greater than or equal to the value for *RowClassification*, SELECT access to the requested row is granted.
If the value for *UserClearance* is less than the value for *RowClassification*, SELECT access to the requested row is not granted.

See *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100 for information about sensitivity labels and No Read Up, No Write Down security policies.

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about writing external function routines to support row-level security constraints.

```
CREATE FUNCTION SYSLIB.ReadClassification (
    UserClearance    SMALLINT,
    RowClassification SMALLINT)
RETURNS CHARACTER(1)
SPECIFIC SYSLIB.ReadClassification
LANGUAGE C
DETERMINISTIC
NO SQL
EXTERNAL NAME 'cs!ReadClassification!c:\udf_ReadClassification.c'
PARAMETER STYLE TD_GENERAL;
```

The C code for ReadClassification is as follows.

```
#define SQL_TEXT Latin_Text
#include <sys/types.h>
#include "sqltypes_td.h"
void ReadClassification(short int *UserClearance,
                        short int *RowClassification,
                        char *AccessAllowed,
{
    //Enforce no read up policy - user clearance must dominate row classification
    if (*UserClearance >= *RowClassification)
        // SELECT is allowed
        *AccessAllowed = 'T';
    else
        // SELECT is not allowed
```

```

        *AccessAllowed = 'F';
    return;
}

```

Related Information

- [CREATE AUTHORIZATION AND REPLACE AUTHORIZATION](#)
- [CREATE GLOBAL TEMPORARY TRACE TABLE](#)
- [DROP AUTHORIZATION](#)
- [SET SESSION FUNCTION TRACE](#)
- [SHOW object](#)

For guidelines about coding external user-defined functions, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

For detailed information about row-level security and how it is enforced using scalar UDFs to enforce security policies, see *Teradata Vantage™ - Database Administration*, B035-1093.

For information about using the XML data type, see *Teradata Vantage™ - XML Data Type*, B035-1140.

CREATE FUNCTION MAPPING and REPLACE FUNCTION MAPPING

CREATE FUNCTION MAPPING creates a new function mapping with the name that you specify.

REPLACE FUNCTION MAPPING replaces the definition of an existing function mapping or, if the specified function mapping does not exist, creates a new function mapping by that name.

You use a function mapping to specify a simple name for executing a function or table operator within a database, user, or on an external server. The function mapping defines input tables, output tables, and other parameters to use during function execution. See [Example: Defining a Function Mapping](#), [Example: Replacing a Function Mapping Definition](#), and [Example: Function Mapping Definition for a Function Within the Database](#).

To execute the function or table operator, you specify the function mapping in a SELECT statement FROM clause using the table operator syntax. For information about function processing, see the information about the Table operator in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Note:

In the READ_NOS table function, you can use either a function mapping or the AUTHORIZATION clause. Using both is an error.

Required Privileges

For a table operator on an external server, you must have the EXECUTE FUNCTION privilege on the server. For a table operator stored within the database or user, you must have the EXECUTE FUNCTION privilege on the table operator.

For CREATE FUNCTION MAPPING, you must have CREATE FUNCTION privilege on the database or user where you create the function mapping.

For REPLACE FUNCTION MAPPING where you are creating a function mapping, you must have CREATE FUNCTION privilege on the database or user.

For REPLACE FUNCTION MAPPING where you are replacing a function mapping, you must have the DROP FUNCTION privilege on the function mapping or containing database or user.

For a function mapping that includes a scalar subquery (SSQ), you must have the SELECT privilege on the table referenced by the subquery.

Privileges Granted Automatically

When you create or replace a function mapping object, you are granted EXECUTE and DROP FUNCTION privileges on the function mapping object.

CREATE FUNCTION MAPPING and REPLACE FUNCTION MAPPING Syntax

```
{ CREATE | REPLACE } FUNCTION MAPPING [ database_name_1. |
user_name_1. ] function_mapping_name
  FOR [ { database_name_2. | user_name_2. } [ schema_name. ] ] function_name
  [ SERVER [ database_name_3. | user_name_3. ] server_name ]
  [ EXTERNAL SECURITY [ DEFINER | INVOKER ] TRUSTED ] authorization_name
  [ MAP JSON (json_document) ]
  [ STOREDAS ( { 'PARQUET' | 'TEXTFILE' } ) ]
  [ USING table_list ] [;]
```

table_list

```
{ ANY IN TABLE | name [(value)] [ IN TABLE | OUT TABLE ] } [,...]
```

CREATE FUNCTION MAPPING and REPLACE FUNCTION MAPPING Syntax Elements

database_name_1

Name of database to contain the function mapping, if other than the current database.

user_name_1

Name of user to contain the function mapping, if other than the current user.

function_mapping_name

Name of function mapping. For information on naming objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

database_name_2

Name of the database containing the function, if other than the current database for the session.

user_name_2

Name of the user containing the function, if other than the current user for the session.

schema_name

Name of schema associated with function.

function_name

Name of function or table operator to map.

You must make sure that the function is present in the location you specify.

For functions or table operators stored within the database, if you do not specify a database or user, the location defaults to the following, in order of precedence:

- Current default database or user for the session
- SYSLIB
- TD_SYSFNLIB

See [Example: Function Mapping Definition for a Function Within the Database](#).

SERVER

Name of external server where the function or table operator is located.

For a function or table operator within the database, you do not specify the SERVER clause.

database_name_3

Name of database to contain the function, if other than the current database.

user_name_3

Name of user to contain the function, if other than the current user.

server_name

Name of server containing the function.

EXTERNAL SECURITY

Specifies an authorization for accessing remote storage. See [CREATE AUTHORIZATION](#) and [REPLACE AUTHORIZATION](#).

DEFINER

Specify DEFINER to share an authorization object with multiple users of the database in which it resides. You can create the authorization in any database.

INVOKER

Specify INVOKER to allow exclusive access by a user. You must create the authorization in the database of the current user.

TRUSTED

Required keyword.

authorization_name

Name of the authorization object.

json_document

JSON format document containing input to function in the form of parameters and values.

STOREDAS

Use only when creating a mapping for the WRITE_NOS table operator.

Specifies the formatting style of the external data:

Option	Description
'PARQUET'	External data is formatted as Parquet. For Parquet data, you must specify STOREDAS ('PARQUET').
'TEXTFILE' (default)	External data uses a text-based format, such as CSV or JSON.

table_list

Optionally, you can define input tables, output tables, and parameters to use during function processing.

Note:

You must define all input tables, output tables, and parameters that users can specify during function processing.

See [Example: Function Mapping Definition with IN TABLE, OUT TABLE, and Parameters Options](#).

ANY IN TABLE

You can use this clause to correspond to input tables specified in the table operator ON clause during function processing that do not have a correlation name and do not match any of the named parameters. See [Example: Function Mapping with Multiple ANY IN TABLE Clauses](#).

You can also specify a name to substitute for non-mapped tables in a table operator ON clause. In the example below, a non-mapped name specified during function processing is replaced with *ConversionEvents*:

```
ConversionEvents(ANY) IN TABLE
```

See [Example: Function Mapping with \(ANY\) IN TABLE Clause Substitution](#).

name (value)

Parameters and values to use as defaults during function processing. The default value is used only when a SELECT statement specifies a parameter without a value in the table operator USING clause. See [Example: Function Mapping Definition that Includes a Variable with a Default Value](#).

For *value*, you can specify:

- Literal value list

You can specify character, numeric, data, time, interval, period, and graphic literals. For information about the literals you can specify, see *Teradata Vantage™ - Data Types and Literals*.

- Scalar subquery expression (SSQ)

You can specify a scalar subquery expression.

A scalar subquery can include a system variable. For example, the parameter *ValueColumn* has a default value specified by a scalar subquery that includes the system variable USER:

```
ValueColumn(SELECT colname FROM ssqtbl WHERE username=USER)
```

See [Example: Function Mapping Definition with Scalar Subquery \(SSQ\) Substitution](#) and [Example: Function Mapping with Scalar Subquery \(SSQ\) Substitution that Includes a System Variable](#).

- System variables

You can specify the following system variables: USER, CURRENT_USER, ROLE, CURRENT_ROLE, DATE, CURRENT_DATE, TIME, CURRENT_TIME, CURRENT_TIMESTAMP, PROFILE, DATABASE, SESSION, TD_HOST, ACCOUNT, or ZONE.

For example, *TimeOfEntry* defaults to the current time, as specified by the CURRENT_TIME system variable:

```
TimeOfEntry(CURRENT_TIME)
```

- User variables

A variable can be a parameter name and can contain a default value, a default value list, a scalar subquery, or another variable.

Variables specified for parameters in the function mapping definition are resolved before the parameter is sent for function processing. The value for the variable can be derived either from a default value specified in function or specified during function processing. See [Example: Function Mapping Definition with Multiple Variable Substitutions](#).

You can nest variables, for example:

```
MaxStep(Maxnum) ,
Maxnum(Maxnumber),
Maxnumber(10),
```

However, the resolution of nested variables cannot result in a circular reference.

In the example below, the variable *MaxStep* has a default value of 150:

```
MaxStep (150)
```

In this example, the variable *MaxStep* corresponds to the variable *maxnum*:

```
MaxStep(maxnum)
```

A variable with a default value list cannot include a concatenated variable expression. For example, you cannot specify expressions similar to either of the following:

```
ValueColumn(AttributeValueColumn||'_'||Maxnum)
Maxnum(10,20)
```

```
ValueColumn( AttributeValueColumn, ValCol )
ValCol(ValColumn||'_'||MaxNum)
```

- Concatenated variable expression

A list of string constants or variables separated by the concatenation operator (||). For example, the *ValueColumn* parameter below has a default value specified by a concatenated variable expression:

```
ValueColumn(AttributeValueColumn || '_' || Maxnum)
```

See [Example: Function Mapping with Concatenated Variable Substitution](#).

A concatenated variable expression cannot contain a scalar subquery.

You cannot specify a concatenated variable expression in IN TABLE and OUT TABLE clauses.

name IN TABLE

Correlation name of an input table to use during function processing. See [Example: Function Mapping Definition with IN TABLE Options](#).

You can also specify a substitute name to enable another name to be used during function processing. In the example below, the table *ConversionEvents* is used instead of the table *conversion* for function processing:

```
ConversionEvents(conversion) IN TABLE
```

See [Example: Function Mapping Definition with IN TABLE Clause Substitution](#).

You cannot specify IN TABLE clause variables as parameters or values in other function parameters.

You cannot specify the same variable name in multiple IN TABLE clauses.

You cannot specify ANY as a value if you use the ANY IN TABLE clause.

An IN TABLE clause cannot contain any of the following:

- Concatenated variable expression
- Scalar subquery expression (SSQ)
- Default value
- Nested values

name OUT TABLE

Name of the output table to use during function processing. You can also specify a substitute name to enable another name to be used during function processing. See [Example: Function Mapping Definition with a Substitution in the OUT TABLE Clause](#).

You can nest OUT TABLE parameters, for example:

```
ModelTable(model_table) OUT TABLE ,
model_table(md_tbl) OUT TABLE
```

You cannot specify OUT TABLE clause variables as parameters or values in other function parameters.

You cannot specify the same variable name in multiple OUT TABLE clauses.

An OUT TABLE clause cannot contain any of the following:

- Concatenated variable expression
- Scalar subquery expression (SSQ)
- Default value

Examples

Example: Defining a Function Mapping

Below is an example of a definition for a function mapping named *myfnc* on the database *appl_view_db* for the function *fnc* located on the server *coprocessor*.

```
CREATE FUNCTION MAPPING appl_view_db.myfnc
  FOR fnc SERVER coprocessor;
```

Example: Replacing a Function Mapping Definition

This function mapping definition replaces the function mapping *Attribution* for the function *attribution* located on the server *coprocessor* and sets a default value of 10 for *WindowSize*:

```
REPLACE FUNCTION MAPPING appl_view_db.Attribution
  FOR attribution SERVER coprocessor
  MAP JSON ( '{ "function_version": "1.0" }' )
  USING
    ANY IN TABLE,
    conversion IN TABLE,
    excluding IN TABLE,
    optional IN TABLE,
    model1 IN TABLE,
    model2 IN TABLE,
    EventColumn, TimestampColumn, WindowSize (10);
```

Example: Function Mapping Definition for a Function Within the Database

The function mapping definition *user_Attribution* specifies the function *attribution* in the current database.

```
CREATE FUNCTION MAPPING user_Attribution
  FOR attribution
  USING
    ANY IN TABLE,
    ANY IN TABLE,
    conversion IN TABLE,
    excluding IN TABLE,
    optional IN TABLE,
    model1 IN TABLE,
    model2 IN TABLE,
    EventColumn, TimestampColumn, WindowSize;
```

Example: Function Mapping Definition with IN TABLE, OUT TABLE, and Parameters Options

Below is the function mapping definition *glm* for the function *glm* located on the server *coprocessor*.

During function processing, you can specify the following options in the SELECT FROM table operator USING clause:

- *InputTable* as an input table correlation name.
- *OutputTable* as an output table.
- Parameters including *ColumnNames*, *CategoricalColumns*, *Family*, *Link*, *Weight*, *Threshold*, *MaxIterNum*, *Intercept*, and *Step*.

```
CREATE FUNCTION MAPPING appl_view_db.glm
  FOR glm SERVER coprocessor
  MAP JSON ( '{ "function_version": "1.0" }' )
  USING
    InputTable IN TABLE,
    OutputTable OUT TABLE,
    ColumnNames, CategoricalColumns, Family,
    Link, Weight, Threshold, MaxIterNum, Intercept, Step;
```

Example: Function Mapping Definition with IN TABLE and Parameter Options

Below is the function mapping definition *Attribution* for the function *attribution* located on the server *coprocessor*.

During function processing, the following can be specified:

- A single input table without a correlation name.

- Input table correlation names *conversion*, *excluding*, *optional*, *model1*, and *model2*.
- Parameters including *EventColumn*, *TimestampColumn*, *WindowSize*.

```
CREATE FUNCTION MAPPING appl_view_db.Attribution
  FOR attribution SERVER coprocessor
  MAP JSON ( '{ "function_version": "1.0" }' )
  USING
    ANY IN TABLE,
    conversion IN TABLE,
    excluding IN TABLE,
    optional IN TABLE,
    model1 IN TABLE,
    model2 IN TABLE,
    EventColumn, TimestampColumn, WindowSize;
```

Example: Function Mapping Definition that Includes a Variable with a Default Value

This function mapping definition specifies a default value of 10 for the *Maxnum* variable.

If *Maxnum* is not specified during function processing, the value 10 is used.

```
CREATE FUNCTION MAPPING usr_SVMSparse
  FOR SparseSVMTrainer SERVER TD_SERVER_DB.coprocessor
  USING
    InputTable(inp_table) IN TABLE ,
    ModelTable(model_table) OUT TABLE ,
    IDColumn ,
    SampleIdColumn ,
    AttributeColumn(AttributeNameColumn) ,
    LabelColumn(ResponseColumn),
    ValueColumn(AttributeValueColumn),
    HashProjection , "Hash" , HashBuckets ,
    Cost , Bias , ClassWeights ,
    MaxStep(Maxnum) ,
    Maxnum(10),
    Epsilon , Seed , SequenceInputBy
  ;
```


Example: Function Mapping Definition with Multiple Variable Substitutions

The function mapping definition *usr_SVMSparse* specifies the following variable substitutions:

- *AttributeColumn* is substituted for *AttributeNameColumn*
- *LabelColumn* is substituted for *ResponseColumn*
- *ValueColumn* is substituted for *AttributeValueColumn*

If the parameter *AttributeNameColumn* is specified during function processing, *AttributeColumn* is substituted for function processing. Similarly, if *ResponseColumn* is specified during function processing, *LabelColumn* is used. If *AttributeValueColumn* is specified during function processing, *ValueColumn* is used.

```
CREATE FUNCTION MAPPING usr_SVMSparse
FOR SparseSVMTrainer SERVER TD_SERVER_DB.coprocessor
USING
InputTable IN TABLE ,
ModelTable OUT TABLE ,
IDColumn ,
SampleIdColumn ,
AttributeColumn(AttributeNameColumn) ,
LabelColumn(ResponseColumn),
ValueColumn(AttributeValueColumn),
HashProjection , "Hash" , HashBuckets ,
Cost , Bias , ClassWeights ,
MaxStep ,
Epsilon , Seed , SequenceInputBy
;
```

Example: Function Mapping with the ANY IN TABLE Option

Below is the function mapping definition *CCMPPrepare* for the *ccmpprepare* function on the server *coprocessor*.

During function processing, you can specify a single table that does not have a correlation name.

```
CREATE FUNCTION MAPPING appl_view_db.CCMPPrepare
FOR ccmpprepare SERVER coprocessor
MAP JSON ( '{ "function_version": "1.0" }' )
USING
    ANY IN TABLE;
```

Example: Function Mapping with Multiple ANY IN TABLE Clauses

This function mapping definition includes two ANY IN TABLE clauses.

During function processing, the first two ON clauses that do not have correlation names correspond to the two ANY IN TABLE clauses.

```
CREATE FUNCTION MAPPING user_Attribution
  FOR attribution MAP JSON ( '{ "function_version": "1.0" }' )
  USING
    ANY IN TABLE,
    ANY IN TABLE,
    conversion IN TABLE,
    excluding IN TABLE,
    optional IN TABLE,
    model1 IN TABLE,
    model2 IN TABLE,
    EventColumn, TimestampColumn, WindowSize;
```

This SELECT statement includes two ON clauses do not have correlation names: :

- *attribution_sample_table1*
- *attribution_sample_table2*

These tables correspond to the two ANY IN TABLE clauses of the *user_Attribution* function mapping.

```
SELECT * FROM user_ATTRIBUTION(
  ON attribution_sample_table1 PARTITION BY user_id
    ORDER BY time_stamp
  ON attribution_sample_table2 PARTITION BY user_id
    ORDER BY time_stamp
  ON conversion_event_table AS conversion DIMENSION
  ON excluding_event_table AS excluding DIMENSION
  ON optional_event_table AS optional DIMENSION
  ON model1_table AS model1 DIMENSION
  ON model2_table AS model2 DIMENSION
  USING
    EVENT_COLUMN_NAME ('event')
    TIMESTAMP_COLUMN_NAME ('time_stamp')
    WINDOW('rows:10&seconds:20')
  ) as dt ORDER BY user_id, time_stamp;
```

Example: Function Mapping with (ANY) IN TABLE Clause Substitution

The function mapping definition for *user_attribution* specifies an (ANY) IN TABLE clause substitution. If a single table name is specified without a correlation during function processing, *ConversionEvents* is used.

```
CREATE FUNCTION MAPPING user_attribution
FOR Attribution SERVER coprocessor
USING
inputtable IN TABLE,
inputtable1 IN TABLE,
ConversionEvents(ANY) IN TABLE,
excluding IN TABLE,
optional IN TABLE,
model1 IN TABLE,
model2 IN TABLE,
EVENT_COLUMN_NAME ('event'),
TIMESTAMP_COLUMN_NAME ('time_stamp' ),
WINDOW(WindowSize);
```

Example: Function Mapping Definition with IN TABLE Options

Below is the function mapping definition *GLMPredict* for the *glmpredict* function on the server *coprocessor*.

During function processing, the following can be specified in the table operator ON clause:

- A single input table without a correlation name.
- The input table *MODEL* as a correlation name.

```
CREATE FUNCTION MAPPING appl_view_db.GLMPredict
FOR glmpredict SERVER coprocessor
MAP JSON ( '{ "function_version": "1.0" }' )
USING
ANY IN TABLE,
MODEL IN TABLE;
```

Example: Function Mapping Definition with IN TABLE Clause Substitution

The function mapping definition for *user_attribution* includes an IN TABLE clause that substitutes *ConversionEvents* for *conversion*.

If the correlation variable *conversion* is specified during function processing, *ConversionEvents* is substituted.

```
CREATE FUNCTION MAPPING user_attribution
FOR Attribution SERVER coprocessor
USING
inputtable IN TABLE,
ANY IN TABLE,
ConversionEvents(conversion) IN TABLE,
excluding IN TABLE,
optional IN TABLE,
model1 IN TABLE,
model2 IN TABLE,
EVENT_COLUMN_NAME ('event'),
TIMESTAMP_COLUMN_NAME ('time_stamp' ),
WINDOW(WindowSize);
```

Example: Function Mapping Definition with a Substitution in the OUT TABLE Clause

The function mapping definition *usr_SVMSparse* specifies a substitution of *ModelTable* for *model_table* in the OUT TABLE clause.

```
CREATE FUNCTION MAPPING usr_SVMSparse
FOR SparseSVMTrainer SERVER TD_SERVER_DB.coprocessor
USING
InputTable IN TABLE ,
ModelTable(model_table) OUT TABLE ,
IDColumn ,
SampleIdColumn ,
AttributeColumn(AttributeColumnName) ,
LabelColumn(ResponseColumn),
ValueColumn(AttributeValueColumn),
HashProjection , "Hash" , HashBuckets ,
Cost , Bias , ClassWeights ,
MaxStep ,
Epsilon , Seed , SequenceInputBy
;
```

Example: Function Mapping Definition with Scalar Subquery (SSQ) Substitution

The function mapping definition *sessionize_fm* specifies a scalar subquery for the TIMEOUT column value:

```
SELECT timeout FROM ssqtbl WHERE username='John'
```

If TIMEOUT is not specified during function processing, the column value is resolved by processing the scalar subquery.

```
CREATE FUNCTION MAPPING sessionize_fm FOR sessionize SERVER coprocessor
USING
  InputTable IN TABLE,
  ModelTable IN TABLE,
  ClusterTable OUT TABLE,
  TIMECOLUMN,
  TIMEOUT(SELECT timeout FROM ssqtbl WHERE username='John'),
  CLICKLAG,
  EMITNULL;
```

Example: Function Mapping with Scalar Subquery (SSQ) Substitution that Includes a System Variable

The function mapping definition *sessionize_fm* specifies a TIMEOUT parameter derived from a scalar subquery that includes the system variable USER.

If TIMEOUT is not specified during function processing, the column value is resolved by processing the scalar subquery:

```
SELECT timeout FROM ssqtbl WHERE username=USER
```

```
CREATE FUNCTION MAPPING sessionize_fm FOR sessionize SERVER coprocessor
USING
  InputTable IN TABLE,
  ModelTable IN TABLE,
  ClusterTable OUT TABLE,
  TIMECOLUMN,
  TIMEOUT(SELECT timeout FROM ssqtbl WHERE username=USER),
  CLICKLAG,
  EMITNULL;
```

Example: Function Mapping with Concatenated Variable Substitution

The function definition *usr_SVMSparse* includes a concatenated variable expression for the string constant *ValueColumn* that consists of *AttributeValueColumn* and *Maxnum*.

If correlation variables *AttributeValueColumn* and *Maxnum* are both specified during function processing, *ValueColumn* is substituted.

```
CREATE FUNCTION MAPPING usr_SVMSparse
FOR SparseSVMTrainer SERVER TD_SERVER_DB.coprocessor
USING
InputTable IN TABLE ,
ModelTable OUT TABLE ,
IDColumn ,
SampleIdColumn ,
AttributeColumn(AttributeColumnName) ,
LabelColumn(ResponseColumn),
ValueColumn(AttributeValueColumn||'_'||Maxnum),
HashProjection , "Hash" , HashBuckets ,
Cost , Bias , ClassWeights ,
MaxStep(Maxnum) ,
Epsilon , Seed , SequenceInputBy
;
```

Example: Function Mapping for READ_NOS

This statement creates the function mapping *NOS_Reader* for the *READ_NOS* table operator using the authorization *MyObjAuth*.

Note:

To create a function mapping for *READ_NOS*, you must have the *EXECUTE FUNCTION* privilege on *READ_NOS*.

For information about the *READ_NOS* table operator, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Create an authorization object called *MyObjAuth*:

```
CREATE AUTHORIZATION MyObjAuth
AS DEFINER TRUSTED
USER 'YOUR-ACCESS-KEY-ID'
PASSWORD 'YOUR-SECRET-ACCESS-KEY' ;
```

Create the function mapping for READ_NOS:

```
CREATE FUNCTION MAPPING NOS_Reader
FOR READ_NOS
EXTERNAL SECURITY DEFINER TRUSTED MyObjAuth
USING
LOCATION,
BUFFERSIZE,
RETURNTYPE,
SAMPLE_PERC,
STOREDAS,
FULLSCAN,
MANIFEST,
ROWFORMAT,
HEADER,
ANY IN TABLE;
```

Example: Function Mapping for WRITE_NOS

This statement creates the function mapping NOS_Writer for the WRITE_NOS table operator using the authorization MyObjAuth.

Note:

To create a function mapping for WRITE_NOS, you must have the EXECUTE FUNCTION privilege on WRITE_NOS.

For information about the WRITE_NOS table operator, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Create an authorization object called MyObjAuth:

```
CREATE AUTHORIZATION MyObjAuth
AS DEFINER TRUSTED
  USER 'YOUR-ACCESS-KEY-ID'
  PASSWORD 'YOUR-SECRET-ACCESS-KEY';
```

Create the function mapping for WRITE_NOS:

```
CREATE FUNCTION MAPPING NOS_Writer
FOR WRITE_NOS
EXTERNAL SECURITY DEFINER TRUSTED MyObjAuth
USING
LOCATION,
```

```

STOREDAS,
NAMING,
MANIFESTFILE,
MANIFESTONLY,
OVERWRITE,
INCLUDE_ORDERING,
INCLUDE_HASHBY,
MAXOBJECTSIZE,
COMPRESSION,
ANY IN TABLE;

```

ALTER FUNCTION

Performs either or both of the following functions.

- Controls whether an existing function can run in protected mode as a separate process or in unprotected mode as part of the database.
- Recompiles C or C++ functions or relinks Java functions and redistributes them.

Note:

Java UDFs must always run in protected mode, so you cannot use this statement to change their protection mode.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

To alter a function definition, you must have the ALTER FUNCTION privilege on that function.

There are no privileges granted automatically.

ALTER FUNCTION Syntax

```

ALTER { SPECIFIC FUNCTION [ database_name_1. | user_name_1. ]
      specific_function_name |
      FUNCTION [ database_name_2. | user_name_2. ] function_name
        [ ( { data_type | [SYSUDTLIB.] UDT_name } [,...] ) ]
      }
      { EXECUTE [NOT] PROTECTED | COMPILE [ONLY] } [;]

```


data_type

```

{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [ ( fractional_seconds_precision ) ] [WITH TIME
ZONE] |

  INTERVAL YEAR [(precision)] [TO MONTH] |

  INTERVAL MONTH [(precision)] |

  INTERVAL DAY [(precision)]
    [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] }] |

  INTERVAL HOUR [(precision)]
    [TO { MINUTE | SECOND [(fractional_seconds_precision)] }] |

  INTERVAL MINUTE [(precision)] [TO SECOND
[(fractional_seconds_precision)] ] |

  INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) ] |

  PERIOD (DATE) |

  PERIOD ( { TIME | TIMESTAMP } [(precision)] [WITH TIME ZONE] ) |

  REAL |

  DOUBLE PRECISION |

  FLOAT [ ( integer ) ] |

  NUMBER [ ( { integer | *} [, integer].... ) ] |

  { DECIMAL | NUMERIC } [ ( integer [, integer].... ) ] |

  { CHAR | BYTE | GRAPHIC } [ ( integer ) ] |

  { VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [ ( integer ) ] |

  LONG VARCHAR |

  LONG VARGRAPHIC |

```

```

{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
  ( integer [ G | K | M ] ) |

[SYSUDTLIB.] { XML | XMLTYPE } [ ( integer [ G | K | M ] ) ]
  [ INLINE LENGTH integer ] |

[SYSUDTLIB.] JSON [ ( integer [ K | M ] ) ] [ INLINE LENGTH integer ]
  [ CHARACTER SET { UNICODE | LATIN } | STORAGE FORMAT { BSON |
  UBJSON } ] |

[SYSUDTLIB.] ST_GEOMETRY [ ( integer [ K | M ] ) ] [ INLINE LENGTH
integer ] |

[SYSUDTLIB.] DATASET [ ( integer [ K | M ] ) ] [ INLINE LENGTH
integer ]
  storage_format |

[SYSUDTLIB.] { UDT_name | MBR | ARRAY_name | VARRAY_name }
}

```

storage_format

```

STORAGE FORMAT { Avro | CSV [ CHARACTER SET { UNICODE | LATIN } ] }
  [ WITH SCHEMA [database.] schema_name ]

```

ALTER FUNCTION Syntax Elements***database_name_1***

The containing database for *specific_function_name*, if something other than the current database.

user_name_1

The containing user for *specific_function_name*, if something other than the current user.

specific_function_name

The specific function name for the function to be altered.

database_name_2

The containing database for *function_name*, if other than the current database.

user_name_2

The containing user for *function_name*, if other than the current user.

function_name

Name of the function to alter.

data_type

An optional data type specification for the parameters passed to *function_name*.

If you do not specify a data type list, then the specified function must be the only one with that name in the database.

UDT_name

The name of any UDT included in the list of data types for *function_name*.

EXECUTE PROTECTED

Change the execution mode for the specified function from unprotected mode to protected mode. For details, see ALTER FUNCTION in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

EXECUTE NOT PROTECTED

Change the execution mode for the specified function from protected mode to unprotected mode.

This option is not valid for Java UDFs.

COMPILE

This option performs the following actions:

C or C++ function	Java function
Recompiles the code source for the specified function.	Recompiles the code source for the specified function.
Generates the object code.	Generates the object code.
Recreates the .so file.	Distributes the related JAR files for the function.
Distributes the .so file to all nodes of the system.	

The existing function object is replaced by the recompiled version.

ONLY

Recompile the UDF only. When you specify this option, Vantage does not distribute a new dynamic linked library to database nodes.

When you load a UDF onto another platform of a different type, the system marks it as not valid, and it must be recompiled. If there are many UDFs in one database, it saves time to specify the ONLY option for all recompilations to avoid having to generate and distribute a new library, until the last one is compiled in that database leaving off the ONLY option.

The COMPILE ONLY option is not valid for Java UDFs.

Usage Notes

Using ALTER FUNCTION or REPLACE FUNCTION with Row-Level Security UDFs

If you use ALTER FUNCTION or REPLACE FUNCTION on a row-level security UDF, the resulting UDF must retain all parameters specified in the constraint definitions that use the UDF.

ALTER FUNCTION Examples

Example: Setting the Protection Mode of a UDF to EXECUTE NOT PROTECTED

You have created and successfully debugged a UDF named TransXML and you now want to set its protection mode to EXECUTE NOT PROTECTED. The following request assumes there is no other UDF in the database named TransXML:

```
ALTER FUNCTION TransXML EXECUTE NOT PROTECTED;
```

Example: Setting a Recompiled UDF to EXECUTE NOT PROTECTED

An existing UDF named XPathValue must be recompiled for some reason. XPathValue had previously been set to run in unprotected mode. Recompiling a UDF always resets its protection mode back to protected, so you must follow any recompilation with another ALTER FUNCTION request to set its protection mode back to unprotected.

These statements recompile XPathValue and then set its protection mode back to unprotected:

```
ALTER FUNCTION XPathValue COMPILE;
ALTER FUNCTION XPathValue EXECUTE NOT PROTECTED;
```

DROP FUNCTION

Drops the definition of the specified function or specific function from the Data Dictionary and from the containing database or user.

You cannot drop an external UDF being used as a cast, ordering, or transform routine. You must first drop the definition for the cast, ordering, or transform. See [DROP CAST](#), [DROP ORDERING](#), or [DROP TRANSFORM](#).

Note:

You cannot drop a UDF that is referenced in a row-level security constraint definition if the constraint is assigned to a user, profile, or table. You can use ALTER CONSTRAINT to drop or replace a UDF from a row-level security constraint definition, and then drop the UDF. See [ALTER CONSTRAINT](#).

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Required Privileges

You must have the DROP FUNCTION privilege on an external or SQL function or its containing database or user to drop it.

In addition, you must remove the UDF from any constraint objects that reference it before dropping the UDF.

DROP FUNCTION Syntax

```
DROP { SPECIFIC FUNCTION [ database_name_1. ] specific_function_name |
      FUNCTION [ database_name_2. ] function_name [ ( data_type [,...] ) ]
} [;]
```

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [ ( fractional_seconds_precision ) ] [WITH TIME
ZONE] |

  INTERVAL YEAR [(precision)] [TO MONTH] |

  INTERVAL MONTH [(precision)] |

  INTERVAL DAY [(precision)]
    [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] }] |

  INTERVAL HOUR [(precision)]
```

```

    [TO { MINUTE | SECOND [(fractional_seconds_precision)] }] |

    INTERVAL MINUTE [(precision)] [TO SECOND
[(fractional_seconds_precision)]] |

    INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) |

    PERIOD (DATE) |

    PERIOD ( { TIME | TIMESTAMP } [(precision)] [WITH TIME ZONE] ) |

    REAL |

    DOUBLE PRECISION |

    FLOAT [ (integer) ] |

    NUMBER [ ( { integer | *} [, integer]... ) ] |

    { DECIMAL | NUMERIC } [ ( integer [, integer]... ) ] |

    { CHAR | BYTE | GRAPHIC } [ (integer) ] |

    { VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [ (integer) ] |

    LONG VARCHAR |

    LONG VARGRAPHIC |

    { BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
    ( integer [ G | K | M ] ) |

    [SYSUDTLIB.] { XML | XMLTYPE } [ ( integer [ G | K | M ] ) ]
    [ INLINE LENGTH integer ] |

    [SYSUDTLIB.] JSON [ ( integer [ K | M ] ) ] [ INLINE LENGTH integer ]
    [ CHARACTER SET { UNICODE | LATIN } | STORAGE FORMAT { BSON |
    UBJSON } ] |

    [SYSUDTLIB.] ST_GEOMETRY [ (integer [ K | M ]) ] [ INLINE LENGTH
integer ] |

    [SYSUDTLIB.] DATASET [ (integer [ K | M ]) ] [ INLINE LENGTH

```

```

integer ]
    storage_format |

[SYSUDTLIB.] { UDT_name | MBR | ARRAY_name | VARRAY_name }
}

```

storage_format

```

STORAGE FORMAT { Avro | CSV [ CHARACTER SET { UNICODE | LATIN } ] }
[ WITH SCHEMA [database.] schema_name ]

```

DROP FUNCTION Syntax Elements***database_name_1***

Name of the containing database for *specific_function_name*, if different from the current database.

specific_function_name

Specific function name for the external or SQL function to drop.

database_name_2

Name of the containing database for *function_name*, if different from the current database.

function_name

Function name for the external or SQL function to drop.

data_type

Optional data type specification for the parameters passed to *function_name*.

If you do not specify a data type list, then the specified function must be the only one with that name in the database.

Only the data types associated with the function parameters need be specified.

The length specification for a string is not relevant, but you can specify it. Likewise, neither the length for DECIMAL, TIME, TIMESTAMP, and INTERVAL types nor the decimal placement for the DECIMAL type is relevant, but you can specify it.

Usage Notes

Differences Between the External and SQL Forms of DROP FUNCTION

The syntax and privileges for the external and SQL forms of DROP FUNCTION are the same except that SQL functions do not apply to UDTs or linkage information.

DROP FUNCTION Examples

Example: Dropping an SQL Function

These examples drop an SQL function and a specific SQL function.

```
DROP FUNCTION udf2 (INTEGER, INTEGER);
DROP SPECIFIC FUNCTION specific_udf2;
```

Example: Dropping an External Function by Function Name or Specific Function Name

You can drop a function definition using either its function name or its specific function name. For example, consider the following function definition.

```
CREATE FUNCTION Finger_Print_Match(VARBYTE(1000),
VARBYTE(1000))
  RETURNS INTEGER
  SPECIFIC fpm1
  LANGUAGE C
  NO SQL
  PARAMETER STYLE TD_GENERAL
  EXTERNAL;
```

Either of the following DROP FUNCTION requests drops its definition, the first using its specific function name and the second using its function name.

```
DROP SPECIFIC FUNCTION fpm1;
DROP FUNCTION Finger_print_Match(VARBYTE, VARBYTE);
```

Example: Dropping an External Function and Unique Function Names

You can drop a function definition using either its function name or its specific function name. The example also shows that if the name cannot be resolved to a unique object using the DROP FUNCTION statement you specify, you cannot drop it.

Consider the following function definition.


```
CREATE FUNCTION test_image(VARCHAR(5000), INTEGER) ...
  SPECIFIC testi_opt1 ...;
CREATE FUNCTION test_image(VARCHAR(5000), FLOAT)...
  SPECIFIC testi_opt2 ...;
```

The following DROP FUNCTION requests both drop this function.

```
DROP SPECIFIC FUNCTION testi_opt1;
```

```
DROP FUNCTION test_image(VARCHAR, INTEGER);
```

The following statement returns an error because there is not an instance of a function named *test_image* without parameters.

```
DROP FUNCTION test_image();
```

Related Information

- ALTER FUNCTION in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- CREATE FUNCTION (External Form) and REPLACE FUNCTION (External Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- CREATE FUNCTION (SQL Form) and REPLACE FUNCTION (SQL Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- [CREATE GLOBAL TEMPORARY TRACE TABLE](#)
- CREATE GLOBAL TEMPORARY TRACE TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- RENAME FUNCTION (External Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- RENAME FUNCTION (SQL Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- [SET SESSION FUNCTION TRACE](#)
- SET SESSION FUNCTION TRACE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- HELP FUNCTION in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- [SHOW object](#)

Also see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about external functions.

DROP FUNCTION MAPPING

Drops a function mapping.

Required Privileges

You must have the DROP FUNCTION privilege on the function mapping or on the containing database or user.

DROP FUNCTION MAPPING Syntax

```
DROP FUNCTION MAPPING [ database_name. | user_name. ] function_mapping_name [;]
```

DROP FUNCTION MAPPING Syntax Elements

database_name

Name of the containing database, if not the current database.

user_name

Name of the containing user, if not the current user.

function_mapping_name

Name of the function mapping to drop.

Examples

Example: Dropping a Function Mapping

This statement drops the function mapping glm in the database appl_view_db.

```
DROP FUNCTION MAPPING appl_view_db.glm;
```

HELP FUNCTION

Reports the specific function name, list of parameters, the data types of the parameters, whether the function is used to compress or decompress character or graphic data, and any comments associated with the parameters for SQL, scalar, aggregate, and table functions.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have at least one privilege on the function or its containing database.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against a specified function.

HELP FUNCTION Syntax

```
HELP { SPECIFIC FUNCTION [ database_name_1. | user_name_2. ]
      specific_function_name |
      FUNCTION [ database_name_2. | user_name_2. ] function_name
      [(data_type [, ...])]
      } [;]
```

HELP FUNCTION Syntax Elements

SPECIFIC FUNCTION

Specifies that the help request uses the specific name of the user-defined function.

database_name_1

user_name_1

Containing database or user for *specific_function_name* if different from the current database or user.

specific_function_name

Specific name of the user-defined function for which help information is requested. For information about naming database objects, see SQL Fundamentals.

FUNCTION

Specifies that the help request uses the name of the user-defined function.

database_name_2

user_name_2

Containing database or user for *function_name* if different from the current database or user.

function_name

Name of the user-defined function for which help information is requested.

data_type

Data type parameter set that uniquely identifies an overloaded function name.

Examples

Example: HELP FUNCTION Report

This example demonstrates a simple HELP FUNCTION report.

```

HELP FUNCTION first1;

*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.

      Parameter Name P1
              Type I
            Comment ?
            Nullable Y
              Format -(10)9
        Max Length 4
    Decimal Total Digits ?
Decimal Fractional Digits ?
      Table/View? F
        Char Type ?
    Parameter Type I
        UDT Name ?
    Parameter Name P2
              Type F
            Comment ?
            Nullable Y
              Format -9.999999999999999E-999
        Max Length      8
    Decimal Total Digits ?
Decimal Fractional Digits ?
      Table/View? F
        Char Type ?
    Parameter Type I
        UDT Name      ?
    Parameter Name RETURN0
              Type F
            Comment ?
            Nullable Y
              Format -9.999999999999999E-999
        Max Length      8
    Decimal Total Digits ?

```

```

Decimal Fractional Digits ?
      Table/View? F
      Char Type ?
Parameter Type 0
      UDT Name ?

```

Example: HELP FUNCTION Report for a UDF

This example demonstrates a HELP FUNCTION report for a UDF defined with the TD_GENERAL parameter style. Notice that the value of the *Nullable* column is N for all parameters.

```

HELP FUNCTION tdgenfnc;
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.

      Parameter Name P1
              Type I
              Comment ?
              Nullable N
              Format -(10)9
              Max Length 4
      Decimal Total Digits ?
      Decimal Fractional Digits ?
              Table/View? F
              Char Type ?
      Parameter Type I
              UDT Name ?
      Parameter Name P2
              Type F
              Comment ?
              Nullable N
              Format -9.999999999999999E-999
              Max Length 8
      Decimal Total Digits ?
      Decimal Fractional Digits ?
              Table/View? F
              Char Type ?
      Parameter Type I
              UDT Name ?
      Parameter Name RETURN0
              Type F
              Comment ?
              Nullable N

```

```

        Format -9.999999999999999E-999
        Max Length 8
    Decimal Total Digits ?
    Decimal Fractional Digits ?
        Table/View? F
        Char Type ?
    Parameter Type 0
        UDT Name ?

```

Example: Displaying UDT Parameters

The following example shows how UDT parameters are indicated in a HELP FUNCTION report.

```

HELP FUNCTION  SYSUDTLIB.AdhocIntToSQL;

    Parameter Name P1
        Type I
        Comment ?
        Nullable N
        Format -(10)9
        Max Length 4
    Decimal Total Digits ?
    Decimal Fractional Digits ?
        Table/View? F
        Char Type ?
    Parameter Type I
        UDT Name ?
    Parameter Name RETURN0
        Type UT
        Comment ?
        Nullable N
        Format ?
        Max Length ?
    Decimal Total Digits ?
    Decimal Fractional Digits ?
        Table/View? F
        Char Type ?
    Parameter Type 0
        UDT Name UDTINT

```

Example: Displaying Multidimensional ARRAY Parameters

Suppose you create a multidimensional ARRAY named *seismic_cube*. The following example shows how ARRAY parameters are indicated in a HELP FUNCTION report. If *seismic_cube* were a one-dimensional ARRAY type, that would be indicated with A1 rather than AN for the Type attribute.

```
HELP FUNCTION  SYSUDTLIB.3d_array;

      Parameter Name P1
                Type I
                Comment ?
                Nullable N
                Format -(10)9
      Max Length 4
      Decimal Total Digits ?
      Decimal Fractional Digits ?
      Table/View? F
      Char Type ?
      Parameter Type I
      UDT Name ?
      Parameter Name RETURN0
                Type AN
                Comment ?
                Nullable N
                Format ?
      Max Length ?
      Decimal Total Digits ?
      Decimal Fractional Digits ?
      Table/View? F
      Char Type ?
      Parameter Type 0
      UDT Name seismic_cube
```

Example: VARIANT_TYPE Input Parameter Data Type

The following example shows the output from a HELP FUNCTION request when the function specifies VARIANT_TYPE UDTs as input parameters.

```
HELP FUNCTION udf_agch002002dynudt;
*** Help information returned. 2 rows.
*** Total elapsed time was 1 second.
```

```

Parameter Name parameter_1
      Type UT
      Comment ?
      Nullable Y
      Format ?
      Max Length ?
      Decimal Total Digits ?
      Decimal Fractional Digits ?
      Table/View? A
      Char Type ?
Parameter Type I
      UDT Name VARIANT_TYPE
Parameter Name RETURN0
      Type UT
      Comment ?
      Nullable Y
      Format ?
      Max Length ?
      Decimal Total Digits ?
      Decimal Fractional Digits ?
      Table/View? A
      Char Type ?
Parameter Type 0
      UDT Name INTEGERUDT

```

Example: TD_ANYTYPE Input and Output Parameter Data Type

You create a function named *udf_1*. The following SQL text is a partial definition for *udf_1*.

```

CREATE FUNCTION udf_1 (
  A INTEGER
  B TD_ANYTYPE)
RETURNS TD_ANYTYPE
SPECIFIC udf_1
... ;

```

A `HELP FUNCTION` statement for *udf_1* displays the output below, with `++` representing the `TD_ANYTYPE` parameter data type.

```

HELP FUNCTION udf_1;
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.

```


Parameter Name	Type	Comment
-----	-----	-----
A	I	?
B	++	?
RETURNØ	++	?

Example: Displaying a Java UDF with Array and Period Data Types

This statement creates a user defined type named `phonenumbers_ary` as an array:

```
CREATE TYPE phonenumbers_ary AS CHAR(10) ARRAY[5];
```

This statement creates a user defined type named `MYINT` as an integer:

```
CREATE TYPE MYINT AS INTEGER FINAL;
```

The UDF is created using `phonenumbers_ary` and `Period(date)` type parameters and returns a `MYINT` type value.

```
CREATE FUNCTION getPhoneNums(parameter_1 phonenumbers_ary,
                             parameter_2 Period(Date))
  RETURNS MYINT
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
  EXTERNAL NAME
  'UDF_JAR:UserDefinedFunctions.getPhoneNums';
```

Following is an example of `HELP FUNCTION` for a Java UDF:

```
HELP FUNCTION getPhoneNums;
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.
```

Parameter Name	Type	Comment
-----	-----	-----
parameter_1	A1	?
parameter_2	PD	?
RETURNØ	UT	?

Related Information

- [CREATE GLOBAL TEMPORARY TRACE TABLE](#)

- [SET SESSION FUNCTION TRACE](#)
- [SHOW object](#)
- `HELP FUNCTION` in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - SQL External Routine Programming*, B035-1147

RENAME FUNCTION (SQL Form)

Renames either the overloaded calling function name or specific function name for an SQL function.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Other SQL dialects support similar non-ANSI standard statements with names such as `RENAME`.

Required Privileges

You must have both of the following privileges to rename an SQL UDF:

- `DROP FUNCTION` on the SQL UDF or its containing database.
- `CREATE FUNCTION` on the database that contains the SQL UDF.

RENAME FUNCTION Syntax (SQL Form)

```
RENAME {
  SPECIFIC FUNCTION [ database_name_1. ] specific_function_name
    { TO | AS } new_specific_function_name |

  FUNCTION [ database_name_2. ] function_name [ ( data_type [,...] ) ]
    { TO | AS } new_function_name
} [;]
```

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [ ( fractional_seconds_precision ) ] [WITH TIME
ZONE] |

  INTERVAL YEAR [ ( precision ) ] [TO MONTH] |

  INTERVAL MONTH [ ( precision ) ] |

  INTERVAL DAY [ ( precision ) ]
```

```

    [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] }] |

INTERVAL HOUR [(precision)]
    [TO { MINUTE | SECOND [(fractional_seconds_precision)] }] |

INTERVAL MINUTE [(precision)] [TO SECOND
[(fractional_seconds_precision)]] |

INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) ] |

PERIOD (DATE) |

PERIOD ( { TIME | TIMESTAMP } [(precision)] [WITH TIME ZONE] ) |

REAL |

DOUBLE PRECISION |

FLOAT [ (integer) ] |

NUMBER [ ( { integer | *} [, integer]... ) ] |

{ DECIMAL | NUMERIC } [ ( integer [, integer]... ) ] |

{ CHAR | BYTE | GRAPHIC } [ (integer) ] |

{ VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [ (integer) ] |

LONG VARCHAR |

LONG VARGRAPHIC |

{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
    ( integer [ G | K | M ] ) |

{ XML | XMLTYPE } |

JSON [ ( integer ) ] [ CHARACTER SET { UNICODE | LATIN } ] |

[SYSUDTLIB.] { UDT_name | ST_Geometry | MBR | ARRAY_name
| VARRAY_name }
}

```

RENAME FUNCTION Syntax Elements (SQL Form)

database_name_1

Containing database for *specific_function_name* if something other than the current database or user.

specific_function_name

Existing specific SQL function name to be changed.

If you specify *specific_function_name*, you must also specify *new_specific_function_name*, not *new_function_name*.

new_specific_function_name

New specific function name for the SQL function.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

specific_function_name

New specific function name for the SQL function.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

database_name_2

Containing database for *function_name* if something other than the current database or user.

function_name

Existing overloaded calling name of the SQL function to be changed.

If you specify *function_name*, you must also specify *new_function_name*, not *new_specific_function_name*.

data_type

Data type specifications required to uniquely identify an overloaded SQL function to be renamed.

new_function_name

New function name for the SQL function.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

RENAME FUNCTION Examples (SQL Form)

Example: Renaming the Specific Name for a Function

This example renames the specific name for an SQL UDF.

```
RENAME SPECIFIC FUNCTION SpecificUDF1 TO SpecificUDF2;
```

Example: Renaming the Overloaded Calling Name for a Function

This example renames the overloaded calling name for an SQL UDF.

```
RENAME FUNCTION UDF1(INTEGER, INTEGER) TO UDF2;
```

Related Information

- [SHOW object](#)

Also see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

RENAME FUNCTION (External Form)

Renames either the overloaded calling function name or specific function name for an external function.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Other SQL dialects support similar non-ANSI standard statements with names such as RENAME.

Required Privileges

You must have both of the following privileges to rename an external UDF.

- DROP FUNCTION on the external UDF or its containing database.
- CREATE FUNCTION on the database that contains the external UDF.

If the function is a row-level security UDF, you must also have the CONSTRAINT DEFINITION privilege to execute RENAME FUNCTION.

Note:

Before renaming a row-level security UDF, you must use ALTER CONSTRAINT to drop or replace the UDF in any CONSTRAINT objects that specify the UDF.

RENAME FUNCTION Syntax (External Form)

```

RENAME {
  SPECIFIC FUNCTION [ database_name_1. ] specific_function_name
    { TO | AS } new_specific_function_name |

  FUNCTION [ database_name_2. ] function_name [ ( data_type [,...] ) ]
    { TO | AS } new_function_name
} [;]

```

data_type

```

{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [ ( fractional_seconds_precision ) ] [WITH TIME
ZONE] |

  INTERVAL YEAR [ ( precision ) ] [TO MONTH] |

  INTERVAL MONTH [ ( precision ) ] |

  INTERVAL DAY [ ( precision ) ]
    [TO { HOUR | MINUTE | SECOND [ ( fractional_seconds_precision ) ] } ] |

  INTERVAL HOUR [ ( precision ) ]
    [TO { MINUTE | SECOND [ ( fractional_seconds_precision ) ] } ] |

  INTERVAL MINUTE [ ( precision ) ] [TO SECOND
[ ( fractional_seconds_precision ) ] ] |

  INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) ] |

  PERIOD (DATE) |

  PERIOD ( { TIME | TIMESTAMP } [ ( precision ) ] [WITH TIME ZONE] ) |

  REAL |

  DOUBLE PRECISION |

  FLOAT [ ( integer ) ] |

  NUMBER [ ( { integer | *} [, integer]... ) ] |

```

```

{ DECIMAL | NUMERIC } [ ( integer [, integer]. . . ) ] |

{ CHAR | BYTE | GRAPHIC } [ ( integer ) ] |

{ VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [ ( integer ) ] |

LONG VARCHAR |

LONG VARGRAPHIC |

{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
  ( integer [ G | K | M ] ) |

{ XML | XMLTYPE } |

JSON [ ( integer ) ] [ CHARACTER SET { UNICODE | LATIN } ] |

[SYSUDTLIB.] { UDT_name | ST_Geometry | MBR | ARRAY_name
| VARRAY_name }
}

```

RENAME FUNCTION Syntax Elements (External Form)

database_name_1

Containing database for *specific_function_name* if something other than the current database or user.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

specific_function_name

Existing specific function name to be changed.

You cannot rename a table operator function, that is, a function with a parameter style of SQLTABLE. For more information, see the information about table operators for C/C++ and Java user-defined functions in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, method, or Java stored procedure object even if the new object name contains only single-byte characters. Otherwise, Vantage returns an error to the requestor. Instead, use a multibyte session character set.

If you specify *specific_function_name*, then you must also specify *new_specific_function_name*, not *new_function_name*.

new_specific_function_name

New specific function name for the function.

You cannot rename a table operator function. For more information, see the information about table operators for C/C++ and Java user-defined functions in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

database_name_2

Containing database for *function_name* if something other than the current database.

function_name

Existing overloaded calling name of the function to be changed.

You cannot rename a table operator function. For more information, see the information about table operators for C/C++ and Java user-defined functions in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Note:

If the UDF library for your database contains any objects with multibyte characters in the name, you cannot use a single-byte session character set to create a new UDF, UDT, method, or Java stored procedure object even if the new object name contains only single-byte characters. Otherwise, Vantage returns an error to the requestor. Instead, use a multibyte session character set.

If you specify *function_name*, then you must also specify *new_function_name*, not *new_specific_function_name*.

data_type

Data type specifications required to uniquely identify an overloaded function to be renamed.

new_function_name

New function name for the function. For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Note:

You cannot rename a table operator function. For more information, see the information about table operators for C/C++ and Java user-defined functions in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Example 1: Renaming the Specific Name for a Function

This example renames the specific name for a function.

```
RENAME SPECIFIC FUNCTION match_text TO scan_text;
```

Example 2: Renaming the Overloaded Calling Name for a Function

This example renames the overloaded calling name for a function.

```
RENAME FUNCTION imagine_numbers(FLOAT, FLOAT) TO imaginary;
```

Related Information

- [CREATE GLOBAL TEMPORARY TRACE TABLE](#)
- [SET SESSION FUNCTION TRACE](#)
- [SHOW object](#)

Also see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

SHOW FUNCTION MAPPING

Displays the SQL data definition text for the function mapping.

Required Privileges

You must have any privilege on the function mapping or the containing database or user.

SHOW FUNCTION MAPPING Syntax

```
SHOW [IN XML] FUNCTION MAPPING [ database_name. | user_name. ]
function_mapping_name [;]
```

SHOW FUNCTION MAPPING Syntax Elements

IN XML

Displays the function mapping definition in XML format.

The XML schema for the output produced by this option is maintained in:

<http://schemas.teradata.com/dbobject/DBObject.xsd>

database_name

Name of the database containing the function mapping, if other than the current database.

user_name

Name of the user containing the function mapping, if other than the current user.

function_mapping_name

Name of the function mapping for which to show the definition.

Examples

Example: Showing a Function Mapping Definition

This statement displays the definition for the function mapping `Interpolator` in the database `appl_view_db`.

```
SHOW FUNCTION MAPPING appl_view_db.Interpolator;
```

Following is the sample output.

```
CREATE FUNCTION MAPPING appl_view_db.Interpolator
FOR Interpolator SERVER coprocessor
MAP JSON ( '{ "function_version": "1.0" }' )
USING
input_table IN TABLE,
time_table IN TABLE,
count_row_number IN TABLE,
```

```
TimeColumn('periodcol'),
ValueColumns('stockprice');
```

Example: Showing the Definition of a Function Mapping in XML Format

This statement displays the definition in XML format of the function mapping Interpolator in the database appl_view_db.

```
SHOW IN XML FUNCTION MAPPING appl_view_db.Interpolator;
```

Following is the sample output.

```
<TeradataDBObjectSet xmlns="http://schemas.teradata.com/dbobject"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0"
xsi:schemaLocation="http://schemas.teradata.com/dbobject http://
schemas.teradata.com/dbobject/DBObject.xsd"><MappingName dbName="appl_view_db"
name="Interpolator" objId="0:2596" objVer="1" mappingFunction="Interpolator"
remoteServer="coprocessor"><MappingClauseList><ClauseName="TD_MAP_JSON"
value="{ \"function_version\": \"1.0\" }" clauseAttrib="M"/
><ClauseName="input_table" value="" clauseAttrib="I"/><Clause
name="time_table" value="" clauseAttrib="I"/><Clause name="count_row_number"
value="" clauseAttrib="I"/><Clause name="TimeColumn" value=""
clauseAttrib=""><Clause name="ValueColumns" value="" clauseAttrib=""></
MappingClauseList></MappingName><Environment><Server dbRelease="16.50.00.00"
dbVersion="16.20q.00.43alias_d2d" hostName="localhost"/><UseruserId="00000104"
userName="UT1"/><Session charset="ASCII" dateTime="2017-07-24T09:10:39"/></
Environment></TeradataDBObjectSet>
```

User-Defined Method Statements

CREATE METHOD

Defines the body of a method that is associated with a user-defined data type.

See [CREATE TYPE \(Distinct Form\)](#) and [CREATE TYPE \(Structured Form\)](#).

Note:

Customers using Vantage delivered as-a-service cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Required Privileges

You must have the UDTMETHOD privilege on the SYSUDTLIB database to create a method body.

Privileges Granted Automatically

None.

CREATE METHOD Syntax

```
CREATE [ INSTANCE | CONSTRUCTOR ] METHOD [SYSUDTLIB.] method_name
  ( locator_specification [, ...] )
  RETURNS return_data_type [ CAST FROM cast_data_type ] FOR UDT_name
  [ USING GLOP SET GLOP_set_name ]
  EXTERNAL [ NAME { external_method_name | 'item_list [ delimiter... ]' } ]
  [ EXTERNAL SECURITY { DEFINER [ authorization_name ] | INVOKER } ] [;]
```

locator_specification

```
[ parameter_name ] data_type [ AS LOCATOR ]
```

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |
```

```

{ TIME | TIMESTAMP } [ ( fractional_seconds_precision ) ] [WITH TIME
ZONE] |

INTERVAL YEAR [ ( precision ) ] [TO MONTH] |

INTERVAL MONTH [ ( precision ) ] |

INTERVAL DAY [ ( precision ) ]
    [TO { HOUR | MINUTE | SECOND [ ( fractional_seconds_precision ) ] } ] |

INTERVAL HOUR [ ( precision ) ]
    [TO { MINUTE | SECOND [ ( fractional_seconds_precision ) ] } ] |

INTERVAL MINUTE [ ( precision ) ] [TO SECOND
[ ( fractional_seconds_precision ) ] ] |

INTERVAL SECOND [ ( precision [ , fractional_seconds_precision ] ) ] |

PERIOD (DATE) |

PERIOD ( { TIME | TIMESTAMP } [ ( precision ) ] [WITH TIME ZONE] ) |

REAL |

DOUBLE PRECISION |

FLOAT [ ( integer ) ] |

NUMBER [ ( { integer | *} [ , integer ] ... ) ] |

{ DECIMAL | NUMERIC } [ ( integer [ , integer ] ... ) ] |

{ CHAR | BYTE | GRAPHIC } [ ( integer ) ] |

{ VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [ ( integer ) ] |

LONG VARCHAR |

LONG VARGRAPHIC |

{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
    ( integer [ G | K | M ] ) |

```

```

{ XML | XMLTYPE } |

JSON [ ( integer ) ] [ CHARACTER SET { UNICODE | LATIN } ] |

[SYSUDTLIB.] { UDT_name | ST_Geometry | MBR | ARRAY_name
| VARRAY_name }
}

```

item_list

```

{ F delimiter method_entry_name | D | { S | C } S_or_C_item }

```

S_or_C_item

```

{ I delimiter name_on_server delimiter include_name |
  L delimiter library_name |
  O delimiter name_on_server delimiter object_name |
  S delimiter name_on_server delimiter source_name
}

```

CREATE METHOD Syntax Elements

INSTANCE

The object is an instance method.

INSTANCE is the default.

CONSTRUCTOR

The object is a constructor method.

method_name

The calling name for the method.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, or method, even if the new object name contains only single-byte characters. Instead, use a multibyte session character set. Otherwise, Vantage returns an error to the requestor.

The name for a method object must conform to object naming rules. For object naming rules, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

method_name must be unique within the SYSUDTLIB database. You cannot give a method the same name as an existing method or any other database object contained within the SYSUDTLIB database.

method_name must match the spelling and case of its C/C++ method name exactly if you do not specify a *specific_method_name* or *external_method_name*. This applies only to the definition of the method, not to its use.

SQL supports function name overloading within the same method class, so *method_name* does not have to be unique within its class.

Parameter data types and number of parameters are used to distinguish among different methods within the same class that have the same *method_name*.

For more information about function overloading, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

For information about instance methods and constructor methods, see CREATE METHOD in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

return_data_type

The predefined data type or UDT for the value returned by the method.

The method is responsible for providing the returned data with the correct type. If the return type is difficult for the method to create, you should also specify a CAST FROM clause so the system can perform the appropriate data type conversion. For more information about using CAST expressions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

The result type has a dictionary entry in DBC.TVFields under the name RETURN0 [*n*], where *n* is a sequence of digits appended to RETURN0 rows to make each value unique, ensuring that no user-defined parameter names are duplicated. The value of *n* is incremented until it no longer duplicates a parameter name.

If there is no parameter name of RETURN0, then the *n* subscript is not used.

If the external routine for the method is written in C or C++, you can specify TD_ANYTYPE as a parameter data type.

See CREATE FUNCTION or REPLACE FUNCTION in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

This clause is optional.

If you specify a RETURNS clause, it must be the same as the RETURNS clause specified for the corresponding CREATE TYPE.

If you do not specify a RETURNS clause for the method, then the RETURNS clause specified for the corresponding CREATE TYPE applies to this method by default.

cast_data_type

The result type returned by the method that is to be converted to the type specified by the RETURNS clause.

If you specify a CAST FROM clause, there must be an existing cast from the “result cast from” data type to the returns data type. Example:

```
...RETURNS DECIMAL(9,5) CAST FROM FLOAT...
```

If *data type* is a UDT, then the appropriate cast and transform must be defined to handle its conversion. See [CREATE CAST and REPLACE CAST](#) and [CREATE TRANSFORM and REPLACE TRANSFORM](#).

Whenever a LOB that requires data type conversion is passed to a method, the LOB must first be materialized for the conversion to take place.

If *data type* contains a locator indication, then you cannot specify a CAST FROM clause for the method.

UDT_name

Name of the UDT with which the method is associated.

The UDT referenced can be either a distinct UDT or a structured UDT. See CREATE TYPE (Distinct Form) and CREATE TYPE (Structured Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

GLOP_set_name

The name of the GLOP set to associate with this method.

It is not mandatory that the specified GLOP set exist at the time the method is created.

EXTERNAL

The introduction to the mandatory external method body reference clause.

You can specify the following keywords:

- EXTERNAL only.
- EXTERNAL NAME plus an external method name and, optionally, a Parameter Style specification.

external_method_name

The entry point for the method object. This name must be unique within the SYSUDTLIB database.

Case is significant and must match the C or C++ method name.

EXTERNAL SECURITY

Keywords introducing the external security clause.

This clause is mandatory for methods that perform operating system I/O operations.

If you do not specify an external security clause, but the method being defined performs OS I/O, then the results of that I/O are unpredictable. The most likely outcome is crashing the database, and perhaps crashing the entire system.

See `CREATE FUNCTION` or `REPLACE FUNCTION` in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for information about the `EXTERNAL SECURITY` clause and how it is used for UDFs. This information generalizes to all external routines, including methods and external SQL procedures.

See [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#) for information about creating authorizations for external routines.

DEFINER

The method runs in the client user context of the associated security authorization object created for this purpose, which is contained within SYSUDTLIB.

- If you specify an authorization name, you must define an authorization object with that name before you can invoke the method.
- If you do not specify an authorization name, you must define a default `DEFINER` authorization object.

The default authorization object must be defined before a user can run the method.

The system reports a warning if the specified authorization name does not exist at the time the method is created, stating that no authorization name exists.

See [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#).

authorization_name

An optional authorization name.

INVOKER

The method runs in the OS user context with the associated default authorization object that exists for this purpose.

See [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#).

locator_specification

A parenthetical comma-separated list of data types and optional parameter names for the variables to be passed to the function.

The maximum number of parameters a method accepts is 128.

data_type AS LOCATOR

BLOB and CLOB types must be represented by a locator. For a description of locators, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

You cannot specify in-memory LOB parameters. An AS LOCATOR phrase must be specified for each LOB parameter and return value.

parameter_name data_type

Whenever an LOB that requires data type conversion is passed to a method, the LOB must first be materialized for the conversion to take place.

You must specify opening and closing parentheses even if no parameters are to be passed to the function.

parameter_name data_type AS LOCATOR

If you specify one parameter name, then you must specify names for all the parameters to be passed.

If you do not specify parameter names, the system assigns unique names to them in the form *P1*, *P2*, ..., *Pn*. These names are used in the COMMENT statement, displayed in the report produced by the HELP METHOD statement, and appear in the text of error messages.

See [COMMENT \(Comment Placing Form\)](#).

See [HELP METHOD](#).

The data type associated with each parameter is the type of the parameter or returned value. All Vantage predefined data types and UDTs (except for the DYNAMIC type) are valid. If you specify a UDT, then the current user of the method must have one of the following privilege sets:

- UDTMETHOD, UDTTYPE, or UDTUSAGE on the SYSUDTLIB database.
- UDTUSAGE on the specified UDT.

Character data can also specify a CHARACTER SET clause.

For data types that take a length or size specification, like BYTE, CHARACTER, DECIMAL, VARCHAR, and so on, the size of the parameter indicates the largest number of bytes that can be passed.

You cannot specify VARIANT_TYPE as a parameter data type for a method.

If the external routine for the method is written in C or C++, you can specify TD_ANYTYPE as a parameter data type.

item_list

A string that specifies the source and object components needed to build the method.

Depending on the initial code in the sequence, the string specifies either the C/C++ object name for the method or an encoded name or path for the components needed to create the method.

The following list briefly documents the path specifications for the external method. The character `;` represents an arbitrary user-defined delimiter.

You must use the same delimiter throughout the string specification.

You can specify the following file types as external string literals.

- Method object
- Include
- Library
- Object
- Package
- Source

For more information about these file types, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

F

The string that follows is the entry point name of the C or C++ method object.

D

Enables symbolic debugging for the UDM, which shows source code and displays variables by name. Without this option, UDMs can only be debugged at the machine instruction level. You should always specify this option for debugging purposes when UDMs are being tested. This option adds `-g` to the C compiler command line. See [SET SESSION DEBUG FUNCTION](#) and the information about C/C++ command-line debugging for UDFs in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

The D option only applies only to C and C++ code.

Note:

You should not use this option when installing debugged UDMs on production system because it increases the size of the UDM library.

S

The source or object code is stored on the server and the string that follows is the path to its location.

C

The source or object code is stored on the client and the string that follows is the path to its location.

CREATE METHOD Examples

Example: Creating a Constructor Method for a UDT

The following example creates the definition for a constructor method named *address* for the UDT named *address*. The external routine for this method is stored on the Teradata platform at the location specified in the EXTERNAL NAME clause.

```
CREATE CONSTRUCTOR METHOD address( VARCHAR(20), CHARACTER(5) )
  RETURNS address
  FOR address
  EXTERNAL NAME
    'SO!C:\structured_lib\addr_cons.obj!F!addr_constructor';
```

Example: Creating a Method Named in_state for a UDT Named address

The following example creates the definition for an instance method named *in_state* for a UDT named *address*. The external routine for this method is stored at the location specified in the EXTERNAL NAME clause.

```
CREATE METHOD in_state( CHARACTER(2) )
  RETURNS CHARACTER(1)
  FOR address
  EXTERNAL NAME 'SO!C:\structured_lib\addr_in_state.c!F!in_state';
```

Example: Creating a Method Named timezone for a UDT Named address

The following example creates the definition for an instance method named *timezone* for the UDT named *address*. The external routine for this method is stored on the Teradata platform at the location specified in the EXTERNAL NAME clause.

```
CREATE METHOD timezone( address )
  RETURNS CHARACTER(3)
  FOR address
  EXTERNAL NAME 'SO!C:\structured_lib\addr_timezone.c!F!in_state';
```

Example: Creating an Instance Method Using a One-Dimensional ARRAY Type and an SQL Parameter Type

This example creates an instance method named *update_phone* with an SQL parameter type using a one-dimensional ARRAY data type that has 5 elements.

First create the ARRAY type with CHARACTER element type named *phonenumbers_ary*.

```
CREATE TYPE phonenumbers_ary
  AS CHARACTER(10) ARRAY[5];
```

Then create the structured UDT named *address_udt*.

```
CREATE TYPE address_udt AS (
  house_num INTEGER,
  street VARCHAR(100),
  phone CHAR(10))
  INSTANCE METHOD update_phone (
    a1 source_ary)
  ...
  ;
```

Finally, create the instance method *update_phone* for *address_udt* with an SQL parameter style.

```
CREATE INSTANCE METHOD update_phone (
  p1 phonenumbers_ary)
  FOR address_udt
  RETURNS INTEGER
  NO SQL
  PARAMETER STYLE SQL
  DETERMINISTIC
  LANGUAGE C
  EXTERNAL NAME 'CS!update_phone!update_phone.c!F!update_phone';

void update_phone (
  UDT_HANDLE      *thisUdt,
  ARRAY_HANDLE    *aryval,
  INTEGER         *result,
```

```

        int          *indicator_this,
        int          *indicator_aryval,
        int          *indicator_result,
        char         sqlstate[6],
        SQL_TEXT     extname[129],
        SQL_TEXT     specific_name[129],
        SQL_TEXT     error_message[257])
{
/* body function */
}

```

Example: Creating an Instance Method Using a One-Dimensional ARRAY Type and a TD_GENERAL Parameter Type

This example duplicates [Example: Creating an Instance Method Using a One-Dimensional ARRAY Type and an SQL Parameter Type](#) except that for this example, the instance method *update_info* is written with a TD_GENERAL parameter style.

```

CREATE INSTANCE METHOD update_phone (
    p1 phonenumbers_ary)
RETURNS INTEGER
FOR address_udt
NO SQL
PARAMETER STYLE TD_GENERAL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!update_phone!update_phone.c!F!update_phone';

void UpdatePhone (
    UDT_HANDLE      *thisUdt,
    ARRAY_HANDLE     *aryval,
    INTEGER          *result,
    char             sqlstate[6])
{
/* body function */
}

```

Example: Creating an Instance Method Using a Multidimensional ARRAY Type and an SQL Parameter Type

The following is an example of creating a UDM for a structured UDT with a single parameter which is defined using an n-D ARRAY data type. The same example is repeated in [Example: Creating an Instance Method Using a Multidimensional ARRAY Type and a TD_GENERAL Parameter Type](#) to show the DDL and function prototype with both SQL and TD_GENERAL parameter styles.

First create the structured type *measures_udt*.

```
CREATE TYPE measures_udt AS (
  amplitud  INTEGER,
  phase     INTEGER,
  frequency INTEGER)
INSTANCE METHOD update_measures (
  a1 source_ary)
...
;
```

Then create the multidimensional ARRAY type *source_ary*.

```
CREATE TYPE source_ary AS
  INTEGER ARRAY [1:5][1:7][1:20];
```

Finally, create the instance method *update_measures* for type *measures_udt* with a single multidimensional ARRAY parameter.

```
CREATE INSTANCE METHOD update_measures (
  a1 source_ary)
  RETURNS INTEGER
  FOR measures_udt
  NO SQL
  PARAMETER STYLE SQL
  DETERMINISTIC
  LANGUAGE C
  EXTERNAL NAME 'CS!update_measures!update_measures.c!F!update_measures';

void update_measures (
  UDT_HANDLE      *thisUdt,
  ARRAY_HANDLE    *aryval,
  INTEGER          *result,
  int              *indicator_this,
  int              *indicator_aryval,
```

```

int          *indicator_result,
char         sqlstate[6],
SQL_TEXT     extname[129],
SQL_TEXT     specific_name[129],
SQL_TEXT     error_message[257])
{
/* body function */
}

```

Example: Creating an Instance Method Using a Multidimensional ARRAY Type and a TD_GENERAL Parameter Type

This example duplicates [Example: Creating an Instance Method Using a Multidimensional ARRAY Type and an SQL Parameter Type](#) except that for this example, the instance method *update_measures* is written with a TD_GENERAL parameter style.

```

CREATE INSTANCE METHOD update_measures
  ( a1 source_ary)
  RETURNS INTEGER
  FOR MEASURES_UDT
  NO SQL
  PARAMETER STYLE TD_GENERAL
  DETERMINISTIC
  LANGUAGE C
  EXTERNAL NAME 'CS!update_measures!update_measures.c!F!update_measures';

void update_measures (
  UDT_HANDLE      *thisUdt,
  ARRAY_HANDLE     *aryval,
  INTEGER          *result,
  char             sqlstate[6])
{
/* body function */
}

```

Example: Methods That Return a TD_ANYTYPE or INTEGER Data Type

Method *td_anytyp_x1* takes a TD_ANYTYPE parameter and returns an INTEGER.

```

CREATE METHOD td_anytyp_x1 (p1 TD_ANYTYPE)
  RETURNS INTEGER

```



```
FOR myUDT
EXTERNAL NAME 'CS!td_anytyp_x1!td_anytyp_x1.c!F!td_anytyp_x1';
```

Method *td_anytpe_x2* takes a TD_ANYTYPE parameter and returns a TD_ANYTYPE.

```
CREATE METHOD td_anytpe_x2(p1 TD_ANYTYPE)
RETURNS TD_ANYTYPE
FOR myUDT
EXTERNAL NAME 'CS!td_anytpe_x2!td_anytpe_x2.c!F!td_anytpe_x2';
```

Example: Creating an Instance Method for a UDT Stored on the Client

The following example creates the definition for an instance method named *toUS* for the UDT named *euro*. The external routine for this method is stored on the client at the location specified in the EXTERNAL NAME clause.

```
CREATE METHOD toUS()
RETURNS us_dollar CAST FROM DECIMAL(8,2)
FOR euro
EXTERNAL NAME 'CO!C:\sys\math.lib!CO!C:\distinct_lib\euro2us.obj!F!toUS';
```

Related Information

- [ALTER TYPE](#)
- [CREATE TYPE \(Distinct Form\)](#)
- [CREATE TYPE \(Structured Form\)](#)
- [SHOW object](#)

Teradata Vantage™ - SQL External Routine Programming, B035-1147 explains how to write the external routine for a method.

Teradata Vantage™ - JSON Data Type, B035-1150 documents the dot notation used to invoke methods within SQL expressions.

ALTER METHOD

Performs either or both of the following functions.

- Controls whether an existing method can run in protected mode as a separate process or in non-protected mode as part of the database.
- Recompiles or relinks the method and redistributes it.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

To alter a method definition, you must have the UDTMETHOD privilege on the database SYSUDTLIB.

There are no privileges granted automatically.

ALTER METHOD Syntax

```
ALTER { specific_method_clause | method_clause }
      FOR UDT_name { EXECUTE [NOT] PROTECTED | COMPILE [ONLY] } [;]
```

specific_method_clause

```
SPECIFIC METHOD [SYSUDTLIB.]specific_method_name [SYSUDTLIB.]
```

method_clause

```
[ INSTANCE | CONSTRUCTOR ] METHOD [SYSUDTLIB.]method_name
[ ( { data_type | [SYSUDTLIB.]UDT_name } [,...] ) ]
```

ALTER METHOD Syntax Elements

UDT_name

See ALTER PROCEDURE (External Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

EXECUTE PROTECTED

Change the execution mode for the specified method from unprotected mode to protected mode.

EXECUTE NOT PROTECTED

Change the execution mode for the specified method from protected mode to unprotected mode.

COMPILE

Recompile the code source for the specified method, generate the object code, recreate the .so file, and distribute the file to all nodes of the system.

The existing method object is replaced by the recompiled version.

ONLY

Recompile the method only. When you specify this option, a new dynamic linked library is not distributed to database nodes.

When you load a method onto another platform of a different type, the system marks the method as not valid, and the method must be recompiled. If there are many methods in one database, you can save time by specifying the ONLY option for all recompilations to avoid having to generate and distribute a new library, until the last one is compiled in that database without the ONLY option.

specific_method_clause

The following character string is the specific method name for the method to be altered.

SYSUDTLIB.*specific_method_name*

The specific function name for the method to be altered.

The SYSUDTLIB database must contain a method with the same specific method name.

The specification of SYSUDTLIB is optional.

method_clause

The method to be altered.

INSTANCE

that the method named as *method_name* is an instance method.

This is the default.

CONSTRUCTOR

The method named as *method_name* is a constructor method.

SYSUDTLIB.*method_name*

The name (not the specific method name) for the method to be altered.

There must already be a method associated with the specified UDT name that has the same signature.

The specification of SYSUDTLIB is optional.

data_type

An optional data type specification for the parameters passed to *method_name*.

If you do not specify a data type list, then the specified method must be the only one with that name in the SYSUDTLIB database.

ALTER METHOD Examples

Example: Changing the Protection Mode

The following ALTER METHOD request changes the protection mode for the method named `polygon_mbr()` from protected to not protected:

```
ALTER METHOD polygon_mbr() FOR SYSUDTLIB.polygon
EXECUTE NOT PROTECTED;
```

Example: Recompiling a Method Object

The following ALTER METHOD request recompiles the specific method object named `SYSUDTLIB.polygon_mbr`:

```
ALTER SPECIFIC METHOD SYSUDTLIB.polygon_mbr COMPILE;
```

REPLACE METHOD

Creates or replaces an existing method definition.

If a body has not already been defined for the specified method name, and if the UDT the method is associated with has already been created, creates a new method body for it.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the UDTMETHOD privilege on the SYSUDTLIB database to replace a method.

REPLACE METHOD Syntax

```
REPLACE { specific_method_clause | method_clause }
EXTERNAL [ NAME { external_method_name | , item_list [ delimiter... ] , } ] [;]
```

specific_method_clause

```
SPECIFIC METHOD [SYSUDTLIB.]specific_method_name
```

method_clause

```
[ INSTANCE | CONSTRUCTOR ] METHOD [SYSUDTLIB.]method_name
  ( [ parameter_name ] { data_type | [SYSUDTLIB.]UDT_name }
  [ AS LOCATOR ] )
  FOR [SYSUDTLIB.] UDT_name
```

item_list

```
{ S | C } S_or_C_item
```

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [ (fractional_seconds_precision) ] [WITH TIME
ZONE] |

  INTERVAL YEAR [(precision)] [TO MONTH] |

  INTERVAL MONTH [(precision)] |

  INTERVAL DAY [(precision)]
  [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] }] |

  INTERVAL HOUR [(precision)]
  [TO { MINUTE | SECOND [(fractional_seconds_precision)] }] |

  INTERVAL MINUTE [(precision)] [TO SECOND
[(fractional_seconds_precision)] ] |

  INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) |

  PERIOD (DATE) |

  PERIOD ( { TIME | TIMESTAMP } [(precision)] [WITH TIME ZONE] ) |
```

```

REAL |

DOUBLE PRECISION |

FLOAT [ ( integer ) ] |

NUMBER [ ( { integer | *} [, integer ]... ) ] |

{ DECIMAL | NUMERIC } [ ( integer [, integer ]... ) ] |

{ CHAR | BYTE | GRAPHIC } [ ( integer ) ] |

{ VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [ ( integer ) ] |

LONG VARCHAR |

LONG VARGRAPHIC |

{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
  ( integer [ G | K | M ] ) |

{ XML | XMLTYPE } |

JSON [ ( integer ) ] [ CHARACTER SET { UNICODE | LATIN } ] |

[SYSUDTLIB.] { UDT_name | ST_Geometry | MBR | ARRAY_name
| VARRAY_name }
}

```

S_or_C_item

```

{ I delimiter name_on_server delimiter include_name |
  L delimiter library_name |
  O delimiter name_on_server delimiter object_name |
  S delimiter name_on_server delimiter source_name
}

```

REPLACE METHOD Syntax Elements**EXTERNAL**

Introduction to the mandatory external method body reference clause.

This clause can specify either:

- The keyword **EXTERNAL** only.
- The keywords **EXTERNAL NAME** plus an external method name, with optional Parameter Style specification

external_method_name

Entry point for the method object. This name must be unique within the SYSUDTLIB database.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Case is significant and must match the C or C++ method name.

specific_method_clause

This clause is mandatory for overloaded *method_names*, but is otherwise optional and can only be specified once per method definition.

SYSUDTLIB

Unlike *method_name*, *specific_method_name* must be unique within SYSUDTLIB.

specific_method_name

Specific name for the method.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

This name is stored in DBC.TVM as the name of the method database object.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, or method, even if the new object name contains only single-byte characters. Otherwise, Vantage returns an error to the requestor. Instead, use a multibyte session character set.

method_clause

Calling name for the method.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, method, even if the new object name contains only single-byte characters. Instead, use a multibyte session character set. For related information about the following topics, see the external forms of CREATE FUNCTION and REPLACE FUNCTION in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 :

- Function identifiers
- Function names
- Function calling arguments
- Function name overloading
- Parameter names and data types

INSTANCE

The object is an instance method.

INSTANCE is the default.

CONSTRUCTOR

The object is a constructor method.

SYSUDTLIB

method_name must be unique within the SYSUDTLIB database. You cannot give a method the same name as an existing method or any other database object contained within the SYSUDTLIB database.

method_name

method_name must match the spelling and case of its C/C++ method name exactly if you do not specify a *specific_method_name* or *external_method_name*. This applies only to the definition of the method, not to its use.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Parameter data types and number of parameters are used to distinguish among different methods within the same class that have the same *method_name*.

For further information about function overloading, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

parameter_name

The maximum number of parameters a method accepts is 128.

You must specify opening and closing parentheses even if no parameters are to be passed to the function.

If you specify one parameter name, then you must specify names for all the parameters passed to the function.

If you do not specify parameter names, the system assigns unique names to them in the form P1, P2, ..., P *n*. These names are used in the COMMENT statement, displayed in the report produced by the HELP METHOD statement, and appear in the text of error messages. See [COMMENT \(Comment Placing Form\)](#) and [HELP METHOD](#).

data_type

A parenthetical comma-separated list of data types and optional parameter names for the variables to be passed to the function.

The data type associated with each parameter is the type of the parameter or returned value. All Vantage predefined data types and UDTs are valid.

Character data can also specify a CHARACTER SET clause.

For data types that take a length or size specification, like BYTE, CHARACTER, DECIMAL, VARCHAR, and so on, the size of the parameter indicates the largest number of bytes that can be passed.

UDT_name

If you specify a UDT, the current user of the method must have one of the following privilege sets:

- UDTMETHOD, UDTTYPE, or UDTUSAGE on the SYSUDTLIB database.
- UDTUSAGE on the specified UDT.

AS LOCATOR

BLOB and CLOB types must be represented by a locator. Vantage does not support in-memory LOB parameters. You must specify an AS LOCATOR phrase for each LOB parameter and return value. For a description of locators, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Note, however, that whenever a LOB that requires data type conversion is passed to a method, the LOB must be materialized for the conversion to take place.

FOR [SYSUDTLIB.]*UDT_name*

Name of the UDT to which this method applies.

Do not specify this clause for a specific method.

item_list

C

The source or object code for the method is stored on the client and the string that follows is the path to its location.

S

The source or object code for the method is stored on the server and the string that follows is the path to its location.

S_or_C_item

I delimiter name_on_server delimiter include_name

Include.

O delimiter name_on_server delimiter object_name

Library.

Oname_on_server |object_name

Object.

S delimiter name_on_server delimiter source_name

Source.

Usage Notes

How REPLACE METHOD And CREATE METHOD Differ

You can perform CREATE METHOD only once to declare the body definition for a method. If you try to use CREATE METHOD a second time to define the body for an existing method, the system returns an error to the requestor.

Instead of using CREATE METHOD to redefine the method body, you must use REPLACE METHOD. REPLACE METHOD can be performed multiple times to replace the body definition for a method.

Example: Replacing a Method Definition Using Its Specific Method Name

The following example replaces the existing method with the specific method name *polygon_mbr* with a new definition.

```
REPLACE SPECIFIC METHOD SYSUDTLIB.polygon_mbr
EXTERNAL 'CO!C:\sys\spatial.lib!CO!C:\udt_lib\polymbr.obj!F!toUS'
```

Related Information

- [ALTER FUNCTION](#)
- [ALTER METHOD](#)
- [ALTER TYPE](#)
- [CREATE CAST and REPLACE CAST](#)
- [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#)
- [CREATE ORDERING and REPLACE ORDERING](#)
- [CREATE TRANSFORM and REPLACE TRANSFORM](#)
- [CREATE TYPE \(Distinct Form\)](#)
- [CREATE TYPE \(Structured Form\)](#)
- [DROP FUNCTION](#)
- [DROP ORDERING](#)
- [DROP TRANSFORM](#)
- [DROP TYPE](#)
- [RENAME FUNCTION \(External Form\)](#)
- [HELP CAST](#)
- [HELP FUNCTION](#)
- [HELP TRANSFORM](#)
- [HELP TYPE](#)
- [SHOW object](#)

See *Teradata Vantage™ - SQL Data Control Language*, B035-1149 for information about the UDTTYPE, UDTMETHOD, UDTUSAGE, and EXECUTE FUNCTION privileges, particularly as described for “GRANT (SQL Form)”, and especially the topics “UDT-Related Privileges” and “EXECUTE FUNCTION Privilege.”

Also see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about how to write user-defined external routines.

HELP METHOD

Displays the parameter list of the specified method.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

To perform `HELP METHOD`, you must have one or more of the following privileges:

- At least one privilege on the `SYSUDTLIB` database.
- UDT usage on `UDT_name`.

Use the `SHOW` privilege to enable a user to perform `HELP` or `SHOW` requests only against a specified method.

HELP METHOD Syntax

```
HELP { method_clause | specific_method_clause } [;]
```

method_clause

```
[ INSTANCE | CONSTRUCTOR ] METHOD [database_name.] method_name
  [ ( user_defined_type_name [,...] ) ] FOR user_defined_type_name
```

specific_method_clause

```
SPECIFIC METHOD [database_name.]specific_method_name
```

HELP METHOD Syntax Elements

INSTANCE

Returns parameter information for the instance method named *method_name*.
If you do not specify `INSTANCE` or `CONSTRUCTOR`, then `INSTANCE` is assumed by default.

CONSTRUCTOR

Returns parameter information for the constructor method named *method_name*.
If you do not specify `INSTANCE` or `CONSTRUCTOR`, then `INSTANCE` is assumed by default.

method_name

Name of the method whose parameter information is to be returned.

user_defined_type_name

Name of the UDT with which the specified method is associated.

specific_method_name

Name of the specific method whose parameter information is to be returned.

HELP METHOD Examples

Example: HELP METHOD with Sample Output

The following example reports the parameter list of the method named *int_ordering* that is associated with the UDT named *udt_int*.

```

HELP METHOD int_ordering FOR udt_int;
  Parameter Name SELF
    Type UT
    Comment ?
    Nullable Y
    Format ?
    Max Length 0
  Decimal Total Digits ?
  Decimal Fractional Digits ?
    Table/View? M
    Char Type ?
  Parameter Type I
    UDT Name UDTINT
  Parameter Name RETURN0
    Type I
    Comment ?
    Nullable Y
    Format -(10)9
    Max Length 4
  Decimal Total Digits ?
  Decimal Fractional Digits ?
    Table/View? M
    Char Type ?
  Parameter Type 0
    UDT Name ?

```

Example: TD_ANYTYPE Input Parameter Data Type

Suppose you create a method named *method_1*. The following SQL text is a partial definition for *method_1*.

```
CREATE INSTANCE METHOD method_1 (
  p1 TD_ANYTYPE)
FOR SYSUDTLIB ... method_1
EXTERNAL NAME ... ;
```

When you submit a HELP METHOD request on *method_1*, the output looks like this.

Note that the specifications of TD_ANYTYPE as the parameter data type for p1 is returned as ++ in the report returned by this HELP METHOD request.

```
HELP METHOD method_1;
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.
```

Parameter Name	Type	Comment
SELF	UT	?
P1	++	?
RETURN0	I	?

Example: HELP METHOD

The following example requests information for the SPECIFIC method name *polygon_mbr*.

```
HELP SPECIFIC METHOD SYSUDTLIB.polygon_mbr;
```

The following example requests information about the method name *area* for the UDT circle.

```
HELP METHOD SYSUDTLIB.area() FOR circle;
```

The following example requests information about the method name *in_state* for the UDT address.

```
HELP METHOD in_state( CHAR(2) ) FOR address;
```

Related Information

- [CREATE METHOD](#)

- *Teradata Vantage™ - SQL External Routine Programming, B035-1147*

User-Defined Type Statements

CREATE TYPE (Structured Form)

Creates the body of a structured user-defined data type.

Note:

Customers using Vantage delivered as-a-service cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

Required Privileges

You must have either the UDTTYPE or UDTMETHOD privilege on the SYSUDTLIB database to create a distinct UDT.

If the type definition includes a set of method signatures, then you must also have the UDTMETHOD privilege on the SYSUDTLIB database.

Privileges Granted Automatically

None.

CREATE TYPE Syntax (Structured Form)

```
CREATE TYPE [ SYSUDTLIB. ] UDT_name AS ( attribute_specification [,...] )
[ INSTANTIABLE ] NOT FINAL [ method_specification [,...] ] [;]
```

attribute_specification

```
attribute_name {
  predefined_data_type [ CHARACTER SET server_character_set ] |
  UDT_name
}
```


method_specification

```
[ INSTANCE | CONSTRUCTOR ] METHOD [ SYSUDTLIB. ] method_name
( parameter_specification [, ...] )
RETURNS returns_parameter_specification [ AS LOCATOR ]
[ CAST FROM { data_type | [ SYSUDTLIB. ] UDT_name }
[ AS LOCATOR ] ]
[ SPECIFIC [ SYSUDTLIB. ] specific_method_name ]
[ SELF AS RESULT ]
language_and_access_specification
type_attribute [, ...]
```

Note:

You can specify *language_and_access_specification* and *type_attribute* in the reverse order.

parameter_specification

```
[ parameter_name ] {
  data_type [ CHARACTER SET server_character_set ] |
  [ SYSUDTLIB. ] UDT_name
} [ AS LOCATOR ]
```

returns_parameter_specification

```
data_type [ CHARACTER SET server_character_set ] |
[ SYSUDTLIB. ] UDT_name
} [ AS LOCATOR ]
```

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [( fractional_seconds_precision)] [WITH TIME
ZONE] |

  INTERVAL YEAR [( precision)] [TO MONTH] |

  INTERVAL MONTH [( precision)] |
```

```

INTERVAL DAY [( precision)]
    [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] } ] |

INTERVAL HOUR [(precision)]
    [TO { MINUTE | SECOND [(fractional_seconds_precision)] } ] |

INTERVAL MINUTE [(precision)] [ TO SECOND
[(fractional_seconds_precision)] ] |

INTERVAL SECOND [ ( precision [, fractional_seconds_precision] ) |

PERIOD (DATE) |

PERIOD ({ TIME | TIMESTAMP } [(precision)] [ WITH TIME ZONE ]) |

REAL |

DOUBLE PRECISION |

FLOAT [(integer)] |

NUMBER [( { integer | *} [, integer ]...)] |

{ DECIMAL | NUMERIC } [(integer [, integer ]...)] |

{ CHAR | BYTE | GRAPHIC } [(integer)] |

{ VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [(integer)] |

LONG VARCHAR |

LONG VARGRAPHIC |

{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
(integer [ G | K | M ]) |

[SYSUDTLIB.] { XML | XMLTYPE } [(integer [ G | K | M ])]
[ INLINE LENGTH integer ] |

[SYSUDTLIB.] JSON [(integer [ K | M ])] [ INLINE LENGTH integer ]
[ CHARACTER SET { UNICODE | LATIN } ] |

[SYSUDTLIB.] ST_GEOMETRY [(integer [ K | M ])] [ INLINE LENGTH

```

```
integer ] |

[SYSUDTLIB.] { UDT_name | MBR }
}
```

language_and_access_specification

```
language_clause SQL_data_access
```

Note:

If *language_and_access_specification* is before *type_attribute*, you can specify *language_clause* and *SQL_data_access* in the reverse order.

type_attribute

```
{ SPECIFIC [ SYSUDTLIB. ] specific_method_name |
  PARAMETER STYLE { SQL | TD_GENERAL } |
  [NOT] DETERMINISTIC |
  CALLED ON NULL INPUT |
  RETURNS NULL ON NULL INPUT
}
```

CREATE TYPE Syntax Elements (Structured Form)

SYSUDTLIB

The containing database for *UDT_name*. If omitted, the containing database is SYSUDTLIB.

You cannot create a UDF within the SYSUDTLIB database with the same specific name or same routine signature.

If such a UDF already exists in SYSUDTLIB at the time you submit a CREATE TYPE request, the request fails because a routine with the same signature already exists.

UDT_name

Name of the structured UDT to be created.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, method, or Java stored procedure object even if the new object name contains only single-byte characters. Otherwise, the system returns an error to the requestor. Instead, use a multibyte session character set.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

Note that the creation of a structured type consumes two names in the *TVMName/* name space for SYSUDTLIB:

- The first name consumed corresponds to the name of the UDT itself.
- The second name corresponds to the system-generated constructor function for the UDT.

The system constructs this name as explained in the topic on naming conventions.

INSTANTIABLE

An optional keyword specification that makes it possible to create a constructor method for the type, enabling you to instantiate a value having that type.

INSTANTIABLE is the default and only valid specification.

Vantage does not support a NOT INSTANTIABLE option.

NOT FINAL

A mandatory keyword sequence required for the definition of all structured UDTs.

The NOT FINAL keywords indicate that you can create subtypes of the type being defined.

attribute_specification***attribute_name***

Name of an attribute of the structured UDT.

predefined_data_type

The predefined data type on which the current attribute of the structured UDT is based, if that attribute is not a UDT.

You can specify any Teradata data type, except for ARRAY, VARRAY, and DATASET.

server_character_set

If *data_type* is a character type, this is the server character set used by *data_type*. Otherwise, this option is not used.

The CHARACTER SET clause initiates the following system-generated routines when the system creates the *UDF_name* distinct type:

- The system-generated CAST routines are registered to cast to and cast from a character source type of the specified server character set.
- The system-generated ORDERING routine is registered to map to a character source type of the specified server character set.
- The system-generated tosql and fromsql TRANSFORM routines are registered to transform the UDT to and from a character source type of the specified server character set.

If you know that all database transactions occur in languages that can be expressed using the LATIN character set, specify LATIN. Otherwise, for example, if some or all transactions use non-LATIN or multibyte characters, specify UNICODE. See *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Note:

You cannot specify a character server data set of KANJI1.

(UDT_name)

Name of a UDT on which the current attribute of the structured UDT is based if that attribute is not a predefined data type.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

method_specification**INSTANCE**

The method signature being defined is an instance method.

This is the default.

CONSTRUCTOR

The method signature being defined is a constructor method.

SYSUDTLIB

An optional keyword that indicates that *method_name* is being created in the SYSUDTLIB database.

All methods must be created within the SYSUDTLIB database.

method_name

Name for the method whose signature being added to the type definition for *UDT_name*.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

AS LOCATOR

Specifies that a BLOB or CLOB parameter type is represented by a locator. For a description of locators, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146. LOB parameters must always be represented by a locator. Vantage does not support in-memory LOB parameters: an AS LOCATOR phrase must be specified for each LOB parameter and return value.

Whenever a LOB that requires data type conversion is passed to a method, the LOB must be materialized for the conversion to take place.

CAST FROM

The result type returned by the external function that is to be converted to the type specified by the RETURNS clause. Example:

```
...RETURNS DECIMAL(9,5) CAST FROM FLOAT...
```

data_type

Whenever a LOB that requires data type conversion is passed to a method, the LOB must first be materialized for the conversion to take place.

UDT_name

Name of the UDT.

specific_method_name

Specific name of the method whose signature is being added to the type definition for *UDT_name*.

SELF AS RESULT

The method is type-preserving.

If you specify SELF AS RESULT, then the data type specified in the RETURNS clause for the method must have the same name as *UDT_name*.

parameter_specification

parameter_name

A set of parameter names for the method.

The parameter list is a parenthetical comma-separated list of data types, including UDTs, and optional parameter names and locators for the variables to be passed to the method.

Parameter names must be unique within a method definition.

You cannot use the keyword SELF to name method parameters.

The maximum number of parameters a method accepts is 128.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

You must specify opening and closing parentheses even if no parameters are to be passed to the method.

If you specify one parameter name, then you must specify names for all the parameters passed to the method.

If you do not specify parameter names, the system assigns unique names to them in the form P1, P2, ..., Pn. These names are used in the COMMENT statement (see [COMMENT \(Comment Placing Form\)](#)), displayed in the report produced by the HELP METHOD statement in the PARAMETER NAME column (see [HELP METHOD](#)), and appear in the text of error messages.

data_type

Whether you specify a list of parameter names or not, you must specify a set of data types for the parameters specified.

- If a parameter in the *parameter_name* list has a predefined data type, the *data_type* entry is that data type.
- If the type specified as *data_type* is a character type, the *server_character_set* entry is its server character set.

The data type associated with each parameter is the type of the parameter or returned value. All Vantage data types are valid. Distinct and structured UDTs are valid for scalar UDFs only.

For data types that take a length or size specification, like BYTE, CHARACTER, DECIMAL, VARCHAR, and so on, the size of the parameter indicates the largest number of bytes that can be passed.

server_character_set

Character data can also specify a CHARACTER SET clause.

You cannot specify a character server data set of KANJI1. Otherwise, the system returns an error to the requestor.

UDT_name

If a parameter in the *parameter_name* list has a UDT type, the *UDT_name* entry is the name of that UDT.

returns_parameter_specification***data_type***

Name of the data type returned by the method, which can be either a predefined type or a UDT.

server_character_set

If the returned data has a character type, you can also specify its server character set.

You cannot specify a character server data set of KANJI1. Otherwise, the system returns an error to the requestor.

UDT_name

Name of the UDT.

SYSUDTLIB

Optional containing database name, which is always SYSUDTLIB.

type_attribute**SQL**

The parameter style for the method defined by this signature is SQL.

TD_GENERAL

The parameter style for the method defined by this signature is TD_GENERAL.

DETERMINISTIC

The result of invoking the method defined by this signature is deterministic.

NOT DETERMINISTIC

The result of invoking the method defined by this signature is not deterministic.

CALLED ON NULL INPUT

The method defined by this signature is called if any of the arguments passed to is null.

RETURNS NULL ON NULL INPUT

The method defined by this signature is not called if any of the arguments passed to it is null. Instead, it returns a null.

language_clause

The language used to write the source code for the method defined by this signature.

This clause is mandatory for each method signature you specify as part of a distinct UDT definition.

The valid languages for writing methods are C and C++.

- If the method is written in C, then its language code is C.
- If the method is written in C++, then its language code is CPP.

The language clause must be specified as either LANGUAGE C or LANGUAGE CPP even if the external method routine is supplied in object form.

If the external method object is not written in C or C++, it must be compatible with C or C++ object code.

SQL_data_access

Specifies whether SQL statements are permitted within the user-written external routine for the method defined by this signature.

The only valid specification is NO SQL.

Examples**Example: Creating a Structured UDT with 2 Attributes**

The following example creates a structured UDT named *address* that has two attributes defined with predefined data types: *street* and *zip*. This type is also defined with one constructor method (*address* (VARCHAR(20), CHAR(5))) and three instance methods (*city()* , *state()* , and *in_state()*).

Because no SPECIFIC method names are specified for the instance methods *city()* and *in_state(CHAR(2))* in this example, the system autogenerated and registers the following *TVMName/* SPECIFIC names for them in the SYSUDTLIB database:

- *ADDRESS_CITY_R* on behalf of the *city()* method.
- *ADDRESS_IN_STATE_R* on behalf of the *in_state(CHAR(2))* method.

```
CREATE TYPE address AS (
    street VARCHAR(20),
    zip    CHARACTER(5) )
NOT FINAL
CONSTRUCTOR METHOD address( VARCHAR(20), CHARACTER(5) )
RETURNS address
SPECIFIC address_constructor_1
SELF AS RESULT
LANGUAGE C
PARAMETER STYLE SQL
DETERMINISTIC
NO SQL,
INSTANCE METHOD city()
RETURNS VARCHAR(20)
LANGUAGE C
PARAMETER STYLE SQL
DETERMINISTIC
NO SQL,
METHOD state()
RETURNS CHARACTER(2)
SPECIFIC address_state
RETURNS NULL ON NULL INPUT
LANGUAGE C
PARAMETER STYLE SQL
DETERMINISTIC
NO SQL,
METHOD in_state(CHARACTER(2))
RETURNS CHARACTER(1)
LANGUAGE C
PARAMETER STYLE SQL
DETERMINISTIC
NO SQL;
```

Example: Creating a Structured UDT from Predefined Data Types

This example creates a structured UDT named *person* from four predefined data types and the structured UDT *address*, making it a 2-level nested type. The UDT is nested because the type *address* is a structured

UDT having two attributes. In this case, both of those attributes are predefined types, but if one or more of them had been a structured type, *person* would be a 3-level nested type.

```
CREATE TYPE SYSUDTLIB.person AS (
    ssn          CHARACTER(11),
    first_name    VARCHAR(20),
    middle_name   VARCHAR(20),
    last_name     VARCHAR(20),
    domicile      address )
NOT FINAL;
```

Related Information

- [CREATE TYPE \(ARRAY/VARRAY Form\)](#)
- [CREATE TYPE \(Distinct Form\)](#)

CREATE TYPE (Distinct Form)

Creates a user-defined type that is constructed directly from a predefined Vantage data type.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

Other SQL dialects support similar non-ANSI standard statements with names such as the following:

- CREATE DISTINCT TYPE

Required Privileges

You must have either the UDTTYPE or UDTMETHOD privilege on the SYSUDTLIB database to create a distinct UDT.

If the type definition includes a set of method signatures, then you must also have the UDTMETHOD privilege on the SYSUDTLIB database.

Privileges Granted Automatically

None.

CREATE TYPE Syntax (Distinct Form)

```
CREATE TYPE [ SYSUDTLIB. ] UDT_name AS data_type
[ CHARACTER SET server_character_set ]
FINAL [ method_specification ] [;]
```

data_type

```

{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [( fractional_seconds_precision)] [WITH TIME
ZONE] |

  INTERVAL YEAR [( precision)] [TO MONTH] |

  INTERVAL MONTH [( precision)] |

  INTERVAL DAY [( precision)]
    [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL HOUR [(precision)]
    [TO { MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL MINUTE [(precision)] [ TO SECOND
[(fractional_seconds_precision)] ] |

  INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) ] |

  REAL |

  DOUBLE PRECISION |

  FLOAT [(integer)] |

  NUMBER [( { integer | *} [, integer ]...)] |

  { DECIMAL | NUMERIC } [(integer [, integer ]...)] |

  { CHAR | BYTE | GRAPHIC } [(integer)] |

  { VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [(integer)] |

  LONG VARCHAR |

  LONG VARGRAPHIC |

  { BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }

```

```
(integer [ G | K | M ])
}
```

method_specification

```
[ INSTANCE ] METHOD [ SYSUDTLIB. ] method_name
  ( parameter_specification [,...] )
  RETURNS returns_parameter_specification [ AS LOCATOR ]
  [ CAST FROM { data_type | [ SYSUDTLIB. ] UDT_name }
  [ AS LOCATOR ] ]
  [ SPECIFIC [ SYSUDTLIB. ] specific_method_name ]
  [ SELF AS RESULT ]
  language_and_access_specification
  type_attribute [...]
```

Note:

You can specify *language_and_access_specification* and *type_attribute* in the reverse order.

parameter_specification

```
[ parameter_name ] {
  data_type [ CHARACTER SET server_character_set ] |
  [ SYSUDTLIB. ] UDT_name
} [ AS LOCATOR ]
```

returns_parameter_specification

```
data_type [ CHARACTER SET server_character_set ] |
  [ SYSUDTLIB. ] UDT_name
} [ AS LOCATOR ]
```

language_and_access_specification

```
language_clause SQL_data_access
```

Note:

If *language_and_access_specification* is before *type_attribute*, you can specify *language_clause* and *SQL_data_access* in the reverse order.

type_attribute

```
{ SPECIFIC [ SYSUDTLIB. ] specific_method_name |
  PARAMETER STYLE { SQL | TD_GENERAL } |
  [NOT] DETERMINISTIC |
  CALLED ON NULL INPUT |
  RETURNS NULL ON NULL INPUT
}
```

CREATE TYPE Syntax Elements (Distinct Form)

SYSUDTLIB

An optional specification of the containing database for *UDT_name*, which must always be SYSUDTLIB.

UDT_name

Name of the distinct UDT to be created.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, method, or Java stored procedure object even if the new object name contains only single-byte characters. Otherwise, the system returns an error to the requestor. Instead, use a multibyte session character set.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

Creating a distinct type consumes two names in the *TVMNameI* name space for SYSUDTLIB:

- The first name consumed corresponds to the name of the UDT itself.
- The second name corresponds to the system-generated UDF.

The system constructs this name as explained in the topic on naming conventions.

You cannot create a UDF within the SYSUDTLIB database with the same specific name or same routine signature.

If such a UDF already exists in SYSUDTLIB at the time you submit a CREATE TYPE request, then the request fails because a routine with the same signature already exists.

data_type

Predefined data type on which the distinct UDT is based.

You cannot specify the following data types:

- PERIOD(DATE), PERIOD(TIME), and PERIOD(TIMESTAMP)
- JSON
- XML and XMLTYPE
- ST_GEOMETRY
- DATASET

server_character_set

If *data_type* is a character type, then this is the server character set used by *data_type* , otherwise you cannot specify anything.

The purpose of the CHARACTER SET clause is to influence the system-generated routines.

The CHARACTER SET clause causes the following things to occur when the system creates the distinct type named *UDF_name*:

- The system-generated CAST routines are registered to cast to and cast from a character source type of the specified server character set.
- The system-generated ORDERING routine is registered to map to a character source type of the specified server character set.
- The system-generated tosql and fromsql TRANSFORM routines are registered to transform the UDT to and from a character source type of the specified server character set.

If you know that all database transactions occur in languages that can be expressed using the LATIN character set, specify LATIN. Otherwise, for example, if some or all transactions use non-LATIN or multibyte characters, specify UNICODE. See *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Note:

You cannot specify a character server data set of KANJI1. Otherwise, the system returns an error to the requestor.

FINAL

Mandatory keyword required for the definition of all distinct UDTs.

The FINAL keyword indicates that you cannot create subtypes of the type being defined.

method_specification

INSTANCE

The method signature being defined is an instance method.

This is the default.

SYSUDTLIB

An optional keyword that indicates that *method_name* is being created in the SYSUDTLIB database.

All methods must be created within the SYSUDTLIB database.

method_name

Name for the method whose signature being added to the type definition for *UDT_name*.

CAST FROM

The result predefined data type returned by the external function that is to be converted to the type specified by the RETURNS clause.

For example:

```
...RETURNS DECIMAL(9,5) CAST FROM FLOAT...
```

You cannot specify a character server data set of KANJI1. Otherwise, the system returns an error to the requestor.

Whenever a LOB that requires data type conversion is passed to a method, the LOB must first be materialized for the conversion to occur.

specific_method_name

Specific name of the method whose signature is being added to the type definition for *UDT_name*. Optionally, you can specify containing database name, which is SYSUDTLIB.

SELF AS RESULT

Specifies that the method is type-preserving.

If you specify SELF AS RESULT, then the data type specified in the RETURNS clause for the method must have the same name as *UDT_name*.

parameter_specification

parameter_name

A set of parameter names for the method.

The parameter list is a parenthetical comma-separated list of data types, including UDTs, and optional parameter names and locators for the variables to be passed to the method.

Parameter names must be unique within a method definition.

You cannot use the keyword SELF to name method parameters.

The maximum number of parameters a method accepts is 128.

You must specify opening and closing parentheses even if no parameters are to be passed to the function.

If you specify one parameter name, then you must specify names for all the parameters passed to the function.

If you do not specify parameter names, the system assigns unique names to them in the form P1, P2, ..., P *n*. These names are used in the COMMENT statement (see [COMMENT \(Comment Placing Form\)](#)), displayed in the report produced by the HELP METHOD statement in the PARAMETER NAME column (see [HELP METHOD](#)), and appear in the text of error messages.

data_type

Whether you specify a list of parameter names or not, you must specify a set of data types for the parameters specified.

- If a parameter in the *parameter_name* list has a predefined data type, the data type entry is that data type.
- If the type specified as data type is a predefined character data type, the *server_character_set* entry is its server character set.

The predefined data type associated with each parameter is the type of the parameter or returned value. All predefined Vantage data types except the Period types are valid. Distinct and structured UDTs are valid for scalar UDFs only. Character data can also specify a CHARACTER SET clause.

For predefined data types that take a length or size specification, like BYTE, CHARACTER, DECIMAL, VARCHAR, and so on, the size of the parameter indicates the largest number of bytes that can be passed.

If a parameter has a LOB data type, you must specify a locator indication for it.

server_character_set

You cannot specify the *server_character_set* as KANJI1. Otherwise, Vantage returns an error to the requestor.

UDT_name

If a parameter in the *parameter_name* list has a UDT type, the *UDT_name* entry is the name of that UDT.

AS LOCATOR

BLOB and CLOB types must be represented by a locator (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for a description of locators). Vantage does not support in-memory LOB parameters. An AS LOCATOR phrase must be specified for each LOB parameter and return value.

Note, however, that whenever a LOB that requires data type conversion is passed to a method, the LOB must be materialized for the conversion to take place.

returns_parameter_specification***data_type******UDT_name***

Name of the data type returned by the method, which can be either a predefined data type or a UDT.

You cannot specify a character server data set of KANJI1. Otherwise, the system returns an error to the requestor.

type_attribute**SQL**

The parameter style for the method defined by this signature is SQL.

TD_GENERAL

The parameter style for the method defined by this signature is TD_GENERAL.

DETERMINISTIC

The result of invoking the method defined by this signature is deterministic.

NOT DETERMINISTIC

The result of invoking the method defined by this signature is not deterministic.

CALLED ON NULL INPUT

The method defined by this signature is called if any of the arguments passed to is null.

RETURNS NULL ON NULL INPUT

The method defined by this signature is not called if any of the arguments passed to it is null. Instead, it returns a null.

language_clause

The language used to write the source code for the method defined by this signature.

The valid languages for writing methods are C and C++.

- If a method is written in C, then its language clause code is C.
- If a method is written in C++, then its language clause code is CPP.

This must be specified as LANGUAGE C or LANGUAGE CPP even if the external method routine is supplied in object form.

If the external method object is not written in C or C++, it must be compatible with C or C++ object code.

This clause is mandatory for each method signature you specify as part of a distinct UDT definition.

SQL_data_access

Specifies whether SQL statements are permitted within the user-written external routine for the method defined by this signature.

The only valid specification for the SQL data access clause is NO SQL.

CREATE TYPE Examples (Distinct Form)**Example: Creating a Distinct UDT for Euro Conversion**

The following example creates a distinct UDT named *euro* based on the source data type DECIMAL(8,2):

```
CREATE TYPE euro
AS DECIMAL(8, 2)
FINAL
METHOD toUS()
RETURNS us_dollar CAST FROM DECIMAL(8,2)
LANGUAGE C
```

```

DETERMINISTIC
NO SQL
RETURNS NULL ON NULL INPUT;
...

```

Example: Creating a Distinct UDT for Dollar Conversion

The following example creates a distinct UDT named *us_dollar* based on the source data type DECIMAL(8,2):

```

CREATE TYPE SYSUDTLIB.us_dollar
AS DECIMAL(8, 2)
FINAL
METHOD toEuro()
RETURNS euro CAST FROM DECIMAL(8,2)
LANGUAGE C
DETERMINISTIC
NO SQL
RETURNS NULL ON NULL INPUT;

```

Related Information

- [CREATE TYPE \(ARRAY/VARRAY Form\)](#)
- [CREATE TYPE \(Structured Form\)](#)

CREATE TYPE (ARRAY/VARRAY Form)

Creates a user-defined ARRAY or VARRAY data type that is constructed from a predefined Vantage data type, a distinct UDT data type, a structured UDT data type, or an internal UDT data type.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Unlike the ANSI SQL:2011 requirement, you must create a one-dimensional ARRAY or VARRAY type before you can use it. This variance from the ANSI SQL:2011 standard is compatible with Oracle one-dimensional VARRAY types and with IBM SQL PL one-dimensional ARRAY types.

ANSI SQL:2011 does not define a standard for multidimensional ARRAY types, nor do other major relational database management system vendors support them.

One-Dimensional ARRAY - Teradata Form

One-dimensional ARRAY types only support a lower bound of 1. Every one-dimensional ARRAY type you create has a first array element indexed by the number 1.

One-Dimensional VARRAY - Oracle-Compatible Form

One-dimensional VARRAY types only support a lower bound of 1. Every one-dimensional VARRAY type you create has a first array element indexed by the number 1.

Required Privileges

You must have either the UDTTYPE or UDTMETHOD privilege on the SYSUDTLIB database to create an ARRAY or VARRAY UDT.

Privileges Granted Automatically

None.

CREATE TYPE Syntax (ARRAY/VARRAY Form)

One-Dimensional ARRAY - Teradata Form

```
CREATE TYPE [SYSUDTLIB.] array_type_name
  AS data_type ARRAY [ number_of_elements ]
  [ DEFAULT NULL ] [;]
```

One-Dimensional VARRAY - Oracle-Compatible Form

```
CREATE TYPE [SYSUDTLIB.] array_type_name
  AS { VARYING ARRAY | VARRAY } ( number_of_elements ) OF data_type
  [ DEFAULT NULL ] [;]
```

Multidimensional Array - Teradata Form

```
CREATE TYPE [SYSUDTLIB.] array_type_name
  AS data_type ARRAY [ { lower_bound : upper_bound | maximum_size } ]
  [ DEFAULT NULL ] [;]
```

Multidimensional Array - Oracle-Compatible Form

```
CREATE TYPE [SYSUDTLIB.] array_type_name
  AS { VARYING ARRAY | VARRAY } ( { lower_bound : upper_bound | maximum_size } )
  OF data_type [ DEFAULT NULL ] [;]
```

Array Data Types

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

[ TIME | TIMESTAMP ] [(fractional_seconds_precision)] [WITH TIME ZONE] |

INTERVAL YEAR [(precision)] [TO MONTH] |

INTERVAL MONTH [(precision)] |

INTERVAL DAY [(precision)]
    [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] }] } |

INTERVAL HOUR [(precision)]
    [TO { MINUTE | SECOND [(fractional_seconds_precision)] }] |

INTERVAL MINUTE [(precision)] [TO SECOND
[(fractional_seconds_precision)]] |

INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) |

PERIOD (DATE) |

PERIOD ({ TIME | TIMESTAMP }) [(precision)] [WITH TIME ZONE] |

REAL |

DOUBLE PRECISION |

FLOAT [(integer)] |

NUMBER [( { integer | *} [, integer]... )] |

{ DECIMAL | NUMERIC } [(integer [, integer]...)] |

{ CHAR | BYTE | GRAPHIC } [(integer)] |

{ VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [(integer)] |

LONG VARCHAR |
```

```

    LONG VARGRAPHIC
}

```

CREATE TYPE Syntax Elements (ARRAY/VARRAY Form)

One-Dimensional ARRAY - Teradata Form

SYSUDTLIB

The optional name of the containing database for *array_type* name.

The only valid database name is SYSUDTLIB.

array_type_name

The name of the one-dimensional ARRAY type to be created.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, method, or Java stored procedure object even if the new object name contains only single-byte characters. Otherwise, the system returns an error to the requestor. Instead, use a multibyte session character set.

data_type

The data type on which the one-dimensional ARRAY type is based.

The element types of an array must be chosen from an existing Vantage data type categories, including Distinct UDTs, Structured UDTs, and Period data types.

You can specify any Vantage predefined data types except for:

- BLOB
- CLOB
- Distinct and structured LOB-based UDT
- Geospatial
- One-dimensional ARRAY types
- Multidimensional ARRAY types
- JSON

- XML and XMLTYPE
- DATASET

You cannot specify a character server data set of KANJI1. Otherwise, the system returns an error to the requestor.

For a description of the predefined data types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

number_of_elements

The number of elements in the one-dimensional ARRAY type being created. This must be an unsigned INTEGER value.

A one-dimensional ARRAY type can have a maximum size of 64,256 bytes.

The maximum size of the auto-generated transform string for the ARRAY is 64,256 bytes. The size of an auto-generated transform string for an ARRAY is limited to the maximum size of a DBS VARCHAR type. The overall limit on the number of elements is dependent on the size of the elements and the size of the transform. For details on how the size of the transform is calculated, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

DEFAULT NULL

Initialize all of the elements of *array_type_name* to null when the type is constructed.

One-Dimensional VARRAY - Oracle-Compatible Form

SYSUDTLIB

The optional name of the containing database for *array_type* name.

The only valid database name is SYSUDTLIB.

array_type_name

The name of the one-dimensional VARRAY type to be created.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, method, or Java stored procedure object even if the new object name contains only single-byte characters. Otherwise, the system returns an error to the requestor. Instead, use a multibyte session character set.

number_of_elements

The number of elements in the one-dimensional VARRAY type being created. This must be an unsigned INTEGER value.

A one-dimensional ARRAY type can have a maximum size of 64,256 bytes.

The auto-generated transform string for the ARRAY can have a maximum size of 64,256 bytes. The size of an auto-generated transform string for an ARRAY is limited to the maximum size of a DBS VARCHAR type. The overall limit on the number of elements is dependent on the size of the elements and the size of the transform. For details on how the size of the transform is calculated, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

data_type

The data type on which the one-dimensional VARRAY type is based.

The element types of an array must be chosen from an existing Vantage data type including distinct UDTs, structured UDTs, and Period data types. You can specify any Vantage predefined data type, except for:

- BLOB
- CLOB
- Distinct and structured LOB-based UDT
- Geospatial
- One-dimensional ARRAY types
- Multidimensional ARRAY types
- JSON
- XML and XMLTYPE
- DATASET

You cannot specify a character server data set of KANJI1. Otherwise, the system returns an error to the requestor.

For a description of the predefined data types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

DEFAULT NULL

Initialize all of the elements of *array_type_name* to null when the type is created.

Multidimensional Array - Teradata Form

You must specify a minimum of 2 dimensions for a multidimensional ARRAY type. The maximum number of dimensions you can specify is 5.

A multidimensional ARRAY type can have a maximum size of 64,256 bytes. The auto-generated transform string for the ARRAY can have a maximum size of 64,256 bytes. The size of an auto-generated transform string for the ARRAY is limited to the maximum size of a DBS VARCHAR type. The overall limit on the number of elements is dependent on the size of the elements and the size of the transform. For details on how the size of the transform is calculated, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

SYSUDTLIB

Optionally, the name of the containing database for *array_type* name.

The database name is SYSUDTLIB.

array_type_name

The name of the multidimensional ARRAY type to be created.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, method, or Java stored procedure object even if the new object name contains only single-byte characters. Otherwise, the system returns an error to the requestor. Instead, use a multibyte session character set.

data_type

The data type on which the multidimensional ARRAY type is based.

The element types of an array must be chosen from an existing Vantage data type including Distinct UDTs, Structured UDTs, and Period data types.

You can specify any supported Vantage predefined data types, except for:

- BLOB
- CLOB
- Distinct and structured LOB-based UDT
- Geospatial
- One-dimensional ARRAY types
- Multidimensional ARRAY types
- JSON
- XML and XMLTYPE
- DATASET

You cannot specify a character server data set of KANJI1. Otherwise, the system returns an error to the requestor.

For a description of the predefined data types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

lower_bound

A signed INTEGER number that defines the lower bound for a multidimensional ARRAY type dimension. The lower boundary can be a negative number and must be less than or equal to *upper_bound*.

upper_bound

A signed INTEGER number that defines the upper bound for a multidimensional ARRAY type dimension. The upper boundary can be a negative number and must be greater than or equal to *lower_bound*.

maximum_size

The maximum size of a dimension of a type created as a multidimensional ARRAY. The current dimension of the multidimensional ARRAY can have a maximum size of 64,256 bytes.

When you specify this option, the lower bound for the current dimension is implicitly defined to be 1.

This must be an unsigned INTEGER value.

DEFAULT NULL

Initialize all of the elements of *array_type_name* to null when the type is created.

Multidimensional Array - Oracle-Compatible Form

You must specify a minimum of 2 dimensions for a multidimensional VARRAY type. The maximum number of dimensions you can specify is 5.

A multidimensional VARRAY can have a maximum size of 64,256 bytes. The auto-generated transform string for the VARRAY can have a maximum size of 64,256 bytes. The size of an auto-generated transform string for the VARRAY is limited to the maximum size of a DBS VARCHAR type. The overall limit on the number of elements is dependent on the size of the elements and the size of the transform. For details on how the size of the transform is calculated, see *SQL Data Types and Literals*.

SYSUDTLIB

The optional name of the containing database for *array_type* name.

The only valid database name is SYSUDTLIB.

array_type_name

The name of the multidimensional VARRAY type to be created.

Note:

If the UDF library for your database contains any objects with multibyte characters in their name, you cannot use a single-byte session character set to create a new UDF, UDT, method, or Java stored procedure object even if the new object name contains only single-byte characters. Otherwise, the system returns an error to the requestor. Instead, use a multibyte session character set.

lower_bound

Signed INTEGER number that defines the lower bound for a multidimensional VARRAY type dimension. The lower boundary can be a negative number and must be less than or equal to *upper_bound*.

upper_bound

Signed INTEGER number that defines the upper bound for a multidimensional VARRAY type dimension. The upper boundary can be a negative number and must be greater than or equal to *lower_bound*.

maximum_size

Maximum size of a type created as a multidimensional VARRAY. The current dimension of the multidimensional VARRAY can have a maximum size 64,256 bytes.

When you specify this option, the lower bound for the current dimension is implicitly defined to be 1.

This must be an unsigned INTEGER value.

data_type

The data type on which the multidimensional ARRAY type is based.

The element types of an array must be chosen from an existing Vantage data type including Distinct UDTs, Structured UDTs, and Period data types.

You can specify any Vantage predefined data types, except for:

- BLOB
- CLOB
- Distinct and structured LOB-based UDT
- Geospatial
- One-dimensional ARRAY types
- Multidimensional ARRAY types

- JSON
- XML and XMLTYPE
- DATASET

You cannot specify a character server data set of KANJI1. Otherwise, the system returns an error to the requestor.

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for documentation of the predefined data types supported by Vantage.

DEFAULT NULL

Initialize all of the elements of *array_type_name* to null when the type is created.

Examples

Example: One-Dimensional ARRAY Type

These examples all create a one-dimensional ARRAY type named *phonenumbers_ary*.

The first example uses Teradata-style syntax and initializes all of its elements to nulls.

```
CREATE TYPE phonenumbers_ary AS
CHARACTER(10) ARRAY[5]
DEFAULT NULL;
```

The next example uses Oracle-style syntax to create an identical VARRAY type to the type created in the first example.

```
CREATE TYPE phonenumbers_ary AS
VARRAY(5) OF CHARACTER(10)
DEFAULT NULL;
```

The final example uses Oracle-style syntax to create an identical VARRAY type, but does not initialize the elements to nulls.

```
CREATE TYPE phonenumbers_ary AS
VARRAY(5) OF CHARACTER(10);
```

Example: UDT and a One-Dimensional ARRAY Type

This example first creates a structured UDT named *employee* and then creates ARRAY types based on it using both Teradata-style and Oracle-style syntax.

```
CREATE TYPE employee AS (
  first_name  CHARACTER(10),
  last_name   CHARACTER(10),
  employee_id INTEGER)
...;
```

The first example creates the ARRAY type *structudt_ary*, which is based on *employee*, using Teradata-style syntax.

```
CREATE TYPE structudt_ary AS
employee ARRAY[20];
```

The next example creates the same ARRAY type as a VARRAY type using Oracle-style syntax.

```
CREATE TYPE structudt_ary AS
VARRAY(20) OF employee;
```

Example: Three-Dimensional ARRAY Type

These examples all create a 3-dimensional ARRAY type named *seismic_cube* based on the UDT *measures_udt*, which has the following SQL create text.

```
CREATE TYPE measures_udt AS (
  amplitude INTEGER,
  phase      INTEGER,
  frequency  INTEGER);
```

The first example uses Teradata-style syntax and initializes all of its elements to nulls.

```
CREATE TYPE seismic_cube AS
  measures_udt ARRAY [1:5][1:7][1:20]
DEFAULT NULL;
```

The next example also uses Teradata-style syntax, but does not initialize its elements to nulls.

```
CREATE TYPE seismic_cube AS
measures_udt ARRAY [1:5][1:7][1:20];
```

Example: Three-Dimensional ARRAY Type Using Different Syntax

These examples create the same 3-dimensional ARRAY type as [Example: Three-Dimensional ARRAY Type](#), but use different syntax variations to define them.

The first example uses Teradata-style syntax to define a variant of the *seismic_cube* type, *seismic_cube_2*, based on the type *measures_udt*, which is defined in [Example: Three-Dimensional ARRAY Type](#).

```
CREATE TYPE seismic_cube_2 AS
measures_udt ARRAY [5][7][20];
```

The next example uses another variant of Teradata-style syntax to define an equivalent *seismic_cube* type named *seismic_cube_3*.

```
CREATE TYPE seismic_cube_3 AS
measures_udt ARRAY [1:5][7][1:20];
```

Example: Three-Dimensional VARRAY Type

These examples all create an Oracle-style 3-dimensional VARRAY type named *seismic_cube* based on the UDT *measures_udt*, which has the following SQL create text.

This example initializes all of its elements to nulls.

```
CREATE TYPE seismic_cube AS
VARRAY (1:5)(1:7)(1:20) OF measures_udt
DEFAULT NULL;
```

This example does not initialize its elements to nulls.

```
CREATE TYPE seismic_cube AS
VARRAY (1:5)(1:7)(1:20) OF measures_udt;
```

The next example defines an equivalent *seismic_cube* type, *seismic_cube_2*.

```
CREATE TYPE seismic_cube_2 AS
VARRAY (5)(7)(20) measures_udt;
```

The final example defines an equivalent *seismic_cube* type, *seismic_cube_3*.

```
CREATE TYPE seismic_cube_3 AS
VARRAY (1:5)(7)(1:20) measures_udt;
```

Example: Three-Dimensional ARRAY and VARRAY Types With a Negative Lower Bound on a Dimension

The following example creates a 3-dimensional ARRAY type that has a negative lower bound value for its third dimension.

The following example uses Teradata-style syntax.

```
CREATE TYPE seismic_cube_4 AS
measures_udt ARRAY [5][7][-10:10];
```

The following example uses Oracle-style syntax to define an equivalent *seismic_cube_4* type, *seismic_cube_5*.

```
CREATE TYPE seismic_cube_5 AS
VARRAY (5)(7)(-10:10) OF measures_udt;
```

Related Information

- [CREATE TYPE \(Distinct Form\)](#)
- [CREATE TYPE \(Structured Form\)](#)
- *Teradata Vantage™ - Data Types and Literals*, B035-1143

CREATE *storage_format* SCHEMA

Creates a schema for a specific storage format of the DATASET type.

For information on the DATASET data type, see *Teradata Vantage™ - DATASET Data Type*, B035-1198.

Required Privileges

You must have the CREATE DATASET SCHEMA privilege on database SYSUDTLIB.

CREATE *storage_format* SCHEMA Syntax

```
CREATE storage_format SCHEMA [SYSUDTLIB.] schema_name AS schema [;]
```


CREATE *storage_format* SCHEMA Syntax Elements

storage_format

Storage format of the DATASET type to use in creating the schema.

SYSUDTLIB

Name of the database containing the schema. Schemas that you create are registered in the SYSUDTLIB database.

schema_name

Name of the schema. Schema names must be unique across the entire system.

schema

You must specify the schema as a constant, that is, literal value. The literal can contain the schema in its text format or in a byte format which contains the UTF-8 encoded characters that compose the schema. Schemas are stored as a Teradata VARBYTE which contains the UTF-8 encoded characters that compose the schema, to a maximum of 63,000 bytes.

Examples

Example: Schema for DATASET Type with Avro Storage Format

This example creates a schema for the Avro storage format of the DATASET type:

```
CREATE Avro SCHEMA chemDatasetSchema AS
'{
  "namespace": "example.avro",
  "type": "record",
  "name": "User",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "favorite_number", "type": ["int", "null"]},
    {"name": "favorite_color", "type": ["string", "null"]}
  ]
}';
```

This example creates the same schema using a byte literal which contains the UTF-8 encoded characters of the schema:

```
CREATE Avro SCHEMA chemDatasetSchema AS
'7B226E616D657370616365223A226578616D706C652E6176726F222C2274797065223A227265636
```

```
F7264222C226E616D65223A2255736572222C226669656C6473223A5B7B226E616D65223A226E616
D65222C2274797065223A22737472696E67227D2C7B226E616D65223A226661766F726974655F6E7
56D626572222C2274797065223A5B22696E74222C226E756C6C225D7D2C7B226E616D65223A22666
1766F726974655F636F6C6F72222C2274797065223A5B22737472696E67222C226E756C6C225D7D5
D7D'xb;
```

Example: Schema for DATASET Type with CSV Storage Format

For this example, refer to the sample of the rivers.tsv file:

```
09379180    LAGUNA CREEK AT DENNEHOTSO, AZ
09379910    COLORADO RIVER BELOW GLEN CANYON DAM, AZ
09380000    COLORADO RIVER AT LEES FERRY, AZ
09382000    PARIA RIVER AT LEES FERRY, AZ
09383300    FILLER DITCH AT GREER, AZ
09383400    LITTLE COLORADO RIVER AT GREER, AZ
09383100    COLORADO R ABV LITTLE COLORADO R NR DESERT VIEW
09379200    CHINLE CREEK NEAR MEXICAN WATER, AZ
09379050    LUKACHUKAI CREEK NEAR LUKACHUKAI, AZ
09379025    CHINLE CREEK AT CHINLE, AZ
```

This statement creates a schema for the rivers.tsv file.

```
CREATE CSV SCHEMA Rivers_Schema AS
'{"field_delimiter":"\t",
"field_names":["site_no","location"]}';
```

Usage Notes

Schemas are Converted to UTF-8

The schemas you specify are converted to UTF-8 before insertion into the DBC.DatasetSchemaInfo table.

CREATE CAST and REPLACE CAST

Creates or replaces a cast operation for a UDT.

ANSI Compliance

CREATE CAST is ANSI SQL:2011-compliant with extensions.

REPLACE CAST is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have either the UDTTYPE or UDTMETHOD privilege on the *SYSUDTLIB* database to create or replace a cast.

If you specify a *cast_function*, then you must also have the EXECUTE FUNCTION privilege on the UDF, and it must be contained within the *SYSUDTLIB* database.

Privileges Granted Automatically

None.

CREATE CAST and REPLACE CAST Syntax

```
{ CREATE | REPLACE } CAST ( source AS target )
  WITH { specific_method | specific_function }
  [ AS ASSIGNMENT ] [;]
```

source

```
{ source_predefined_data_type | [SYSUDTLIB.] source_UDT_name }
```

target

```
{ target_predefined_data_type | [SYSUDTLIB.] target_UDT_name }
```

specific_method

```
{ SPECIFIC METHOD specific_method_name |
  [ INSTANCE ] METHOD method_name ( [ data_name | UDT_name
  ][,...] )
  } FOR UDT_name
```

specific_function

```
{ SPECIFIC FUNCTION specific_function_name |
  FUNCTION function_name ( [ data_name | UDT_name ][,...] )
  }
```

CREATE CAST and REPLACE CAST Syntax Elements

source_predefined_data_type

Either a structured or distinct UDT or any valid predefined data type. You cannot specify an ARRAY or VARRAY type. For a list of data types, see [Data Types Syntax](#).

Either *source_data_type* or *target_data_type* or both must be a UDT. This can be either a user-created UDT or an internal UDT.

If *source_data_type* is a predefined data type, then its cast routine must be a cast function.

See [specific_function](#).

target_predefined_data_type

Either a structured or distinct UDT or any valid predefined data type. You cannot specify an ARRAY or VARRAY type.

Either *target_data_type* or *source_data_type* or both must be a UDT.

specific_method

If you specify a specific method, then a method with the same specific method name must be defined in the SYSUDTLIB database.

METHOD

The name of the method to be used to convert a source UDT value to the target data type.

You cannot invoke an SQL UDF from a CREATE CAST or REPLACE CAST request.

method_name

A method associated with the specified *UDT_name* must exist with the same signature.

The method must satisfy the following conditions.

- Associated UDT is the same as the source data type.
- Declared parameter list is empty.
- Result data type is the same as the target data type.
- Must be deterministic.

If no method is found, an error is returned.

INSTANCE

The object is an instance method.

FOR *UDT_name*

The name of the UDT associated with *method_name*.

specific_function***function_name***

The specific name of the external UDF to be used to convert *source_data_type* to *target_data_type*. The *specific_function_name* must specify a unique function.

You cannot invoke an SQL UDF from a CREATE CAST or REPLACE CAST request.

If you do not specify a set of data types for a function name, then the specified function must be the only external UDF with that name in the database.

The specified function must satisfy the following conditions:

- It can have only one parameter.
Its parameter data type must be the same as *source_data_type*.
- Its result data type must be the same as *target_data_type*.
- It must be defined to be deterministic.
- Its function name must identify an external UDF in the SYSUDTLIB database.

FUNCTION***function_name***

The name of the external UDF to be used to convert *source_data_type* to *target_data_type*. The *function_name* must specify a unique function.

You cannot invoke an SQL UDF from a CREATE CAST or REPLACE CAST request.

If you do not specify a set of data types for a function name, the function must be the only external UDF with that name in the database.

The function can have only one parameter. The result data type must be the same as *target_data_type*. The function must be defined to be deterministic. The function name must identify an external UDF in the SYSUDTLIB database.

data type

The parameter data type must be the same as *source_data_type*.

UDT_name

The parameter data type must be the same as *source_data_type*.

AS ASSIGNMENT

To implicitly invoke the specified cast routine on an assignment operation if both of the following statements are true:

- The source data type of the assignment is compatible with the source data type of the cast result.
- The target data types are identical.

Note that an assignment operation is defined to be any of the following operations:

- INSERT
- UPDATE
- parameter passing

You can disable this functionality by means of the `DisableImplCastForSysFuncOp` DBS Control parameter.

Note:

Be very careful if you are performing a REPLACE CAST statement because the system updates the UDTCast dictionary to reflect the value of the AS ASSIGNMENT flag.

CREATE CAST and REPLACE CAST Examples

Example: Casting Source Data as VARCHAR

The following CREATE CAST request creates a cast operation that converts the source data type `circle` to the target data type `VARCHAR(32)` using the method having the specific method name *circleToVarchar*.

```
CREATE CAST (circle AS VARCHAR(32))
WITH SPECIFIC METHOD SYSUDTLIB.circleToVarchar
```

Example: Casting Source Data using a UDF

The following CREATE CAST request creates a cast operation that converts the source data type `VARCHAR(32)` to the target data type `circle` using the UDF having the specific function name *VarcharToCircle*.

```
CREATE CAST (VARCHAR(32) AS circle)
WITH SPECIFIC FUNCTION SYSUDTLIB.VarcharToCircle
```

Example: Casting Source Data as VARCHAR with an Assignment Operation

The following CREATE CAST request creates a cast operation that converts the source data type *euro* to the target data type `VARCHAR(20)` using the method *euroString*.

The casting routine is defined so it is implicitly invoked on an assignment operation assuming the necessary conditions are met for the assignment.

```
CREATE CAST (euro AS VARCHAR(20))
WITH METHOD SYSUDTLIB.euroString( ) FOR euro
AS ASSIGNMENT
```

Related Information

- *CREATE CAST in Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184

For information about using the SQL CAST function with predefined data types, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

For information about coding methods for external routines to support methods and UDFs, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

See the following Teradata Tools and Utilities books for details about how the software they document deals with UDTs:

- *Teradata® FastExport Reference*, B035-2410
- *Teradata® FastLoad Reference*, B035-2411
- *Teradata JDBC Driver Reference*, available at <https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html>
- *Teradata® MultiLoad Reference*, B035-2409
- *ODBC Driver for Teradata® User Guide*, B035-2526
- *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446
- *Teradata® Parallel Data Pump Reference*, B035-3021
- *Teradata® Parallel Transporter Reference*, B035-2436

CREATE ORDERING and REPLACE ORDERING

Creates or replaces a map ordering routine used to compare UDT values.

ANSI Compliance

CREATE ORDERING is ANSI SQL:2011-compliant.

REPLACE ORDERING is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have either the UDTTYPE or UDTMETHOD privilege on the SYSUDTLIB database.

If the map routine you specify is a UDF, then you must also have the EXECUTE FUNCTION privilege on the UDF, which must be contained within the SYSUDTLIB database.

Privileges Granted Automatically

None.

CREATE ORDERING and REPLACE ORDERING Syntax

```
{ CREATE | REPLACE } ORDERING FOR [ SYSUDTLIB. ] UDT_name
  ORDER FULL BY MAP WITH { method_specification | function_specification } [;]
```

method_specification

```
{ SPECIFIC METHOD [ SYSUDTLIB. ] specific_method_name |
  [ INSTANCE ] METHOD [ SYSUDTLIB. ] method_name
  ( [ { data_type | [ SYSUDTLIB. ] UDT_name } [,...] ] )
} FOR UDT_name
```

function_specification

```
{ SPECIFIC FUNCTION [ SYSUDTLIB. ] specific_function_name |
  FUNCTION [ SYSUDTLIB. ] function_name
  ( [ { data_type | [ SYSUDTLIB. ] UDT_name } [,...] ] )
}
```

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [( fractional_seconds_precision)] [WITH TIME
ZONE] |

  INTERVAL YEAR [( precision)] [TO MONTH] |

  INTERVAL MONTH [( precision)] |

  INTERVAL DAY [( precision)]
  [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL HOUR [(precision)]
  [TO { MINUTE | SECOND [(fractional_seconds_precision)] } ] |

  INTERVAL MINUTE [(precision)] [ TO SECOND
[(fractional_seconds_precision)] ] |
```



```

INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) |
PERIOD (DATE) |
PERIOD ({ TIME | TIMESTAMP } [(precision)] [ WITH TIME ZONE ]) |
REAL |
DOUBLE PRECISION |
FLOAT [(integer)] |
NUMBER [( { integer | *} [, integer ]...)] |
{ DECIMAL | NUMERIC } [(integer [, integer ]...)] |
{ CHAR | BYTE | GRAPHIC } [(integer)] |
{ VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [(integer)] |
LONG VARCHAR |
LONG VARGRAPHIC |
{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
  (integer [ G | K | M ]) |
{ XML | XMLTYPE } |
JSON [ ( integer ) ] [ CHARACTER SET { UNICODE | LATIN } ] |
[SYSUDTLIB.] { UDT_name | ST_Geometry | MBR | ARRAY_name
| VARRAY_name }
}

```

CREATE ORDERING and REPLACE ORDERING Syntax Elements

*UDT_name***FOR SYSUDTLIB.UDT_name**

The name of the UDT with which this ordering map is associated. This can be either a user-created UDT or an internal UDT.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

*method_specification***specific_method_name**

The specific name for the method that maps the UDT to a predefined data type value so it can be compared.

The specified method must already be defined. If you specify the name of a method that has not yet been created, the system returns an error to the requestor.

method_name

The name of the method.

The specified method must already be defined. If you specify the name of a method that has not yet been created, the system returns an error to the requestor.

data_type

Data type list that maps the UDT to a predefined data type value so it can be compared.

UDT_name

The name of the UDT with which the specified instance method is associated.

*function_specification***specific_function_name**

The specific name for the external UDF that maps the UDT to a predefined data type value so it can be compared.

You cannot invoke an SQL UDF from a CREATE ORDERING or REPLACE ORDERING statement.

function_name

The name of the function.

data_type

Data type list that maps the UDT to a predefined data type value so it can be compared.

CREATE ORDERING and REPLACE ORDERING Examples

Example: Order Mapping for a UDT using a UDF

The following order mapping definition for a UDT named *rectangle* uses the UDF named *compare_rect()*.

```
CREATE ORDERING FOR rectangle
ORDER FULL BY MAP WITH FUNCTION compare_rect();
```

Example: Order Mapping for a UDT using a Method

The following order mapping definition for a UDT named *circle* uses the method named *compare_circ()*, which is also associated with the *circle* UDT.

```
CREATE ORDERING FOR circle
ORDER FULL BY MAP WITH METHOD compare_circ() FOR circle;
```

Example: Order Mapping for a UDT using a Method

The following order mapping definition for a UDT named *circle* uses the method named *compare_circ()*, which is also associated with the *circle* UDT.

```
CREATE ORDERING FOR circle
ORDER FULL BY MAP WITH METHOD compare_circ() FOR circle;
```

Related Information

- [ALTER FUNCTION](#)
- [ALTER METHOD](#)
- [CREATE FUNCTION \(External Form\) and and REPLACE FUNCTION \(External Form\)](#)
- [CREATE METHOD](#)
- [CREATE TRANSFORM and REPLACE TRANSFORM](#)

- [CREATE TYPE \(Distinct Form\)](#)
- [CREATE TYPE \(Structured Form\)](#)
- [DROP FUNCTION](#)
- [DROP TRANSFORM](#)
- [RENAME FUNCTION \(External Form\)](#)
- [REPLACE METHOD](#)
- [HELP FUNCTION](#)
- [HELP METHOD](#)
- [SHOW object](#)

See *Teradata Vantage™ - SQL Data Control Language*, B035-1149 for information about the UDTTYPE, UDTMETHOD, UDTUSAGE, and EXECUTE FUNCTION privileges, particularly the UDT-related and EXECUTE FUNCTION privileges for GRANT (SQL Form).

Also see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about how to write user-defined external routines.

CREATE TRANSFORM and REPLACE TRANSFORM

Creates or replaces transform groups for importing UDT data from a client system to a Vantage and exporting UDT data from a Vantage to a client system. You can use CREATE TRANSFORM to add transform groups in addition to the existing transform groups. REPLACE TRANSFORM drops all of the existing transform groups and adds the transform groups you specify.

ANSI Compliance

CREATE TRANSFORM is ANSI SQL:2011-compliant.

REPLACE TRANSFORM is a Teradata extension to the ANSI SQL-2011 standard.

Required Privileges

You must have at least one of the following privileges on the SYSUDTLIB database to perform CREATE TRANSFORM or REPLACE TRANSFORM:

- UDTMETHOD
- UDTTYPE

If any of the external routines you specify is a UDF, then you must also have the EXECUTE FUNCTION privilege on the UDF, and the UDF must be contained within the SYSUDTLIB database.

CREATE TRANSFORM and REPLACE TRANSFORM Syntax

```
{ CREATE | REPLACE } TRANSFORM FOR [SYSUDTLIB.] UDT_name
transform_specification [...] [;]
```

transform_specification

```
transform_group ( TO SQL WITH to_method_or_function FROM SQL WITH
from_method_or_function )
```

to_method_or_function***from_method_or_function***

```
{ specific_method | instance_method | specific_function | function }
```

specific_method

```
SPECIFIC [SYSUDTLIB.] specific_method_name FOR [SYSUDTLIB.] UDT_name
```

instance_method

```
[INSTANCE] METHOD [SYSUDTLIB.] method_name
( [ data_type | [SYSUDTLIB.] UDT_name ] [,...] ) FOR
[SYSUDTLIB.] UDT_name
```

specific_function

```
SPECIFIC FUNCTION [SYSUDTLIB.] specific_function_name
```

function

```
FUNCTION [SYSUDTLIB.] function_name
( [ data_type | [SYSUDTLIB.] UDT_name ] [,...] )
```

CREATE TRANSFORM and REPLACE TRANSFORM Syntax Elements

SYSUDTLIB

Optional specification of containing database.

UDT_name

Name of the UDT associated with the transform group. This can be a structured or distinct UDT only. You cannot create transforms for complex data types such as ARRAY or XML.

See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

transform_specification

transform_group

A mandatory name for the transform group.

You can create or replace up to 16 transform groups per UDT. The first transform group you specify is the default transform group. A UDT cannot have two transform groups with the same name.

The name of a transform group cannot begin with the characters TD_, which are reserved as a transform group name prefix for internal use. For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

UDT transform names need not be unique across UDTs.

Note:

Transform group names are stored in DBC.UDTTransform, not in DBC.TVM.

TO SQL

The tosql routine maps a predefined data type value to a UDT value. The system invokes it automatically whenever a UDT value is transferred to Vantage from a client application.

Note that the external routine invoked by a tosql clause must be an external UDF. It cannot be a method.

FROM SQL

The fromsql routine maps a UDT value to a predefined data type value. The system invokes it automatically whenever a UDT value is transferred from Vantage to a client application.

The external routine invoked by a fromsql clause can be either an external UDF or a method.

to_method_or_function

specific_method

instance_method

SYSUDTLIB

specific_function

function

from_method_or_function

specific_method

instance_method

specific_function

function

TO SQL WITH SPECIFIC METHOD *specific_method_name*

Name of the specific method to be used as the tosql routine for this transform group.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

SYSUDTLIB

Optional specification of containing database.

FOR UDT_ *name*

The user-defined data type for *specific_method_name*.

TO SQL WITH METHOD *method_name*

TO SQL WITH INSTANCE METHOD *method_name*

Name of the method to be used as the tosql routine for this transform group.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

SYSUDTLIB

Optional specification of containing database.

(data_type)

The list of data types that uniquely identifies an overloaded *method_name* for this transform group. For a list of data types, see [Data Types Syntax](#).

UDT_name

The user-defined data type list for *method_name*.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

TO SQL WITH SPECIFIC FUNCTION *specific_function_name*

Name of the specific external UDF to be used as the tosql routine for this transform group.

You cannot invoke an SQL UDF from a CREATE TRANSFORM or REPLACE TRANSFORM request.

SYSUDTLIB

Optional specification of containing database.

TO SQL WITH FUNCTION *function_name*

Name of the external UDF to be used as the tosql routine for this transform group.

You cannot invoke an SQL UDF from a CREATE TRANSFORM or REPLACE TRANSFORM request.

SYSUDTLIB

Optional specification of containing database.

(data_type)

List of data types that uniquely identifies an overloaded *function_name* for this transform group. For a list of data types, see [Data Types Syntax](#).

UDT_name

User-defined data type list for *function_name*.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

FROM SQL WITH SPECIFIC *specific_method_name*

Name of the specific method to be used as the fromsql routine for this transform group.

SYSUDTLIB

Optional specification of containing database.

FROM SQL WITH METHOD *method_name***FROM SQL WITH INSTANCE METHOD *method_name***

Name of the method to be used as the fromsql routine for this transform group.

SYSUDTLIB

Optional specification of containing database.

(*data_type*)

List of data types that uniquely identifies an overloaded *method_name* for this transform group. For a list of data types, see [Data Types Syntax](#).

UDT_name

User-defined data type list for *method_name*.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

FROM SQL WITH SPECIFIC FUNCTION *specific_function_name*

Name of the specific external UDF to be used as the fromsql routine for this transform group.

You cannot invoke an SQL UDF from a CREATE TRANSFORM or REPLACE TRANSFORM request.

SYSUDTLIB

Optional specification of containing database.

FROM SQL WITH FUNCTION *function_name*

Name of the external UDF to be used as the fromsql routine for this transform group.

You cannot invoke an SQL UDF from a CREATE TRANSFORM or REPLACE TRANSFORM request.

SYSUDTLIB

Optional specification of containing database.

(data_type)

List of data types that uniquely identifies an overloaded *function_name* for this transform group. For a list of data types, see [Data Types Syntax](#).

UDT_name

User-defined data type list for *function_name*.

SYSUDTLIB

Optional specification of containing database.

CREATE TRANSFORM and REPLACE TRANSFORM Examples

Example: Creating a Transform Request

The following CREATE TRANSFORM request creates the transform group named *client_IO*, which is composed of the tosql transform *stringToAddress* and the fromsql transform *toString* for the UDT named *address*.

```
CREATE TRANSFORM FOR address client_IO (
  TO SQL WITH SPECIFIC FUNCTION SYSUDTLIB.stringToAddress,
  FROM SQL WITH SPECIFIC METHOD toString);
```

Example: Creating Multiple Transform Groups

Create the functions for the FROM SQL WITH and TO SQL WITH clauses. This example uses functions for integer, character, and binary values.

```
CREATE FUNCTION SYSUDTLIB.xml_struct2int_fromsql(xml_struct2)
RETURNS INTEGER
NO SQL
CALLED ON NULL INPUT
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!xml_struct2int_fromsql!xml_struct2int_fromsql.c!
F!xml_struct2int_fromsql';
```

```
CREATE FUNCTION SYSUDTLIB.xml_struct2int_tosql(INTEGER)
RETURNS xml_struct2
NO SQL
```

```

PARAMETER STYLE TD_GENERAL
RETURNS NULL ON NULL INPUT
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!xmld_struct2int_tosql!xmld_struct2int_tosql.c!
F!xmld_struct2int_tosql';

```

```

CREATE FUNCTION SYSUDTLIB.xmld_struct2char_fromsql(xmld_struct2)
RETURNS CHAR
NO SQL
CALLED ON NULL INPUT
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!xmld_struct2char_fromsql!xmld_struct2char_fromsql.c!
F!xmld_struct2char_fromsql';

```

```

CREATE FUNCTION SYSUDTLIB.xmld_struct2char_tosql(CHAR)
RETURNS xmld_struct2
NO SQL
PARAMETER STYLE TD_GENERAL
RETURNS NULL ON NULL INPUT
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!xmld_struct2char_tosql!xmld_struct2char_tosql.c!
F!xmld_struct2char_tosql';

```

```

CREATE FUNCTION SYSUDTLIB.xmld_struct2byte_fromsql(xmld_struct2)
RETURNS BYTE
NO SQL
CALLED ON NULL INPUT
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!xmld_struct2byte_fromsql!xmld_struct2byte_fromsql.c!
F!xmld_struct2byte_fromsql';

```

```

CREATE FUNCTION SYSUDTLIB.xmld_struct2byte_tosql(BYTE)
RETURNS xmld_struct2
NO SQL
PARAMETER STYLE TD_GENERAL
RETURNS NULL ON NULL INPUT
DETERMINISTIC

```

```
LANGUAGE C
EXTERNAL NAME 'CS!xml_struct2byte_tosql!xml_struct2byte_tosql.c!
F!xml_struct2byte_tosql';
```

This statement creates three transform groups:

```
CREATE TRANSFORM FOR XMLD_STRUCT2
XMLD_STRUCT2INT (TO SQL WITH SPECIFIC FUNCTION SYSUDTLIB.XMLD_STRUCT2INT_TOSQL,
                 FROM SQL WITH SPECIFIC
                 FUNCTION SYSUDTLIB.XMLD_STRUCT2INT_FROMSQL)
XMLD_STRUCT2CHAR(TO SQL WITH SPECIFIC FUNCTION SYSUDTLIB.XMLD_STRUCT2CHAR_TOSQL,
                 FROM SQL WITH SPECIFIC
                 FUNCTION SYSUDTLIB.XMLD_STRUCT2CHAR_FROMSQL)
XMLD_STRUCT2BYTE(TO SQL WITH SPECIFIC FUNCTION SYSUDTLIB.XMLD_STRUCT2BYTE_TOSQL,
                 FROM SQL WITH SPECIFIC
                 FUNCTION SYSUDTLIB.XMLD_STRUCT2BYTE_FROMSQL);
```

Related Information

- [ALTER FUNCTION](#)
- [CREATE METHOD](#)
- [REPLACE METHOD](#)
- [HELP METHOD](#)

See the documentation for the following statements, as appropriate, for additional information about user-defined functions.

- [ALTER FUNCTION](#)
- [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#)
- [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#)
- [CREATE FUNCTION and REPLACE FUNCTION \(Table Form\)](#)
- [HELP FUNCTION](#)

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for information about the standard SQL CAST function and its uses with predefined data types.

See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about coding methods for external routines to support methods and UDFs.

See the following Teradata Tools and Utilities books for details about how the software they document deals with UDTs:

- *Basic Teradata® Query Reference*, B035-2414
- *Teradata® FastExport Reference*, B035-2410
- *Teradata® FastLoad Reference*, B035-2411

- *Teradata JDBC Driver Reference*, available at <https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html>
- *Teradata® MultiLoad Reference*, B035-2409
- *ODBC Driver for Teradata® User Guide*, B035-2526
- *Teradata® Preprocessor2 for Embedded SQL Programmer Guide*, B035-2446
- *Teradata® Parallel Data Pump Reference*, B035-3021
- *Teradata® Parallel Transporter Reference*, B035-2436
- *Teradata® Tools and Utilities Access Module Reference*, B035-2425

SET TRANSFORM GROUP FOR TYPE

Sets the active transform group for Complex Data Types (CDTs) that have multiple transform groups.

You can set transform groups for the following CDTs:

- JSON
- XML
- ST_GEOMETRY
- DATASET

The transform group you set is in effect for the current session. You can use this statement multiple times in a single session to switch transforms for the same CDT. If default transform groups are already set for the user, the SET TRANSFORM GROUP FOR TYPE statement overrides the default settings for the current session.

For information about CDTs, including macros you can use to display the current transform settings, see *Teradata Vantage™ - JSON Data Type*, B035-1150, *Teradata Vantage™ - XML Data Type*, B035-1140, *Teradata Vantage™ - DATASET Data Type*, B035-1198, and *Teradata Vantage™ - Geospatial Data Types*, B035-1181.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Privileges Required

None.

SET TRANSFORM GROUP FOR TYPE Syntax

```
SET TRANSFORM GROUP FOR TYPE UDT_name transform_group [;]
```

SET TRANSFORM GROUP FOR TYPE Syntax Elements

UDT_name

Name of user-defined data type (UDT) for which to set transform.

transform_group

Name of transform group to assign to the user-defined type (UDT). For information on creating transform groups, see [CREATE TRANSFORM](#) and [REPLACE TRANSFORM](#).

Examples

Examples: Setting Transform Groups for UDTs

This statement sets the JSON data type with the Latin character set to the TD_JSON_VARCHAR transform.

```
SET TRANSFORM GROUP FOR TYPE JSON CHARACTER SET LATIN TD_JSON_VARCHAR;
```

This statement sets the JSON data type with the BSON storage format to the TD_JSON_VARCHAR transform.

```
SET TRANSFORM GROUP FOR TYPE JSON STORAGE FORMAT BSON TD_JSON_VARCHAR;
```

This statement sets the ST_GEOMETRY data type to the TD_GEO_VARCHAR transform.

```
SET TRANSFORM GROUP FOR TYPE ST_GEOMETRY TD_GEO_VARCHAR;
```

The following statement uses the TD_JSON_VARCHAR transform for column c1, which is BSON data type, and uses the TD_GEO_VARCHAR transform for column c2.

```
SELECT NEW JSON('{}', BSON) AS c1, NEW ST_GEOMETRY('POINT(1 1)') AS c2;
```

Usage Notes

JSON and XML in MultipartRecord Mode

JSON and XML in MultipartRecord Mode have a special transfer protocol that does not use transforms. External JSON and external XML data transferred from clients to Vantage are also processed without transforms. In this mode, the SET TRANSFORM GROUP FOR TYPE statement does not apply. However,

JSON and XML can use transforms in Field, Record, and Indicator modes. You can use the SET TRANSFORM GROUP FOR TYPE statement to set the session transform for JSON and XML data in Field, Record, and Indicator modes.

ALTER TYPE

Performs any of the following operations for a UDT.

- Add a new attribute to a structured UDT definition.
- Drop an existing attribute from a structured UDT definition.
- Add a method to a distinct or structured UDT definition.
- Drop a method from a distinct or structured UDT definition.
- Recompile the source code for a distinct, structured, ARRAY, or VARRAY type definition.

ANSI Compliance

ALTER TYPE is ANSI SQL-2011-compliant with Teradata extensions.

Other SQL dialects support similar non-ANSI standard statements with names such as DROP METHOD.

Required Privileges

To add or drop an attribute, you must have the UDTTYPE or UDTMETHOD privilege on the SYSUDTLIB database.

To add or drop a method, you must have the UDTMETHOD privilege on the SYSUDTLIB database.

Privileges Granted Automatically

None.

ALTER TYPE Syntax

```
ALTER TYPE [SYSUDTLIB.] UDT_name {
  [ ADD | DROP ] { attribute_clause | method_clause | specific_method_clause } |
  COMPILE [ONLY]
} [;]
```

attribute_clause

```
ATTRIBUTE attribute_specification [, ...]
```

method_clause

```
[ INSTANCE | CONSTRUCTOR ] METHOD [SYSUDTLIB.] method_name
[ ( data_type_or_UDT_name [, ...] ) ]
```

```

{ returns_clause |
  LANGUAGE { C | CPP } |
  PARAMETER STYLE { SQL | TD_GENERAL } |
  [NOT] DETERMINISTIC |
  NO SQL
}

```

specific_method_clause

```

SPECIFIC METHOD [SYSUDTLIB.] specific_method_name
  [ ( data_type_or_UDT_name [,...] ) ]
  FOR [SYSUDTLIB.] UDT_name

```

returns_clause

```

RETURNS data_type_or_UDT_name
  CAST FROM data_type_or_UDT_name

```

attribute_specification

```

attribute_name data_type_or_UDT_name

```

data_type_or_UDT_name

```

{ data_type | [SYSUDTLIB.] UDT_name }

```

data_type

```

{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [ (fractional_seconds_precision) ] [WITH TIME
ZONE] |

  INTERVAL YEAR [ (precision) ] [TO MONTH] |

  INTERVAL MONTH [ (precision) ] |

  INTERVAL DAY [ (precision) ]
  [TO { HOUR | MINUTE | SECOND [ (fractional_seconds_precision) ] } ] |

```



```

INTERVAL HOUR [(precision)]
    [TO { MINUTE | SECOND [(fractional_seconds_precision)] }] |

INTERVAL MINUTE [(precision)] [TO SECOND
[(fractional_seconds_precision)]] |

INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) |

PERIOD (DATE) |

PERIOD ( { TIME | TIMESTAMP } [(precision)] [WITH TIME ZONE] ) |

REAL |

DOUBLE PRECISION |

FLOAT [ (integer) ] |

NUMBER [ ( { integer | *} [, integer]... ) ] |

{ DECIMAL | NUMERIC } [ ( integer [, integer]... ) ] |

{ CHAR | BYTE | GRAPHIC } [ (integer) ] |

{ VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [ (integer) ] |

LONG VARCHAR |

LONG VARGRAPHIC |

{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
    ( integer [ G | K | M ] ) |

{ XML | XMLTYPE } |

JSON [ ( integer ) ] [ CHARACTER SET { UNICODE | LATIN } ] |

[SYSUDTLIB.] { UDT_name | ST_Geometry | MBR | ARRAY_name
| VARRAY_name }
}

```

ALTER TYPE Syntax Elements

UDT_name

The name of the UDT, ARRAY, or VARRAY whose definition is to be altered.

SYSUDTLIB.

The specification of SYSUDTLIB is optional.

COMPILE

Do all of the following:

- Recompile the code source for the specified type.
- Generate the object code.
- Recreate the appropriate .so file.
- Distribute the recreated .so file to all nodes of the system.

Vantage replaces the existing type with the recompiled version.

COMPILE ONLY

Do both of the following:

- Recompile the code source for the specified type.
- Generate the object code.

If you specify COMPILE ONLY, Vantage does not generate or distribute the .so file.

The system does replace the existing type with the recompiled version.

ADD ATTRIBUTE *attribute_name*

Add the attribute named *attribute_name* to the definition of the structured UDT named *UDT_name*.

You cannot specify this option for an ARRAY or VARRAY type.

You cannot add more than 512 attributes at a time.

You cannot specify a character server data set of KANJI1.

data_type

The data type of the attribute to be added.

UDT_name

The user-defined type to which the attribute is added.

SYSUDTLIB.

The specification of SYSUDTLIB is optional.

DROP ATTRIBUTE *attribute_name*

Drop the attribute named *attribute_name* from the definition of the structured UDT named *UDT_name*.

You cannot specify this option for an ARRAY or VARRAY type.

ADD METHOD

Add the instance method signature for the method named *method_name* to the definition of the UDT named *UDT_name*.

You cannot specify this option for an ARRAY or VARRAY type.

INSTANCE

The specification of INSTANCE is optional.

INSTANCE METHOD is the default.

CONSTRUCTOR

Add the constructor method signature for the method named *method_name* to the definition of the UDT named *UDT_name*.

SYSUDTLIB.

The specification of SYSUDTLIB is optional.

method name

You must use the CREATE METHOD statement to create the body for *method_name* before you can invoke it. See [CREATE METHOD](#). See *Teradata Vantage™ - SQL Fundamentals*, B035-1141 for the rules for naming database objects.

data_type

The data type list for the method to be added.

SYSUDTLIB. UDT_name

The specification of SYSUDTLIB is optional.

You cannot specify a character server data set of KANJI1.

RETURNS *data_type*

The list of the data types returned by *method_name*.

You cannot specify this option for an ARRAY or VARRAY type.

RETURNS SYSUDTLIB.UDT_name

The specification of SYSUDTLIB is optional.

CAST FROM

The result type returned by the method that is to be converted to the type specified by the RETURNS clause.

Example:

```
...RETURNS DECIMAL(9,5) CAST FROM FLOAT...
```

data type

Whenever a LOB that requires data type conversion is passed to a method, the LOB must first be materialized for the conversion to take place.

SYSUDTLIB

Optional containing database name, which is always SYSUDTLIB.

UDT_name

Name of the UDT.

LANGUAGE

The code for the language in which the external routine for *method_name* is written:

- C specifies the C language.
- CPP specifies the C++ language.

You cannot specify this option for an ARRAY or VARRAY type.

PARAMETER STYLE

The parameter passing style for *method_name*.

- SQL allows the code body to indicate null data.

This is the default parameter style.

- TD_GENERAL does not allow the code body to indicate null data.

You cannot specify this option for an ARRAY or VARRAY type.

DETERMINISTIC

The *method_name* returns the same results for identical inputs.

You cannot specify this option for an ARRAY or VARRAY type.

NOT DETERMINISTIC

The *method_name* does not necessarily return the same results for identical inputs.

You cannot specify this option for an ARRAY or VARRAY type.

NO SQL

Specifies whether SQL language calls are permitted within the user-written external routine for *method_name*.

The only valid specification is NO SQL.

You cannot specify this option for an ARRAY or VARRAY type.

ADD SPECIFIC METHOD

Add the specific method named *specific_method_name* to the definition of the UDT named *UDT_name*.

You cannot specify this option for an ARRAY or VARRAY type.

SYSUDTLIB.

The specification of SYSUDTLIB is optional.

specific_method_name

Name of the specific method to add to the definition of the UDT named *UDT_name*.

data_type

The data type list for the *specific_method_name*.

SYSUDTLIB.*UDT_name*

The specification of SYSUDTLIB is optional.

You cannot specify a character server data set of KANJI1.

FOR *UDT_name*

The name of the UDT to associate with the specific method.

The specification of SYSUDTLIB is optional.

DROP INSTANCE METHOD

Drop the instance method named *method_name* from the definition of the UDT named *UDT_name*.

You cannot specify this option for an ARRAY or VARRAY type.

You cannot drop observer or mutator methods.

To replace the external routine for a method, use the REPLACE METHOD statement (see [REPLACE METHOD](#)).

INSTANCE is the default.

SYSUDTLIB.method_name

The specification of SYSUDTLIB is optional.

method_name

The instance method to drop from the definition of the UDT named *UDT_name*.

DROP CONSTRUCTOR METHOD

Drop the constructor method named *method_name* from the definition of the UDT named *UDT_name*.

You cannot specify this option for an ARRAY or VARRAY type.

You cannot drop observer or mutator methods.

To replace the external routine for a method, use the REPLACE METHOD statement (see [REPLACE METHOD](#)).

SYSUDTLIB.method_name

The specification of SYSUDTLIB is optional.

method_name

Name for the method whose signature being added to the type definition for *UDT_name*.

data_type

The data type of the attribute to be dropped.

You cannot specify a character server data set of KANJI1.

UDT_name

User-defined type to be dropped.

DROP SPECIFIC METHOD SYSUDTLIB.*specific_method_name*

Drop the specific method named *specific_method_name* from the definition of the UDT named in the FOR *UDT_name* clause.

The specification of SYSUDTLIB is optional.

When you drop a method definition, the system also deletes all files associated with that method.

You cannot specify this option for an ARRAY or VARRAY type.

To replace the external routine for a method, use the REPLACE METHOD statement (see [REPLACE METHOD](#)).

data_type

The data type of the attribute to be dropped.

UDT_name

User-defined type to be dropped.

FOR *UDT_name*

The name of the UDT from which the association with the specified set of method signatures is to be dropped.

You cannot specify this option for an ARRAY or VARRAY type.

Examples

Example: Using the ALTER TYPE Statement with the COMPILE Option

The following example recompiles the UDT named *address*, generates the appropriate object code, generates the new .so library, and redistributes the file to all system nodes.

```
ALTER TYPE address COMPILE;
```

The following example recompiles the one-dimensional ARRAY *phonenumbers_ary*, generates the appropriate object code, generates the new .so library, and redistributes the file to all system nodes.

```
ALTER TYPE phonenumbers_ary COMPILE;
```

Example: Using the ALTER TYPE Statement with the COMPILE ONLY Option

The following example recompiles the UDT named *address* and generates the appropriate object code, but does not generate the new .so file, or redistribute the file to all system nodes.

```
ALTER TYPE address COMPILE ONLY;
```

The following example recompiles the multidimensional ARRAY named *seismic_cube* and generates the appropriate object code, but does not generate the new .so file, or redistribute the file to all system nodes.

```
ALTER TYPE seismic_cube COMPILE ONLY;
```

Example: Adding an Attribute to a Structured UDT

The following example adds an attribute named *country* to the structured UDT named *address*.

```
ALTER TYPE address      /* Add attribute to structured type */
ADD ATTRIBUTE country VARCHAR(15);
```

Example: Adding Multiple Attributes to a Structured UDT

The following example adds three attributes named *country*, *continent*, and *zone* to the structured UDT named *address*:

```
ALTER TYPE address      /* Add 3 attributes to structured type */
ADD ATTRIBUTE country   VARCHAR(15),
              continent VARCHAR(20),
              zone       INTEGER;
```

Example: Adding Multiple Instance Methods to a Structured UDT

The following example adds two instance methods named *toYen()* and *toPeso()* to the structured UDT named *euro*:

```
ALTER TYPE euro          /* Add 2 original methods */
ADD INSTANCE METHOD
  toYen()
  RETURNS yen
```



```

LANGUAGE C
PARAMETER STYLE SQL
DETERMINISTIC
NO SQL,
toPeso()
RETURNS peso
LANGUAGE C
PARAMETER STYLE SQL
DETERMINISTIC
NO SQL;

```

Example: Adding an Instance Method to a Structured UDT

The following example adds a new instance method named *toYen()* to the structured UDT named *euro*:

```

ALTER TYPE euro    /* Add original method */
ADD INSTANCE METHOD toYen()
RETURNS yen
LANGUAGE C
PARAMETER STYLE SQL
DETERMINISTIC
NO SQL;

```

Example: Dropping an Attribute From a Structured UDT

The following example drops the attribute named *country* from the structured UDT named *address*:

```

ALTER TYPE address    /* Drop attribute of structured type */
DROP ATTRIBUTE country;

```

Example: Dropping a Method From a Structured UDT Using Its Specific Method Name

The following example drops the method having the specific method name *polygon_mbr* from the structured UDT named *polygon*:

```

ALTER TYPE polygon    /* Drop method */
DROP SPECIFIC METHOD SYSUDTLIB.polygon_mbr;

```

Related Information

- [ALTER TYPE](#) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- [ALTER METHOD](#)
- [CREATE AUTHORIZATION and REPLACE AUTHORIZATION](#)
- [CREATE METHOD](#)
- [CREATE ORDERING and REPLACE ORDERING](#)
- [DROP AUTHORIZATION](#)
- [HELP FUNCTION](#)
- [HELP METHOD](#)
- [SHOW object](#)

See *Teradata Vantage™ - SQL Data Control Language*, B035-1149 for information about the UDTTYPE, UDTMETHOD, UDTUSAGE, and EXECUTE FUNCTION privileges, particularly the UDT-related and EXECUTE FUNCTION privileges for GRANT (SQL Form).

Also see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147 for information about how to write user-defined external routines.

DROP TYPE

Drops the definition for a specified UDT.

Before you can drop a UDT definition, the UDT:

- Cannot be directly or indirectly referenced by any database object in the system.
- Cannot be used as the data type of a column in any table in any database in the system.
- Cannot be an attribute of structured UDT.
- Cannot be referenced by any user-defined cast, method, or UDF in any database in the system.
- Cannot have an ordering or transform definition.

Note:

These prerequisites also apply to one-dimensional and multidimensional ARRAY and VARRAY types.

When you drop a UDT definition, the system also drops:

- All methods associated with the specified UDT.
- All external routines and libraries associated with the dropped methods.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Required Privileges

You must have either the UDTTYPE privilege or the UDTMETHOD privilege on the SYSUDTLIB database to drop a UDT.

DROP TYPE Syntax

```
DROP TYPE [ database_name. ] user_defined_type_name [;]
```

DROP TYPE Syntax Elements

database_name

Name of the containing database for *user_defined_type_name*.

This is always SYSUDTLIB.

user_defined_type_name

Name of the user-defined type, whether ARRAY/VARRAY, distinct, or structured, whose definition is to be dropped from the dictionary.

Before you drop a UDT, you must ensure that you resolve any dependencies that exist on it. You can execute the system macro SYSUDTLIB.HelpDependencies to find any existing dependencies for a type you intend to drop.

DROP TYPE Examples

Example: Dropping the Definition of a UDT 1

This example drops the definition of the UDT named *employee*.

```
DROP TYPE SYSUDTLIB.employee;
```

Example: Dropping the Definition of a UDT 2

This example drops the definition of the UDT named *circle*.

```
DROP TYPE SYSUDTLIB.circle;
```

Related Information

- CREATE TYPE (ARRAY Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184

- CREATE TYPE (Distinct Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- CREATE TYPE (Structured Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184

DROP *storage_format* SCHEMA

Drops a schema.

Required Privileges

You must have the DROP DATASET SCHEMA privilege on the schema or database containing the schema.

DROP *storage_format* SCHEMA Syntax

```
DROP storage_format SCHEMA [SYSUDTLIB.] schema_name [;]
```

DROP *storage_format* SCHEMA Syntax Elements

storage_format

Storage format of the DATASET type for the schema being dropped.

SYSUDTLIB

Name of the database containing the schema. Schemas that you create are registered in the SYSUDTLIB database.

schema_name

Name of the schema.

DROP *storage_format* SCHEMA Example

This example drops a schema in the Avro storage format of the DATASET type:

```
DROP Avro SCHEMA chemDatasetSchema;
```

DROP CAST

Drops a cast definition for a UDT from the data dictionary.

You can also drop the system-generated casts for distinct UDTs. The system does not drop the associated external cast routine when you perform DROP CAST.

You should not drop a cast if it is referenced by a view, stored procedure, trigger, or macro. There are no restrictions on dropping a cast. The system always honors a DROP CAST request whether it is referenced by a view, stored procedure, trigger, or macro. However, after you drop a cast, any attempt to use a view, stored procedure, trigger, or macro that references the dropped cast returns an error to the requestor.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Required Privileges

You must have either the UDTTYPE or the UDTMETHOD privilege on the SYSUDTLIB database to drop a user-defined cast for a UDT.

DROP CAST Syntax

```
DROP CAST [ database_name. ] ( source_data_type AS target_data_type ) [;]
```

DROP CAST Syntax Elements

source_data_type

The source data type from the cast definition.

Either *source_data_type* or *target_data_type* or both must be a UDT.

target_data_type

The target data type from the cast definition.

Either *target_data_type* or *source_data_type* or both must be a UDT.

DROP CAST Examples

Example: Dropping the Cast of a Geospatial UDT to a Geospatial UDT

The following example drops the cast of UDT *circle* to UDT *ellipse*.

```
DROP CAST (circle AS ellipse);
```

Example: Dropping the Cast of a Geospatial UDT to a VARCHAR

The following example drops the cast of UDT *circle* to VARCHAR(32).

```
DROP CAST (circle AS VARCHAR(32));
```

Example: Dropping the System-Generated Cast from Source Data Type to Distinct UDT

The following example drops the system-generated cast of DECIMAL to the UDT *euro*.

```
DROP CAST (DECIMAL AS euro);
```

Example: Dropping the System-Generated Cast from Distinct UDT to Source Data Type

The following example drops the system-generated cast of UDT *euro* to DECIMAL.

```
DROP CAST (euro AS DECIMAL);
```

Related Information

- CREATE TYPE (ARRAY Form), CREATE TYPE (Distinct Form), CREATE TYPE (Structured Form), CREATE CAST, REPLACE CAST, and DROP CAST in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - SQL External Routine Programming*, B035-1147

DROP ORDERING

Drops the ordering definition for a UDT.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Required Privileges

You must have either the UDTTYPE or the UDTMETHOD privilege on the SYSUDTLIB database to drop a user-defined ordering for a UDT.

DROP ORDERING Syntax

```
DROP ORDERING [ database_name. ] user_defined_type_name [;]
```

DROP ORDERING Syntax Elements***database_name***

Containing database for *user_defined_type_name*.

This is always SYSUDTLIB.

user_defined_type_name

Name of the UDT on which the ordering is to be dropped.

DROP ORDERING Examples

Example: Dropping the Ordering for a UDT 1

The following example drops the ordering for the UDT named *circle* with the restriction that no table in the database can be using the *circle* UDT in any of its column definitions.

```
DROP ORDERING FOR circle;
```

Example: Dropping the Ordering for a UDT 2

The following example drops the ordering for the UDT named *address* with the restriction that no table in the database can be using the *address* UDT in any of its column definitions.

```
DROP ORDERING FOR address;
```

Related Information

- CREATE ORDERING and REPLACE ORDERING in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - SQL External Routine Programming*, B035-1147

DROP TRANSFORM

Drop transform groups for a specified UDT.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Required Privileges

You must have either the UDTTYPE privilege or the UDTMETHOD privilege on the SYSUDTLIB database to drop transform groups.

DROP TRANSFORM Syntax

```
DROP TRANSFORM [ database_name. ] { transform_group_name | ALL }  
FOR user_defined_type_name [ ; ]
```

DROP TRANSFORM Syntax Elements

database_name

Name of the database that contains the UDT for which the transform group is to be dropped.

This is always SYSUDTLIB.

transform_group_name

Name of the transform to be dropped.

ALL

Drops all transform groups for the user defined data type.

user_defined_type_name

Name of the UDT for which the transform is defined.

DROP TRANSFORM Examples

Example: Dropping a Transform Group for a UDT

This example drops the transform group named *c_routine* from the UDT named *address* with the restriction that no table in the entire system can be using *address* as the data type for any of its columns.

```
DROP TRANSFORM c_routine FOR SYSUDTLIB.address;
```

Example: Drop All Transform Groups for a UDT

This statement drops all transform groups for the XMLD_STRUCT2 user defined data type.

```
DROP TRANSFORM ALL FOR XMLD_STRUCT2;
```

Related Information

- CREATE TRANSFORM and REPLACE TRANSFORM in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184

HELP TYPE

Displays the attributes for a UDT and, optionally, all the attributes of a structured UDT or all the methods associated with the UDT.

HELP TYPE ATTRIBUTE is not valid for Teradata internal UDT types such as Geospatial types, nor is it valid for Period types.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have at least one of the following privileges to perform `HELP TYPE`:

- At least one privilege on the `SYSUDTLIB` database.
- The `UDTUSAGE` privilege on the UDT or the `ARRAY/VARRAY` type.

Use the `SHOW` privilege to enable a user to perform `HELP` or `SHOW` requests only against the specified UDT or `ARRAY/VARRAY` type.

HELP TYPE Syntax

```
HELP TYPE [SYSUDTLIB.] UDT_name [ ATTRIBUTE | METHOD ] [;]
```

HELP TYPE Syntax Elements

SYSUDTLIB

Name of the containing database for *UDT_name*. This is always `SYSUDTLIB`.

UDT_name

Name of a UDT or `ARRAY/VARRAY` type for which a help report is requested.

ATTRIBUTE

Composition of the specified UDT or `ARRAY/VARRAY` type is the help information requested.

- If the UDT is distinct, Vantage returns one row that describes the predefined source data type.
- If the UDT is structured, Vantage returns one row for each attribute of the UDT.
- If the type is `ARRAY` or `VARRAY`, Vantage returns one row that describes the element type of the `ARRAY` or `VARRAY`.

METHOD

Report of the methods associated with *UDT_type* is requested. One row is returned for each method associated with the UDT, including one row for each system-generated observer or mutator method.

Usage Notes

HELP COLUMN Report on a Unicode View Differs From a Compatibility View

The result of a HELP COLUMN statement using a Unicode view differs from the results for the same column using a Compatibility view. For details, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Note:

Only the first 30 characters of object names are available from Compatibility views. Teradata strongly encourages you to use Unicode views instead. For a list of the Compatibility views that have been deprecated, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

In this example, HELP COLUMN returns information for the *tablename* column from the Unicode TablesV view:

```
HELP COLUMN dbc.tablesv.tablename;
*** Help information returned. One row.
Column Name TableName
                Type CV
                Nullable N
                Format X(128)
                Max Length      256
Decimal Total Digits ?
Decimal Fractional Digits ?
                Range Low      ?
                Range High     ?
                UpperCase N
Table/View? V
Indexed? Y
Unique? N
Primary? S
                Title ?
Column Constraint ?
                Char Type  2
IdCol Type ? ?
UDT Name ? ?
```

The following HELP COLUMN request uses the DBC.Tables view instead of the DBC.TablesV view:

```

HELP COLUMN dbc.tables.tablename;
*** Help information returned. One row.
*****Column Name TableName
                Type CF
                Nullable Y
                Format X(30)
                Max Length          30
                Decimal Total Digits ?
                Decimal Fractional Digits ?
                Range Low           ?
                Range High          ?
                UpperCase N
                Table/View? V
                Indexed? N
                Unique? ?
                Primary? ?
                Title ?
                Column Constraint ?
                Char Type 1
                IdCol Type ? ?
                UDT Name ? ?

```

The differences between this result and the result using the DBC.TablesV view are highlighted in boldface. The Type, Format, Max Length, and Char Type are different because the Compatibility view casts the object name to 30 fixed characters in the Kanji1, or Latin, server character set.

Text strings work in a similar fashion. For example, the ColumnTitle column of the Columns compatibility view and the ColumnsV Unicode view returns different values for Type, Format, Max Length, and Char Type.

HELP TYPE Examples

Example: HELP TYPE with No Options Specified

The following example shows the standard report returned by HELP TYPE when you do not specify any options.

```

HELP TYPE euro;
                Name euro
                Internal Type UD
                External Type D
                Max Length          8
                Decimal Total Digits ?

```

```

Decimal Fractional Digits      ?
    Contains LOB N
    Ordering F
Ordering Category M
    Ordering Routine SYSUDTLIB.euro_to_decimal
    Cast Y
    Transform Y
    Method Y
    Char Type 1
    Array (Y/N) N
    Dimensions                  ?
    Element Type ?
    UDT Name ?
    Array Scope ?

```

Example: HELP TYPE for a Distinct UDT Type

This example shows the standard report returned by HELP TYPE for a distinct UDT type.

Suppose you create the following ARRAY type.

```

CREATE TYPE mydistinct AS
INTEGER FINAL;

```

This type returns the following report to a HELP TYPE request.

```

HELP TYPE mydistinct;
    Name mydistinct
    Internal Type UD
    External Type I
    Max Length          4
    Decimal Total Digits ?
    Decimal Fractional Digits ?
    Contains LOB N
    Ordering F
    Ordering Category M
    Ordering Routine SYSTEM
    Cast Y
    Transform Y
    Method N
    Char Type ?
    Array (Y/N) N
    Dimensions          ?

```

```

Element Type ?
    UDT Name ?
    Array Scope ?

```

Example: HELP TYPE for a UDT with the ATTRIBUTE Option

The following examples show the standard report returned by HELP TYPE when you specify the ATTRIBUTE option.

The first example shows how the name for the structured UDT named address is returned with full qualification.

```

HELP TYPE address ATTRIBUTE;
    Attribute Name street
        Type CV
        Max Length          20
        Decimal Total Digits ?
        Decimal Fractional Digits ?
        Range Low ?
        Range High ?
        Char Type 1
        UDT Name SYSUDTLIB.UDTName(assuming street is a UDT)
    Attribute Name zip
        Type CF
        Max Length 5
        Decimal Total Digits ?
        Decimal Fractional Digits ?
        Range Low ?
        Range High ?
        Char Type 1
        UDT Name ?

```

The second example shows a report returned when the specified structured type *insv_structured_smallint* has multiple attributes, two of which are themselves structured UDTs.

```

HELP TYPE insv_structured_smallint ATTRIBUTE;
    Attribute Name Attribute1
        Type I2
        Max Length          2
        Decimal Total Digits ?
        Decimal Fractional Digits ?
        Range Low ?
        Range High ?

```

```

Char Type ?
UDT Name ?
Attribute Name Attribute2
Type UD
Max Length 4
Decimal Total Digits ?
Decimal Fractional Digits ?
Range Low ?
Range High ?
Char Type ?
UDT Name SYSUDTLIB.UDTINT
Attribute Name Attribute3
Type UD
Max Length 50
Decimal Total Digits ?
Decimal Fractional Digits ?
Range Low ?
Range High ?
Char Type ?
UDT Name SYSUDTLIB.UDTBLOB10

```

Example: HELP TYPE with METHOD Option

The following example shows the report produced by submitting HELP TYPE with the METHOD option. Suppose you have the following type definition.

```

CREATE TYPE my_structured_type AS (
  attribute_1 DECIMAL(8,2),
  attribute_2 REAL,
  attribute_3 VARCHAR(20) )
NOT FINAL
/* Signature of constructor method my_structured_type */
CONSTRUCTOR METHOD my_structured_type( P1 INTEGER )
RETURNS my_structured_type
SPECIFIC my_structured_type_c1
SELF AS RESULT
LANGUAGE C
PARAMETER STYLE SQL
DETERMINISTIC
NO SQL,
/* Signature of instance method Method1 */
METHOD method_1(VARCHAR(20))

```

```

RETURNS INTEGER
LANGUAGE C
NO SQL
PARAMETER STYLE SQL;

```

Now you submit the following HELP TYPE request for *my_structured_type* with the METHOD option.

```

HELP TYPE my_structured_type METHOD;
*** Help information returned. 8 rows.
*** Total elapsed time was 1 second.
  Method Name  ATTRIBUTE3
  Specific Name MYSTRUCTUR_ATTRIBUTE3_04E1_M
  Method Type  M
  Null Call    Y
  Exec Mode    NP
Deterministic  Y
  Method Name  ATTRIBUTE3
  Specific Name MYSTRUCTUR_ATTRIBUTE3_04E0_O
  Method Type  O
  Null Call    N
  Exec Mode    NP
Deterministic  Y
  Method Name  ATTRIBUTE2
  Specific Name MYSTRUCTUR_ATTRIBUTE2_04DF_M
  Method Type  M
  Null Call    Y
  Exec Mode    NP
Deterministic  Y
  Method Name  ATTRIBUTE2
  Specific Name MYSTRUCTUR_ATTRIBUTE2_04DE_O
  Method Type  O
  Null Call    N
  Exec Mode    NP
Deterministic  Y
  Method Name  ATTRIBUTE1
  Specific Name MYSTRUCTUR_ATTRIBUTE1_04DD_M
  Method Type  M
  Null Call    Y
  Exec Mode    NP
Deterministic  Y
  Method Name  ATTRIBUTE1
  Specific Name MYSTRUCTUR_ATTRIBUTE1_04DC_O
  Method Type  O
  Null Call    N

```

```

      Exec Mode      NP
Deterministic      Y
      Method Name    MYSTRUCTUREDTYPE
      Specific Name  MYSTRUCTUREDTYPE_C1
      Method Type    C
      Null Call      Y
      Exec Mode      P
Deterministic      Y
      Method Name    METHOD1
      Specific Name  MYSTRUCTUR_METHOD1_04DB_R
      Method Type    I
      Null Call      Y
      Exec Mode      P
Deterministic      N

```

Example: HELP TYPE for a One-Dimensional ARRAY

This example shows the standard report returned by HELP TYPE for a single-dimensional ARRAY.

Suppose you create the following ARRAY type.

```

CREATE TYPE arrayvec AS
INTEGER ARRAY[1000];

```

This type returns the following report to a HELP TYPE request.

```

HELP TYPE arrayvec;
      Name arrayvec
      Internal Type A1
      External Type CV
      Max Length      12,001
      Decimal Total Digits      ?
      Decimal Fractional Digits  ?
      Contains LOB N
      Ordering F
      Ordering Category M
      Ordering Routine LOCAL
      Cast N
      Transform Y
      Method Y
      Char Type 1
      Array (Y/N) Y
      Dimensions      1

```



```

Element Type I
      UDT Name ?
      Array Scope [1:1000]

```

Example: HELP TYPE for an ARRAY with the ATTRIBUTE Option

This example extends the previous example to specify the `ATTRIBUTE` option for the `HELP TYPE` request. The `ARRAY` type used for the example is `arrayvec`, which was defined in [Example: HELP TYPE for a One-Dimensional ARRAY](#).

The `HELP TYPE` request for the `ARRAY` type `arrayvec` specified with the `ATTRIBUTE` option returns the following report.

```

HELP TYPE arrayvec ATTRIBUTE;
      Attribute Name arrayvec
            Type I
            Max Length          4
      Decimal Total Digits      ?
      Decimal Fractional Digits ?
            Range Low           ?
            Range High          ?
            Char Type 1
            UDT Name ?

```

Example: HELP TYPE with No Options for a One-Dimensional Array

Suppose you create either of the following equivalent one-dimensional `ARRAY/VARRAY` types, the first using Teradata format and the second using Oracle-compatible format.

```

CREATE TYPE phonenumbers_ary AS
CHARACTER(10) ARRAY[5];
CREATE TYPE phonenumbers_ary AS
VARRAY(5) OF CHARACTER(10);

```

A `HELP TYPE` request for `phonenumbers_ary` returns the following information.

```

HELP TYPE phonenumbers_ary;
*** Help information returned. 19 rows.
*** Total elapsed time was 1 second.
Name phonenumbers_ary
      Internal Type A1
      External Type CV

```

Max Length	121
Array (Y/N)	Y
Dimensions	1
Element Type	CF
UDT Name	?
Array Scope	[1:5]
Total Digits	?
Fractional Digits	?
Contains LOB	N
Ordering	F
Ordering Category	M
Ordering Routine	LOCAL
Cast	N
Transform	Y
Method	Y
Char Type	?

Related Information

- [SHOW object](#)
- *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - SQL External Routine Programming*, B035-1147

HELP *storage_format* SCHEMA

Displays the name of a schema, the storage format of the DATASET type associated with the schema, and the length of the schema. The length is listed in UNICODE characters.

Required Privileges

None.

HELP *storage_format* SCHEMA Syntax

```
HELP storage_format SCHEMA [SYSUDTLIB.] schema_name [;]
```

HELP *storage_format* SCHEMA Syntax Elements

storage_format

Storage format of the DATASET type for the schema.

SYSUDTLIB

Database containing the schema.

schema_name

Name of the schema.

HELP *storage_format* SCHEMA Example

This example displays the HELP information for the schema from the CREATE *storage_format* SCHEMA example:

```
HELP Avro SCHEMA chemDatasetSchema;
      SchemaName chemDatasetSchema
      StorageFormat Avro
      Length 201
```

HELP CAST

Returns the available cast operations for the specified UDT.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have at least one privilege on the SYSUDTLIB database to perform HELP CAST.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against SYSUDTLIB.

HELP CAST Syntax

```
HELP CAST [SYSUDTLIB.] UDT_name [ SOURCE | TARGET ] [;]
```

HELP CAST Syntax Elements**SYSUDTLIB**

The containing database for *UDT_name*.

UDT_name

Name of the UDT for which information about casts is requested.

If you do not specify SOURCE or TARGET, both are reported.

SOURCE

Reports all cast operations that have the specified data type as their source data type.

TARGET

Reports all cast operations that have the specified data type as their target data type.

HELP CAST Examples

Example: Source Castings Only

The following example reports the source castings for the UDT named *euro*:

```
HELP CAST SYSUDTLIB.euro SOURCE;
Source      Target      Cast Routine      As Assignment
-----
euro        DECIMAL(10,2)  System            YES
euro        us_dollar    EurotoUS          NO
```

Example: Target Castings Only

The following example reports the target castings for the UDT named *euro*:

```
HELP CAST SYSUDTLIB.euro TARGET;
Source      Target      Cast Routine      As Assignment
-----
DECIMAL(10,2) euro        System            YES
us_dollar   euro        UstoEuro          YES
```

Example: All Castings for a UDT That Has Multiple Casting Pairs

The following example shows the output of `HELP CAST euro` with no options:

```
HELP CAST euro;
Source      Target      Cast Routine      As Assignment
-----
euro        DECIMAL(10,2)  System            YES
DECIMAL(10,2) euro        System            YES
us_dollar   euro        UstoEuro          YES
euro        us_dollar    EurotoUS          NO
```

Note the following things about this report:

- Because this UDT has two casting pairs (euro:DECIMAL and euro:us_dollar), there are four rows in the report.
- The euro:DECIMAL casting pair was system-generated (its cast routine in both cases is named System), while the euro:us_dollar casting pair was user-defined.
- There is no implicit casting for the euro:us_dollar cast because it was not defined with the AS ASSIGNMENT option.

Example: All Castings For A UDT That Has A Single Casting Pair

The following example reports both the source and target castings for the UDT named *address*:

```
HELP CAST address;
```

Source	Target	Cast Routine	As Assignment

address	VARCHAR(80)	address_2_char	YES
VARCHAR(80)	address	char_2_address	YES

Note the following things about this report:

- The castings from address to VARCHAR(80) and from VARCHAR(80) to address are complementary to one another.
- Both casting routines are user-defined, because neither is named System.
- This UDT has only one casting pair defined for it, neither of which is named System, which suggests that it is probably a structured type.

Castings for structured types are not system-generated, while the system generates default casts for distinct types. Because it is also possible to drop the system-generated casts for a distinct UDT and replace them with user-defined casts, the inference that the casting reported in this example is for a structured UDT cannot be made with certainty. See CREATE CAST and REPLACE CAST.

Related Information

- [SHOW object](#)
- *Teradata Vantage™ - SQL External Routine Programming*, B035-1147

HELP TRANSFORM

Reports the fromsql and tosql transform routines for a specified UDT or predefined data type.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have at least one privilege on the SYSUDTLIB database to perform HELP TRANSFORM.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against the specified UDT transform.

HELP TRANSFORM Syntax

```

HELP TRANSFORM [ database_name. | user_name. ] UDT_name [;]

```

HELP TRANSFORM Syntax Elements

- UDT_name**
Name of the UDT for which the transform group information is being reported.
- SYSUDTLIB**
Name of the containing database for the transform group being reported. This is always SYSUDTLIB.

Examples

Example: HELP TRANSFORM for a UDT

The following example reports the transform group name (client_IO), predefined data type to which the UDT is converted (VARCHAR(40)), fromsql external routine name (toString) and tosql external routine name (stringToAddress) for the UDT named address.

```

HELP TRANSFORM address;
Group      Predefined Type from-sql      to-sql
-----
client_IO  VARCHAR(40)      SYSUDTLIB.toString  SYSUDTLIB.stringToAddress

```

Related Information

- Teradata Vantage™ - SQL Data Definition Language Detailed Topics
- Teradata Vantage™ - SQL External Routine Programming, B035-1147

Database Statements

CREATE DATABASE

Creates a database in which other database objects can be created.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

CREATE DATABASE is functionally similar to the ANSI CREATE SCHEMA statement.

Required Privileges

To create a database, you must have the CREATE DATABASE privilege on the immediate owner database.

No privilege is needed to specify a DEFAULT JOURNAL TABLE in a database other than the database being created. However, when a table is created that specifies a journal, the user creating the table must have INSERT privilege on the journal table.

Privileges Granted Automatically

The privileges granted to the creator and owner on the new database are WITH GRANT OPTION.

The following privileges are automatically granted to a database when it is created:

- CHECKPOINT
- CREATE AUTHORIZATION
- CREATE DATABASE
- CREATE MACRO
- CREATE TABLE
- CREATE TRIGGER
- CREATE USER
- CREATE VIEW
- DELETE
- DROP AUTHORIZATION
- DROP DATABASE
- DROP FUNCTION
- DROP MACRO
- DROP TABLE
- DROP PROCEDURE
- DROP TABLE
- DROP TRIGGER

- DROP USER
- DROP VIEW
- DUMP
- EXECUTE
- INSERT
- RESTORE
- REVOKE
- SELECT
- STATISTICS
- UPDATE

Privileges Not Granted Automatically

The following privileges on itself are not automatically granted to a database when it is created:

- ALTER EXTERNAL PROCEDURE
- ALTER FUNCTION
- ALTER PROCEDURE
- CREATE DATABASE
- CREATE EXTERNAL PROCEDURE
- CREATE FUNCTION
- CREATE PROCEDURE
- CREATE USER
- DROP DATABASE
- DROP USER
- EXECUTE FUNCTION
- EXECUTE PROCEDURE

CREATE DATABASE Syntax

```
{ CREATE DATABASE | CD } name [ FROM database_name ] AS database_attribute
[[,]...] [;]
```

Syntax Elements

database_attribute

```
{ { PERMANENT | PERM | SPOOL | TEMPORARY } = { n | constant_expression
} [BYTES]
  [ SKEW = { constant_expression | DEFAULT } [ PERCENT ] ] |

ACCOUNT = 'account_string' |
```



```

DEFAULT MAP = { map_name | NULL } [ OVERRIDE [NOT] ON ERROR ] |
[NO] FALLBACK [PROTECTION] |
[ NO | DUAL ] [BEFORE] JOURNAL |
[ NO | DUAL | [NOT] LOCAL ] AFTER JOURNAL |

DEFAULT JOURNAL TABLE = [ database_name. ] table_name
}

```

CREATE DATABASE Syntax Elements

name

Name of the new database.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

FROM *database_name*

Name of the immediate owning user or database. The default is the user name associated with the current session.

PERMANENT

Number of bytes to be reserved for permanent storage for *database_name*. The space is taken from unallocated space in the database or user of the immediate owner.

You must specify a value for this option. There is no default.

Note:

You can specify a password for a user, but not for a database.

n

You can enter the number of bytes as an integer, decimal, or floating point value or as a constant expression whose evaluation determines the number of bytes. You can also enter the value using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

The value of *n* or the number of bytes determined from the evaluation of *constant_expression* cannot exceed the permanent space of the owner.

constant_expression

A constant expression is any SQL expression that does not make any column references. Specifying an appropriate constant expression for the PERM space size of a database enables Vantage to assign an optimal quantity of PERM space that scales to the size of your system by allocating PERM space on a per AMP basis.

When you specify a PERM space size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

Optional keyword that redundantly specifies the unit for the amount of space allowed.

SKEW

Keyword that you use to specify a skew limit for PERMANENT space. You can specify a skew limit percentage that allows the maximum AMP space usage to be above the per-AMP quota, that is, the system maximum space limit divided by the number of AMPs.

constant_expression

Constant expression or constant from 0 through 10000. Specify a value from 1 to 9999 to indicate an AMP-level limit, which is the per-AMP quota * $(1 + \text{permskewlimit}/100)$. Specify 0 to set space accounting to the per-AMP level, that is, no skew. A value of 10000 indicates unlimited skew, up to the system maximum space limit.

DEFAULT

Use the value of DBS Control DefaultPermSkewLimitPercent.

PERCENT

Optional keyword that you can include for readability to indicate that the *constant_expression* or DEFAULT keyword specifies a percentage of allowable skew.

SPOOL

Number of bytes to be allowed for spool and volatile temporary tables in *database_name*. The default is the largest value that is not greater than the immediate owner spool space and that is a multiple of the number of AMPs on the system.

n

You can enter the number of bytes as an integer, decimal, or floating point value or as a constant expression whose evaluation determines the number of bytes. You can also enter the value using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

n cannot exceed the spool space parameter in the profile of the creator. If no spool space is defined for that profile, then Vantage uses the spool space limit defined for the individual user-creator.

constant_expression

A constant expression is any SQL expression that does not make any column references. Specifying an appropriate constant expression for the SPOOL space size of a database enables Vantage to assign an optimal quantity of SPOOL that scales to the size of your system by allocating SPOOL space on a per AMP basis.

When you specify a SPOOL size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

Optional keyword that redundantly specifies the unit for the amount of space allowed.

SKEW

Keyword that you use to specify a skew limit for SPOOL space. You can specify a skew limit percentage that allows the maximum AMP space usage to be above the per-AMP quota, that is, the system maximum space limit divided by the number of AMPs.

constant_expression

Constant expression or constant from 0 through 10000. Specify a value from 1 to 9999 to indicate an AMP-level limit, which is the per-AMP quota * (1+*spool/skewlimit*/100). Specify 0 to set space accounting to the per-AMP level, that is, no skew. A value of 10000 indicates unlimited skew, up to the system maximum space limit.

DEFAULT

Use the value of DBS Control DefaultSpoolSkewLimitPercent.

PERCENT

Optional keyword that you can include for readability to indicate that the *constant_expression* or DEFAULT keyword specifies a percentage of allowable skew.

TEMPORARY

Definition for how many bytes are to be allowed by default for creating materialized global temporary tables by users within this database. The number of bytes can be an integer, decimal, or floating point value.

The space used by a global temporary table (table header only) is allocated from the PERM space of the database or user where the global temporary table resides. A CREATE GLOBAL TEMPORARY TABLE request creates table header for the table on each AMP, so the database or user must have adequate PERM space to accommodate the global temporary table header on each AMP. Table header size varies by table definition and the maximum size for a table header can be up to 1 Mbyte.

The space used by the materialized temporary table (table header plus possible data row) is allocated from the login user's temporary space. To materialize a temporary table, the user must have adequate temporary space on each AMP to accommodate the materialized temporary table header and the data row (if the AMP owns the row). Note that the user inherits temporary space from its immediate owner if temporary space was not specified when the user was created.

n

You can enter the number of bytes as an integer, decimal, or floating point value or as a constant expression whose evaluation determines the number of bytes. You can also enter the value using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

n cannot exceed the temporary space parameter in the profile of the owner. If no temporary space is defined for that profile, then the system uses the temporary space limit defined for the individual user-creator.

constant_expression

A constant expression is any SQL expression that does not make any column references. Specifying an appropriate constant expression for the TEMPORARY space size of a database enables Vantage to assign an optimal quantity of TEMPORARY that scales to the size of your system by allocating TEMPORARY space on a per AMP basis.

When you specify a TEMPORARY size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

Optional keyword that redundantly specifies the unit for the amount of space allowed.

SKEW

Keyword that you use to specify a skew limit for TEMPORARY space. You can specify a skew limit percentage that allows the maximum AMP space usage to be above the per-AMP quota, that is, the system maximum space limit divided by the number of AMPs.

constant_expression

Constant expression or constant from 0 through 10000. Specify a value from 1 to 9999 to indicate an AMP-level limit, which is the per-AMP quota * (1+*temp skew limit*/100). Specify 0 to set space accounting to the per-AMP level, that is, no skew. A value of 10000 indicates unlimited skew, up to the system maximum space limit.

DEFAULT

Use the value of DBS Control DefaultTempSkewLimitPercent.

PERCENT

Optional keyword that you can include for readability to indicate that the *constant_expression* or DEFAULT keyword specifies a percentage of allowable skew.

ACCOUNT = 'account_string'

Account to be charged for the space used by this database. If not specified, the account defaults to the default account of the immediate owner of the database.

Note:

When specifying an account string for a database, consider that a user session may default to the account_string specified for the database that owns the user if no other user or profile based account applies.

Each *account_string* must follow the standard Vantage naming rules. See *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

DEFAULT MAP

You can specify an existing contiguous or sparse map as the default map for the database.

You must have been granted the specified map unless the map is the same as the default map of the database creator.

map_name

Name of an existing contiguous or sparse map.

You cannot specify TD_DataDictionaryMap or TD_GlobalMap.

To specify a sparse map as the default map, you must be in the same secure zone as the sparse map and the sparse map must be in the same secure zone as the database.

OVERRIDE ON ERROR

Use the default map if an error occurs when the MAP clause is specified for a CREATE TABLE, CREATE JOIN INDEX, or CREATE HASH INDEX statement. For example, if the MAP clause specifies a nonexistent contiguous map or a sparse map in another secure zone, a warning message displays. The message indicates that the specified map was not used and lists the name of the default map used instead.

This is the default. If you do not specify a default map, the system-default map is used.

OVERRIDE NOT ON ERROR

Do not use the default map if an error occurs when the MAP clause is specified for a CREATE TABLE, CREATE JOIN INDEX, or CREATE HASH INDEX statement.

NULL

A default map is not associated with the database.

FALLBACK PROTECTION

Whether to create and store a duplicate copy of each table created in the new database.

The FALLBACK keyword used alone implies PROTECTION.

NO

Sets the default to not provide duplicate copy protection for data tables created in the database. This setting can be overridden for a particular data table when the table is created.

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

JOURNAL

The JOURNAL keyword without NO or DUAL implies single copy journaling.

The JOURNAL keyword without BEFORE implies both types (BEFORE and AFTER) of images.

Journal options are not supported for tables with row sizes greater than 64KB.

BEFORE

A before change image is maintained.

DUAL

Number of before change images to be maintained by default for each data table created in the new database.

If journaling is specified, a DUAL journal is maintained for data tables with FALLBACK protection.

NO

No journaling.

AFTER JOURNAL

Type of image to be maintained by default for data tables created in the new database. The JOURNAL keyword without AFTER implies BEFORE and AFTER images.

If only AFTER JOURNAL is specified, then a before change image is not maintained. If both types are specified, the two specifications must not conflict.

You can override this setting for individual tables when the table is created. See [CREATE TABLE and CREATE TABLE AS](#).

DUAL

If journaling is specified, a DUAL journal is maintained for data tables with FALLBACK protection.

LOCAL

Single after-image journal rows for non-fallback data tables are written on the same virtual AMP as the changed data rows. See the CREATE DATABASE “Local Journaling” topic in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

NOT LOCAL

Single after-image journal rows for non-fallback data tables are written on another virtual AMP in the cluster than the changed data rows.

NO

The default is no journaling.

DEFAULT JOURNAL TABLE

Default table that is to receive the journal images of data tables created in the new database.

A database can contain only one default journal table. However, any table in a particular database can use a journal table in a different database.

table_name

You must define a *table_name*. The table is created in the new database, if you do not specify a database. Otherwise, the table is created in the database you specify.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

database_name

If you specify another database, the database must exist and *table_name* must be defined as the default journal table.

Usage Notes

Journal Tables and the Default Map

If the CREATE DATABASE statement specifies a journal table and a default map, the journal table is created using this default map. The journal table map must be a contiguous map with the same AMPs as TD_GlobalMap.

If the CREATE DATABASE statement specifies a journal table without specifying a default map, the system determines the map for the journal table in the following order of precedence:

- Default map, if defined, for the profile of the database creator. See the [DEFAULT MAP](#) option of CREATE PROFILE.
- Default map, if defined, for the database creator. See the [DEFAULT MAP](#) option of CREATE USER.
- System-default map.

CREATE DATABASE Examples

Example: Creating a New Database

To create a new database for the Finance department, use the CREATE DATABASE request as follows.

```
CREATE DATABASE finance FROM sysadmin
AS PERMANENT = 60000000,
```



```

SPOOL = 120000000,
FALLBACK PROTECTION,
AFTER JOURNAL,
BEFORE JOURNAL,
DEFAULT JOURNAL TABLE = finance.journals,
ACCOUNT = 'ACCTG';

```

The *finance* database is created from the space available in *sysadmin*. The 60,000,000 value represents the amount of storage in bytes. To create a database, the initiator must have CREATE DATABASE privileges on the FROM user or database. The new database receives all privileges that have been granted to the initiator.

In the example, the FROM clause allocates space for *finance* from the *sysadmin* space rather than from user space; therefore, *sysadmin* is the immediate owner of *finance*, and *marks* is not in the hierarchy of ownership. However, *marks* is granted automatic creator privileges on *finance*, which include the privilege to create other objects in the space owned by *finance*.

The CREATE DATABASE statement does not include the PASSWORD and STARTUP clauses or the DEFAULT DATABASE and COLLATION options. Because these clauses and options affect the session environment, and only a user can establish a session, they do not apply to a database.

Example: Using a Constant Expression to Specify the PERM, SPOOL, and TEMPORARY Space for a Database

The following statement creates PERM space for database *production_development* with a size based on the constant expression $2,000,000 * (\text{HASHAMP}() + 1)$, SPOOL with a size based on the constant expression $2,000,000 * (\text{HASHAMP}() + 1)$, and TEMPORARY with a size based on the constant expression $2,000,000 * (\text{HASHAMP}() + 1)$. The expressions calculate the number of AMPs in the current system and scale the PERM, SPOOL, and TEMPORARY space for the *production_development* database to that size.

```

CREATE DATABASE production_development AS
  PERM = 2000000*(HASHAMP()+1),
  SPOOL = 2000000*(HASHAMP()+1),
  TEMPORARY = 2000000*(HASHAMP()+1);

```

Example: Creating a Database with a Permanent Space Skew Limit

On a system with 4 AMPs, you create a database with 1 gigabyte of permanent space. The per-AMP quota of permanent space is 250 megabytes. If you set the permanent space skew limit to 10%, any AMP is allowed a skew limit of 25 megabytes over the per-AMP quota of permanent space. An AMP can use up

to 275 megabytes of permanent space as long as total permanent space used does not exceed the 1 gigabyte global limit for permanent space.

```
CREATE DATABASE d1 AS
  PERM = 1e9 SKEW = 10 PERCENT;
```

Example: Creating a Database with a Spool Space Skew Limit

On a system with 4 AMPs, you create a database with 1 gigabyte of spool space. The per-AMP quota of spool space is 250 megabytes. If you set the spool space skew limit to 20%, any AMP is allowed a skew limit of 50 megabytes over the per-AMP quota of spool space. An AMP can use up to 300 megabytes of spool space as long as total spool space used does not exceed the 1 gigabyte global limit for spool space.

```
CREATE DATABASE d1 AS
  PERM = 1e9
  SPOOL = 1e9 SKEW = 20 PERCENT;
```

Example: Creating a Database with a Temporary Space Skew Limit

For a system with 4 AMPs, you create a database with 1 gigabyte of temporary space. The per-AMP quota of temporary space is 250 megabytes. If you set the temporary skew limit to 20%, any AMP is allowed a skew limit of 50MB above the per-AMP quota of temporary space. An AMP can use up to 300 megabytes of temporary space as long as total temporary space used does not exceed the 1 gigabyte global limit for temporary space.

```
CREATE DATABASE d1 AS
  PERM = 1e9,
  TEMPORARY = 1e9 SKEW = 20 PERCENT;
```

Example: Creating a Database with Fallback and Journaling

The following request creates a database named *personnel* from database *administration*:

```
CREATE DATABASE personnel FROM administration
  AS PERMANENT = 5000000 BYTES, FALLBACK, BEFORE JOURNAL,
  DUAL AFTER JOURNAL, DEFAULT JOURNAL TABLE = personnel.fin_copy;
```

The **FALLBACK** keyword specifies that for each table created in the *personnel* database, the default is to store a secondary, duplicate copy in addition to the primary copy.

The JOURNAL option specifies that the default journaling for each data table is to maintain a single copy of the before change image and dual copies of the after change image. A duplicate before change image is maintained automatically for any table in this database that uses both the fallback and the journal defaults.

The DEFAULT JOURNAL TABLE clause is required because journaling is requested. This clause specifies that a new journal table named *fin_copy* is to be created in the new database.

Permanent journaling is activated because you specified the BEFORE JOURNAL and AFTER JOURNAL options. Either option, by itself, is sufficient to activate journaling.

DELETE DATABASE

Deletes all data tables, views, triggers, SQL procedures, macros, and user-installed files (UIFs) from a database.

To delete the definition for the database from the data dictionary, you must use DROP DATABASE. See [DROP DATABASE](#).

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the DROP privilege on the specified database.

DELETE DATABASE Syntax

```
[ DELETE | DEL ] DATABASE database_name [ALL] [;]
```

DELETE DATABASE Syntax Elements

database_name

Name of the database from which all database objects are to be deleted.

ALL

All objects, including the materialized global temporary tables contained by the specified database, are to be dropped.

If you do not specify ALL and the specified database contains materialized global temporary tables, the system returns an error.

Example: Deleting a Database

This statement deletes all tables, views, triggers, SQL procedures, user-defined functions, macros, and user-installed files (UIFs) from the *used_cars* database.

```
DELETE DATABASE used_cars;
```

Related Information

- [CREATE ERROR TABLE](#)
- [CREATE TABLE and CREATE TABLE AS](#)
- [CREATE TABLE \(Queue Table Form\)](#)
- *Teradata Vantage™ - Data Dictionary*, B035-1092

MODIFY DATABASE

Changes the parameters for the specified database.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Other SQL dialects support similar non-ANSI standard statements with names such as ALTER DATABASE.

Required Privileges

You must have the DROP DATABASE privilege on the referenced database.

To modify users, see [MODIFY USER](#).

When you perform a MODIFY DATABASE statement, the system places an exclusive lock on the database that is being modified.

If necessary, the system changes the defined PERM, SPOOL, or TEMPORARY space to the next higher value that is a multiple of the number of AMPs on the system.

You can change one or more options in the same MODIFY statement. A changed option applies to all subsequent operations that occur for the database.

If an existing option is not changed by a MODIFY statement, that option retains its previously defined value.

Privileges Granted Automatically

When you submit a successful MODIFY DATABASE DEFAULT JOURNAL TABLE statement, the following privileges are granted automatically to the database on the journal table:

- DROP TABLE
- INSERT

MODIFY DATABASE Syntax

```
MODIFY DATABASE database_name AS database_attribute [[,]...] [;]
```

database_attribute

```
{ { PERMANENT | PERM | SPOOL | TEMPORARY } = { n | constant_expression
} [BYTES]
  [ SKEW = { constant_expression | DEFAULT } [ PERCENT ] ] |

ACCOUNT = 'account_string' |

DEFAULT MAP = { map_name | NULL } [ OVERRIDE [NOT] ON ERROR ] |

[NO] FALLBACK [PROTECTION] |

[ NO | DUAL ] [BEFORE] JOURNAL |

[ NO | DUAL | [NOT] LOCAL ] AFTER JOURNAL |

DEFAULT JOURNAL TABLE = [ database_name. ] table_name |

DROP DEFAULT JOURNAL TABLE = [ database_name. ] table_name
}
```

MODIFY DATABASE Syntax Elements

database_name

Name of the database to be modified.

AS

An introduction to the list of options for modifying the database.

PERMANENT

A revised value for fixed space allocation for the specified database, in bytes.

n

You can enter the number of bytes as an integer, decimal, or floating point value or as a constant expression whose evaluation determines the number of bytes. You can also enter the value using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

The value of *n* or the number of bytes determined from the evaluation of *constant_expression* cannot exceed the permanent space of the owner.

constant_expression

A constant expression is any SQL expression that does not make any column references. Specifying an appropriate constant expression for the PERM space size of a database enables Vantage to assign an optimal quantity of PERM space that scales to the size of your system by allocating PERM space on a per AMP basis.

When you specify a PERM space size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

Optional keyword that redundantly specifies the unit for the amount of space allowed.

SKEW

Keyword that you use to specify a skew limit for PERMANENT space. You can specify a skew limit percentage that allows the maximum AMP space usage to be above the per-AMP quota, that is, the system maximum space limit divided by the number of AMPs.

constant_expression

Constant expression or constant from 0 through 10000. Specify a value from 1 to 9999 to indicate an AMP-level limit, which is the per-AMP quota * (1+*permskewlimit*/100). Specify 0 to set space accounting to the per-AMP level, that is, no skew. A value of 10000 indicates unlimited skew, up to the system maximum space limit.

DEFAULT

Use the value of DBS Control DefaultPermSkewLimitPercent.

PERCENT

Optional keyword that you can include for readability to indicate that the *constant_expression* or DEFAULT keyword specifies a percentage of allowable skew.

TEMPORARY

Disk usage for a materialized global temporary table is charged to the temporary space allocation of the user who referenced the table. Temporary space is reserved prior to spool space for any user defined with this characteristic.

If default temporary space is not defined for a database, then the space allocated for any materialized global temporary tables created in that database is set to the maximum temporary space allocated for its immediate owner.

n

Number of bytes to be allowed by default for creating global temporary tables by users within this database.

constant_expression

Any SQL expression that does not make any column references. Specifying an appropriate constant expression for the TEMPORARY space size of a user enables Vantage to assign an optimal quantity of TEMPORARY space that scales to the size of your system by allocating TEMPORARY space on a per AMP basis.

When you specify a TEMPORARY space size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

Optional keyword that redundantly specifies the unit for the amount of space allowed.

SKEW

Keyword that you use to specify a skew limit for TEMPORARY space. You can specify a skew limit percentage that allows the maximum AMP space usage to be above the per-AMP quota, that is, the system maximum space limit divided by the number of AMPs.

constant_expression

Constant expression or constant from 0 through 10000. Specify a value from 1 to 9999 to indicate an AMP-level limit, which is the per-AMP quota * (1+*tempskewlimit*/100). Specify 0 to set space accounting to the per-AMP level, that is, no skew. A value of 10000 indicates unlimited skew, up to the system maximum space limit.

DEFAULT

Use the value of DBS Control DefaultTempSkewLimitPercent.

PERCENT

Optional keyword that you can include for readability to indicate that the *constant_expression* or DEFAULT keyword specifies a percentage of allowable skew.

SPOOL

The maximum number of bytes allowed for spool files in *database_name*.

n

You can enter the number of bytes as an integer, decimal, or floating point value or as a constant expression whose evaluation determines the number of bytes. You can also enter the value using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

n cannot exceed the spool space parameter in the profile of the creator. If no spool space is defined for that profile, then Vantage uses the spool space limit defined for the individual user-creator.

constant_expression

A constant expression is any SQL expression that does not make any column references. Specifying an appropriate constant expression for the SPOOL size of a database enables Vantage to assign an optimal quantity of SPOOL that scales to the size of your system by allocating SPOOL space on a per AMP basis.

When you specify a SPOOL size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

Optional keyword that redundantly specifies the unit for the amount of space allowed.

SKEW

Keyword that you use to specify a skew limit for SPOOL space. You can specify a skew limit percentage that allows the maximum AMP space usage to be above the per-AMP quota, that is, the system maximum space limit divided by the number of AMPs.

constant_expression

Constant expression or constant from 0 through 10000. Specify a value from 1 to 9999 to indicate an AMP-level limit, which is the per-AMP quota * (1+*spoolskewlimit*/100). Specify 0 to set space accounting to the per-AMP level, that is, no skew. A value of 10000 indicates unlimited skew, up to the system maximum space limit.

DEFAULT

Use the value of DBS Control DefaultSpoolSkewLimitPercent.

PERCENT

Optional keyword that you can include for readability to indicate that the *constant_expression* or DEFAULT keyword specifies a percentage of allowable skew.

ACCOUNT

The account to be charged for the space used by this database. If not specified, the account defaults to the default account of the immediate owner of the database.

account_string

Each *account_string* must follow the standard Vantage object naming rules. See *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Note:

When specifying an account string for a database, consider that a user session may default to the *account_string* specified for the database that owns the user if no other user or profile based account applies.

DEFAULT MAP

You can specify an existing contiguous or sparse map as the default map for the database. You can also remove a default map that is defined for the database by setting the default map to null.

You must have been granted the specified map unless the map is the same as the default map of the database modifier.

map_name

Name of an existing contiguous or sparse map.

You cannot specify TD_DataDictionaryMap or TD_GlobalMap.

To specify a sparse map as the default map, you must be in the same secure zone as the sparse map and the sparse map must be in the same secure zone as the database.

OVERRIDE ON ERROR

Use the default map if an error occurs when the MAP clause is specified for a CREATE TABLE, CREATE JOIN INDEX, or CREATE HASH INDEX statement. For example, if the MAP clause specifies a nonexistent contiguous map or a sparse map in another secure

zone, a warning message displays. The message indicates that the specified map was not used and lists the name of the default map used instead.

This is the default.

OVERRIDE NOT ON ERROR

Do not use the default map if an error occurs when the MAP clause is specified for a CREATE TABLE, CREATE JOIN INDEX, or CREATE HASH INDEX statement.

NULL

A default map is not associated with the database. You can specify NULL to remove the default map that is currently set for the database.

FALLBACK

A new default for duplicate copy protection of each data table subsequently created in the database. The current fallback setting for existing data tables remains unchanged.

PROTECTION

Optional keyword that you can specify with FALLBACK. The FALLBACK keyword used alone implies PROTECTION.

NO

Sets the default to not provide duplicate copy protection for data tables created in the database.

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

JOURNAL

A new default for the number of before change images to be maintained for data tables subsequently created in the database.

This option can appear twice in the same request: once to specify a BEFORE or AFTER image, and again to specify the alternate type. If only one type is specified, then the current default is modified only for that type.

Journal options are not supported for tables with row sizes greater than 64KB.

NO

Terminates any current journaling default.

DUAL

If the JOURNAL keyword is specified without NO or DUAL, single copy journaling is implied. If journaling is specified, a DUAL journal is maintained for data tables with FALLBACK protection.

Existing images are not affected until the corresponding table is updated.

BEFORE

The JOURNAL keyword without BEFORE or AFTER indicates that both types of images are to be maintained. In this case, the current default for either or both types is modified accordingly. For example, if only AFTER JOURNAL is specified, the current default for before change images remains in effect.

If BEFORE and AFTER are specified, the two must not conflict.

AFTER JOURNAL

The type of image to be maintained for the table. Any existing images are not affected until the table is updated.

NO

After-change images are not maintained for the table.

DUAL

Dual after-change images are maintained for the table.

LOCAL

Single after-image journal rows for non-fallback data tables are written on the same virtual AMP as the changed data rows.

NOT LOCAL

Single after-image journal rows for non-fallback data tables are written on another virtual AMP in the cluster.

DEFAULT JOURNAL TABLE

A redefinition for the current journal table.

Specifying this option does not change the status of existing data tables in the modified database.

journal_table_name

Name of the default journal table. The *journal_table_name* parameter is required if this clause is specified without the DROP keyword.

database_name

If a database name is not specified, then the database being modified is assumed.

If the database being modified does not have a journal table, the system creates *journal_table_name* by default.

If a different database is specified, then it must already exist and *journal_able_name* must have been defined as its default journal table.

DROP DEFAULT JOURNAL TABLE

The DROP keyword removes the default status of the journal table currently defined as the default for the database being modified. If the journal table resides in the database being modified, DROP also deletes the table from the system.

An error message is returned if the DROP request would delete a journal table that is being used by active data tables.

Usage Notes

Journal Tables and the Default Map

Modifying the default map of the database does not change the map being used for the journal table.

If the MODIFY DATABASE statement specifies a journal table and a default map, the journal table is created using this default map. The journal table map must be a contiguous map with the same AMPs as TD_GlobalMap.

If the MODIFY DATABASE statement specifies a journal table without specifying a default map, the system determines the map for the journal table in the following order of precedence:

- Default map, if defined, for the profile of the database creator. See the [DEFAULT MAP](#) option for CREATE PROFILE.
- Default map, if defined, for the database creator. See the [DEFAULT MAP](#) option for CREATE USER.
- System-default map.

MODIFY DATABASE Examples

Example: Change Permanent Space Allocation

Change the permanent space allocation for the personnel database to 6,000,000 bytes.

```
MODIFY DATABASE personnel AS
PERMANENT = 6000000 BYTES;
```

Example: Use Constant Expression to Modify PERM, SPOOL, and TEMPORARY Database Spaces

The following statement modifies the PERM space for database production_development to a size based on the constant expression $3,000,000 * (\text{HASHAMP}()+1)$, SPOOL to a size based on the constant expression $3,000,000 * (\text{HASHAMP}()+1)$, and TEMPORARY to a size based on the constant expression $3,000,000 * (\text{HASHAMP}()+1)$. The expressions calculate the number of AMPs in the current system and scale the PERM, SPOOL, and TEMPORARY space for the production_development database to that size.

This is the original definition of database production_development.

```
CREATE DATABASE production_development AS
DEFAULT DATABASE = it_dev,
PASSWORD = (EXPIRE = 0),
PERM = 2000000*(HASHAMP()+1),
SPOOL = 2000000*(HASHAMP()+1),
TEMPORARY = 2000000*(HASHAMP()+1);
```

The following MODIFY DATABASE request changes the PERM, SPOOL, and TEMPORARY space allocations for this database as follows.

```
MODIFY DATABASE production_development AS
DEFAULT DATABASE = it_dev,
PASSWORD = (EXPIRE = 0),
PERM = 3000000*(HASHAMP()+1),
SPOOL = 3000000*(HASHAMP()+1),
TEMPORARY = 3000000*(HASHAMP()+1);
```

Example: Add Permanent Space Skew Limit to Database

Assume a system with 4 AMPs and a database with 1 gigabyte of permanent space. The per-AMP quota of permanent space is 250 megabytes. If you set the permanent space skew limit to 10%, any AMP is allowed a skew limit of 25 megabytes over the per-AMP quota of permanent space. An AMP can use up to 275 megabytes of permanent space as long as total permanent space used does not exceed the 1 gigabyte global limit for permanent space.

```
MODIFY DATABASE d1 AS
  PERM = 1e9 SKEW = 10 PERCENT;
```

Example: Add Temporary Space Skew Limit to Database

Assume a system with 4 AMPs and a database with 1 gigabyte of temporary space. The per-AMP quota of temporary space is 250 megabytes. If you set the temporary skew limit to 20%, any AMP is allowed a skew limit of 50MB above the per-AMP quota of temporary space. An AMP can use up to 300 megabytes of temporary space as long as total temporary space used does not exceed the 1 gigabyte global limit for temporary space.

```
MODIFY DATABASE d1 AS
  PERM = 1e9,
  TEMPORARY = 1e9 SKEW = 20 PERCENT;
```

Example: Add Spool Space Skew Limit to Database

Assume a system with 4 AMPs and a database with 1 gigabyte of spool space. The per-AMP quota of spool space is 250 megabytes. If you set the spool space skew limit to 20%, any AMP is allowed a skew limit of 50 megabytes over the per-AMP quota of spool space. An AMP can use up to 300 megabytes of spool space as long as total spool space used does not exceed the 1 gigabyte global limit for spool space.

```
MODIFY DATABASE d1 AS
  PERM = 1e9
  SPOOL = 1e9 SKEW = 20 PERCENT;
```

Example: Change Fallback Protection and Space Allocation

Change fallback protection and space allocation options on the finance database as follows.

```
MODIFY DATABASE finance AS
  PERM = 75000000,
  SPOOL = 150000000,
```

```
NO FALLBACK,  
DROP DEFAULT JOURNAL TABLE = finance.journals;
```

Example: Change Default Journal Table

If the current journal table is not contained within the database being modified, then you could use the following request to change the default journal table from *FinCopy* to *Jrn11*.

```
MODIFY DATABASE Personnel AS  
DEFAULT JOURNAL TABLE = Jrn11;
```

Example: Drop Default Journal Table

Two MODIFY requests are needed to change the default journal table if it is contained within the database being modified.

The journal table *FinCopy* is contained within the personnel database. To change the default journal table from *FinCopy* to *Jrn11*, the present default journal table must be dropped by entering the following request.

```
MODIFY DATABASE Personnel AS  
DROP DEFAULT JOURNAL TABLE;
```

This request removes *FinCopy* as the default and also drops it from the system.

If any existing tables use *FinCopy* as their journal table, the request returns an error message.

Example: Define New Default Journal Table

After default journal has been dropped, use this request to define a new default journal table.

```
MODIFY DATABASE Personnel AS  
DEFAULT JOURNAL TABLE = Jrn11;
```

Related Information

For more information about database definitions, see [CREATE DATABASE](#).

DATABASE

Establishes a new default database for the current session for SQL requests that do not have fully-qualified table, view, or macro names.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

None.

DATABASE Syntax

```
DATABASE database_name [;]
```

DATABASE Syntax Elements

database_name

Name of the new default database, which supersedes the default database specified in the user or profile definition for the current session.

Example: Defining Default Database for Current Session

The following request establishes *personnel* as the default database for the current session.

```
DATABASE personnel;
```

Related Information

- DATABASE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- CREATE USER in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- MODIFY USER in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184

DROP DATABASE

Drops the definition for an empty database from the Data Dictionary.

You must drop all the objects contained by that database before you can drop the database itself.

The drop operation verifies that the database is empty, verifies that the database does not own any other databases or users, drops the database, and adds the PERM and TEMP space that the drop makes available to that of the immediate owner database or user.

After a database is dropped, you cannot recover it by using the Dump and Restore utility unless it is restored.

To delete objects from a database, use the DELETE DATABASE statement. See [DELETE DATABASE](#). To delete objects from a user, use the DELETE USER statement. See [DELETE USER](#).

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

DROP DATABASE is functionally similar to the ANSI SQL:2011 statement DROP SCHEMA.

Required Privileges

You must have the DROP DATABASE privilege on the database to be dropped.

DROP DATABASE Syntax

```
DROP DATABASE database_name [;]
```

DROP DATABASE Syntax Elements

database_name

Name of the database that is to be dropped.

The database that is being dropped cannot own other databases or users, and must be at the bottom of the database hierarchy.

Usage Notes

Dropping a Populated Database

You cannot drop a populated database. To ensure that a database is empty:

- Drop any hash or join indexes contained in any other databases or users that reference a table in the current database. See [DROP INDEX](#), [DROP HASH INDEX](#), and [DROP JOIN INDEX](#).
- Drop any triggers contained in any other databases or users that reference a table in the current database. See [DROP MACRO](#).
- Drop any journal tables in the current database. See [MODIFY DATABASE](#).
- Delete the database or drop all the objects within the database. See [DELETE DATABASE](#).

For more information, see the appropriate statement or procedure call:

- [DROP FUNCTION](#)
- [DROP PROFILE](#)
- [DROP ROLE](#)
- SQLJ.Remove_Jar See *Teradata Vantage™ - SQL External Routine Programming*, B035-1147

Example: Dropping Empty Database

This request drops the empty database named *accounting*.

```
DROP DATABASE accounting;
```

Related Information

- DROP DATABASE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- CREATE DATABASE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- DELETE DATABASE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184

HELP DATABASE

Displays the attributes, sorted by object name, for all tables, views, join indexes, hash indexes, SQL procedures, user-defined functions, and macros contained by a specified database.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must either own the database or have at least one privilege on that database.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against a specified database.

HELP DATABASE Syntax

```
HELP DATABASE database_name [;]
```

HELP DATABASE Syntax Elements

database_name

Name of the database for which information is requested.

HELP DATABASE Examples

Example: HELP DATABASE

The following request returns the following information about the personnel database.

```
HELP DATABASE personnel;
Table/View/Macro Name Kind Comment Protection Creator Name
-----
Department          T          F Administration
.                    .
.
```

Example: HELP DATABASE Output Showing a UDT and a Method

The following example shows that there is a UDT named *euro* and a method named *euro_To_US* in the SYSUDTLIB database.

```
HELP DATABASE SYSUDTLIB;
Table/View/Macro name euro
Kind U
Comment ?
Protection F
Creator Name USER1
Commit Option N
Transaction Log Y
Table/View/Macro name euro_To_US
Kind H
Comment ?
Protection F
Creator Name USER1
Commit Option N
Transaction Log Y
```

Example: Output with a UIF

For installed files, the HELP DATABASE Kind column displays Z.

```
HELP DATABASE db1;
Table/View/Macro name Kind Comment
```

```

-----
          ft1      L      ?
    ft1_contract  C      ?
          map_fn   Z      ?

```

Related Information

- [SHOW object](#)
- HELP DATABASE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100

LOGGING INCREMENTAL ARCHIVE ON FOR *object_list*

Enables incremental restore for databases and tables you specify.

A write lock is placed on the databases and tables you specify while this statement executes.

Privileges Required

You must have the DUMP privilege on the specified tables or on the database containing the tables.

LOGGING INCREMENTAL ARCHIVE ON FOR *object_list* Syntax

```
LOGGING INCREMENTAL ARCHIVE ON FOR object_list [,...] [, DELETE LOG ROWS ] [;]
```

object_list

```
{ database_name[.table_name] | user_name[.table_name] | table_name
} [,...]
```

LOGGING INCREMENTAL ARCHIVE ON FOR *object_list* Syntax Elements

database_name

Name of database.

user_name

Name of user.

table_name

Name of table.

DELETE LOG ROWS

Remove rows from the deleted rows logging subtable.

Examples

Examples: Enable Incremental Restore

This example enables incremental restore on the database MyDB and sets the option to delete log rows.

```
LOGGING INCREMENTAL ARCHIVE ON FOR MyDB DELETE LOG ROWS;
```

This example enables incremental restore on the table tab1 in the current database and sets the option to delete log rows.

```
LOGGING INCREMENTAL ARCHIVE ON FOR tab1 DELETE LOG ROWS;
```

LOGGING INCREMENTAL ARCHIVE OFF FOR *object_list*

Disables incremental restore for databases and tables you specify.

A write lock is placed on the databases or tables while the statement executes.

Required Privileges

You must have the DUMP privilege on the specified tables or on the database containing the tables.

LOGGING INCREMENTAL ARCHIVE OFF FOR *object_list* Syntax

```
LOGGING INCREMENTAL ARCHIVE OFF FOR object_list [;]
```

object_list

```
{ database_name[.table_name] | user_name[.table_name] | table_name  
  } [,...]
```

LOGGING INCREMENTAL ARCHIVE OFF FOR *object_list*

Syntax Elements

database_name

Name of database.

user_name

Name of user.

table_name

Name of table.

Examples

Examples: Disabling Incremental Restore

This example disables incremental restore on the database MyDB.

```
LOGGING INCREMENTAL ARCHIVE OFF FOR MyDB;
```

This example disables incremental restore on the table tab1 in the current database.

```
LOGGING INCREMENTAL ARCHIVE OFF FOR tab1;
```

INCREMENTAL RESTORE ALLOW WRITE FOR *object_list*

Enables read and write access for the databases, users, or tables you specify after an incremental restore. Incremental restore sets tables to read-only access.

An exclusive lock is placed on the tables you specify while this statement executes.

A backup or restore job cannot be in progress when you perform this statement.

Required Privileges

You must have the RESTORE privilege on the specified tables or on the database containing the tables.

INCREMENTAL RESTORE ALLOW WRITE FOR *object_list* Syntax

```
INCREMENTAL RESTORE ALLOW WRITE FOR object_list [;]
```

object_list

```
{ database_name[.table_name] | user_name[.table_name] | table_name
} [, ...]
```

INCREMENTAL RESTORE ALLOW WRITE FOR *object_list*

Syntax Elements

database_name

Name of database.

user_name

Name of user.

table_name

Name of table.

Examples

Examples: Enable Read and Write Access After an Incremental Restore

This example enables read and write access on the database MyDB after an incremental restore.

```
INCREMENTAL RESTORE ALLOW WRITE FOR MyDB;
```

This example enables read and write access on the table tab1 in the current database after an incremental restore.

```
INCREMENTAL RESTORE ALLOW WRITE FOR tab1;
```

User, Profile, and Role Statements

CREATE PROFILE

Creates a profile that defines a set of parameters. You can then assign the profile to multiple users.

Because profiles exist in system space rather than in user or database space, profiles are not part of the database ownership hierarchy.

Note:

The value for a profile attribute supersedes any corresponding values specified in global defaults or in the user definitions of profile member users.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

To use CREATE PROFILE, you must have the CREATE PROFILE privilege.

To assign a row-level security constraint to a profile, you must have the CONSTRAINT ASSIGNMENT privilege.

New users do not implicitly have the CREATE PROFILE privilege. User DBC or a user who has the CREATE PROFILE privilege WITH GRANT OPTION can grant this privilege to another user.

Privileges Granted Automatically

None.

CREATE PROFILE Syntax

```
CREATE PROFILE profile_name [ AS profile ] [ ; ]
```

profile

```
{ ACCOUNT = { 'account_string' | ('account_string' [, ...] ) | NULL } |  
  
  DEFAULT MAP = { map_name | NULL } [ OVERRIDE [NOT] ON ERROR ] |  
  
  DEFAULT DATABASE = { database_name | NULL } |
```



```

SPOOL = { { n | constant_expression } [BYTES] | NULL } |

TEMPORARY = { { n | constant_expression } [BYTES] | NULL } |

PASSWORD [ATTRIBUTES] = { (attribute [,...]) | NULL } |

QUERY_BAND = 'pair [...]' [ ([NOT] DEFAULT) ] |

IGNORE QUERY_BAND VALUES = 'pair [...]' |

TRANSFORM ( transformation [,...] ) |

COST PROFILE = { cost_profile_name | NULL } |

CONSTRAINT = constraint_specification [,...]
} [,...]

```

attribute

```
attribute_name = { value | NULL }
```

pair

```
pair_name = pair_value;
```

transformation

```
data_type = group_name
```

constraint_specification

```

rls_constraint_column_name
  ( { level_specification [,...] | category_name [,...] | NULL } )

```

level_specification

```
level_name [DEFAULT]
```

CREATE PROFILE Syntax Elements

profile_name

Name of the profile to be created.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

A profile name must be unique among profiles on a Vantage system.

ACCOUNT

One or more accounts to be assigned to the profile.

Accounts do not exist independently in the database. Accounts only exist as specified in user, database, and profile definitions. You can specify the same account string in multiple object definitions, if needed.

Accounts defined in the profile supersede the accounts defined in the user definitions of profile member users.

Users with multiple accounts can access a non-default account by specifying the account with one of the following:

- Logon string. See *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.
- SET SESSION ACCOUNT statement. See [SET SESSION ACCOUNT](#).

For a user to specify an account in the logon string or in a SET SESSION ACCOUNT statement, the account must be assigned in the user or profile definition.

account_string

Each position in an account string has a specific meaning and format requirement. See *Teradata Vantage™ - Database Administration*, B035-1093. Each *account_string* must follow the Vantage object naming rules. See *Teradata Vantage™ - SQL Fundamentals*, B035-1141. You must enclose each account string with APOSTROPHE characters. You must separate each entry in a list of account strings with a COMMA character and enclose the list with LEFT PARENTHESIS and RIGHT PARENTHESIS characters.

The first account string you specify becomes the default account for profile members, but you can specify multiple accounts if needed.

If you do not specify an account string for a profile, the system defaults to the accounts specified for individual users. If a user has no account in either the user or profile definitions, the user inherits the first account string that is defined for the immediate owner of the user.

DEFAULT MAP

You can specify an existing contiguous or sparse map as the default map for the profile.

You must have been granted the specified map unless the map is the same as the default map of the profile creator.

map_name

Name of an existing contiguous or sparse map.

You cannot specify TD_DataDictionaryMap or TD_GlobalMap.

To specify a sparse map as the default map, you must be in the same secure zone as the sparse map and the sparse map must be in the same secure zone as the profile.

OVERRIDE ON ERROR

Use the default map if an error occurs when the MAP clause is specified for a CREATE TABLE, CREATE JOIN INDEX, or CREATE HASH INDEX statement. For example, if the MAP clause specifies a nonexistent contiguous map or a sparse map in another secure zone, a warning message displays. The message indicates that the specified map was not used and lists the name of the default map used instead.

This is the default.

OVERRIDE NOT ON ERROR

Do not use the default map if an error occurs when the MAP clause is specified for a CREATE TABLE, CREATE JOIN INDEX, or CREATE HASH INDEX statement.

NULL

A default map is not associated with the database.

DEFAULT DATABASE = *database_name*

Name of the default database established each time a user with this profile logs onto the Vantage.

database_name should be an existing database. The system returns an error when a user with this profile tries to create or reference an object within a nonexistent database.

If the default database is NULL or is not defined in the profile assigned to a user, Vantage uses the setting defined for the individual user. If a default database is not specified for the profile or user, the system uses the username as the default database.

A user can also use SET SESSION DATABASE to specify an alternate default database.

SPOOL

A keyword allowing you to specify the number of bytes allowed for spool files for *profile_name*. The default is null, which causes the system to use the setting defined for the individual user.

If no spool space is defined in the profile assigned to a user, Vantage uses the setting defined for the individual user. If spool space is not defined for either the user profile or the user, the system uses the spool space assigned to the owner of the space in which the user was created.

Profiles can be assigned to a proxy user in a trusted session so that the profile attributes of the permanent proxy user, such as perm space and spool, are used. If a permanent proxy user logs on directly and runs queries while also logged on through a middle tier application trusted session with queries being run as a proxy, the SPOOL and TEMP usage accumulation for that user includes both. The individual user's query fails when the collective usage exceeds the individual user's limits.

n

n cannot exceed the spool space parameter in the profile of the creator. If no spool space is defined for that profile, then the spool space limit defined for the user setting of the creator is used.

n and the evaluation of *constant_expression* both refer to bytes, whether or not the optional BYTES keyword is specified.

You can enter the number of bytes as an integer, decimal, or floating point value or as a constant expression whose evaluation determines the number of bytes. You can also enter the value using exponential notation. For example, you can write *one thousand* as either 1000 or 1E3.

BYTES

BYTES is an optional keyword that redundantly specifies the unit for the amount of SPOOL space allowed.

constant_expression

A constant expression is any SQL expression that does not make any column references. Specifying an appropriate constant expression for the SPOOL size of a profile enables Vantage to assign an optimal quantity of SPOOL that scales to the size of your system.

When you specify a SPOOL size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

TEMPORARY

A keyword allowing you to define how much space to allow for creating materialized global temporary tables.

n

n cannot exceed the temporary space parameter in the profile of the creator.

n can be an integer, a decimal value, or a floating point value.

n refers to bytes, whether or not the optional BYTES keyword is specified.

If no temporary space limit is defined for the profile, the system uses the temporary space limit defined for the individual user. If temporary space is not defined for either the profile or user, the system uses the temporary space limit for the owner of the space in which the user was created.

If both space allocations are defined, temporary space is reserved prior to spool space.

Disk usage for a materialized global temporary table is charged to the temporary space allocation of the user who referenced the table.

constant_expression

Any SQL expression that does not make any column references. Specifying an appropriate constant expression for the TEMPORARY space size of a user enables Vantage to assign an optimal quantity of TEMPORARY space that scales to the size of your system by allocating TEMPORARY space on a per AMP basis.

When you specify a TEMPORARY space size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

BYTES is an optional keyword that redundantly specifies the unit for the amount of TEMPORARY space

NULL

The default is NULL, which causes Vantage to use the setting defined for the user.

PASSWORD

Password controls only affect users authenticated by the database. Externally authenticated users are unaffected.

Some password control attribute values are not applicable to profile members, but only to children of profile members. See [Effects of Profile-Based Password Controls](#).

For a detailed description of password control methods, options, and strategies, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

Note:

Character-related password format controls do not apply to multibyte client character sets on systems enabled with Japanese language support.

ATTRIBUTES

Keywords that introduces a set of password control attributes.

If you specify the PASSWORD or PASSWORD ATTRIBUTES phrase, you must list all 9 of the password control options and specify either a value or NULL (the default) for each. Password security attributes defined in a profile take effect the next time a profile member user logs on.

NULL

If the value for an attribute is NULL or if you do not specify the PASSWORD or PASSWORD ATTRIBUTES phrase, the profile defaults to the global password control settings in DBC.SysSecDefaults. If you specify a value in a profile for a password control, the value supersedes the global value for profile members.

attribute_name = value

Following are brief descriptions of the password controls.

EXPIRE=*n*

Number of days to elapse before the password expires.

A value of 0 for *n* indicates the password never expires.

NULL indicates that the EXPIRE option is not set for the profile.

You can specify any non-zero value to cause new users who are profile members to replace the temporary password specified in their user definitions with a permanent private password at first log on. Users must use the MODIFY USER statement to change their password.

MINCHAR=*n*

Minimum number of characters in a password string.

The valid range for *n* is 1-127 UNICODE characters.

NULL indicates that the MINCHAR option is not set for the profile.

MAXCHAR=*n*

Maximum number of characters in a password string.

The valid range for *n* is 1-127 UNICODE characters.

NULL indicates that the MAXCHAR option is not set for the profile.

DIGITS=*c*

Specifies whether at least one digit must appear in a password string. The table below lists the values you can specify for *c*.

<i>c</i>	Description
N or n	Digits are not permitted in a password string.
R or r	At least one digit is required in a password string.
Y or y	Digits are permitted in a password string, but not required.
NULL	Indicates that the DIGITS option is not set for the profile.

SPECCHAR=*c*

Specifies whether various characters or the user name are allowed, not allowed, or required in a user password string.

The value for SPECCHAR must be one of the single letter option codes shown in [Working with the SPECCHAR Password Control](#), where the letter code you specify for the SPECCHAR option represents a unique set of possible SPECCHAR rules.

MAXLOGONATTEMPTS=*n*

Number of incorrect logon attempts allowed before locking the user from further attempts, where *n* is a value from 0 to 127.

A value of 0 for *n* indicates never to lock the user.

NULL indicates that the MAXLOGONATTEMPTS option is not set for the profile.

LOCKEDUSEREXPIRE=*n*

Number of minutes to elapse before unlocking a locked user.

- If *n* is 0, Vantage unlocks the user immediately.
- If *n* is -1, Vantage locks the user indefinitely.
- NULL indicates that the LOCKEDUSEREXPIRE option is not set for the profile.

REUSE=*n*

Number of days to elapse before a password can be reused.

A value of 0 for *n* allows the password to be reused immediately.

NULL indicates that the REUSE option is not set for the profile.

RESTRICTWORDS=c

Specifies whether certain words are restricted from use within a password string.

The valid values for *c* are listed below.

- If *c* is Y or y, any words listed in *DBC.PasswordRestrictions* cannot be used in password strings.
- If *c* is N or n, use of words listed in *DBC.PasswordRestrictions* in password strings is allowed.
- NULL indicates that the RESTRICTWORDS option is not set for the profile.

For details, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100 and *Teradata Vantage™ - Database Administration*, B035-1093.

QUERY_BAND

Add a query band to the profile. Defines the set of name-value pairs to set at logon of all users assigned the profile.

You must enclose the set of *name=value* pairs for the query band with APOSTROPHE characters (').

The number of *name=value* pairs is limited to the maximum length of the string, which is 4,096 UNICODE characters, including pad characters.

Pair names and values cannot contain any of the following characters:

- SEMICOLON (;)
- a null

If an APOSTROPHE character is embedded within a pair name or value, you must type it twice, to escape it. Otherwise, the system interprets it as a *pair_name=pair_value* string terminator.

Note:

Do not specify reserved pair names or values. For a list of reserved query band names and values, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

QUERY_BAND

Keyword to introduce query band options you specify.

pair_name

Name component of a query band specification.

The maximum size for each *pair_name* is 128 UNICODE characters. For more information about database object names, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

pair_value

Value component of a query band specification.

pair_value can be up to 256 UNICODE characters in length.

DEFAULT

All of the values are considered default values that can be overwritten using the SET QUERY_BAND statement for the session or transaction. If the transaction, session, and profile query bands have name-value pairs with the same name, the name-value pair used for Teradata Active Systems Management (TASM) classification is the first one found in a query band searched in the following order:

- Transaction query band
- Session query band
- Profile query band

See [SET QUERY_BAND](#).

NOT DEFAULT

Overwriting the pairs in the profile query band is not permitted. Any pair in a SET QUERY_BAND statement matching a name in the profile query band is discarded from the query band. If the DEFAULT clause is omitted, the profile query band is set to NOT DEFAULT.

IGNORE QUERY_BAND VALUES

Defines the set of name-value pairs to ignore if specified in a SET QUERY_BAND statement for all users assigned the profile. When a user with the profile issues a SET QUERY_BAND statement containing a name-value pair matching a IGNORE QUERY_BAND VALUES name-value pair, the system performs the following actions:

- The matching name-value pair is ignored.
- The query band is set with the remaining name-value pairs.
- A warning message displays.

You must enclose the set of *name=value* pairs for the query band with APOSTROPHE characters (').

The number of *name=value* pairs is limited to the maximum length of the string, which is 4,096 UNICODE characters, including pad characters.

Pair names and values cannot contain any of the following characters:

- SEMICOLON (;)
- a null

Setting a *pair_name* to an empty string value indicates that a name-value pair with the *pair_name* and any value will be discarded. For example:

```
pair_name=;
```

To specify multiple values for the same *pair_name*, each name-value pair must be specified separately. For example:

```
IGNORE QUERY_BAND VALUES =  
    'TVSTemperature=HOT;TVSTemperature=WARM;'
```

If an APOSTROPHE character is embedded within a pair name or value, you must type it twice, to escape it. Otherwise, the system interprets it as a *pair_name=pair_value* string terminator.

If a profile has a query band that is defined as NOT DEFAULT, it is redundant to define the IGNORE QUERY_BAND VALUES with name-value pairs having the same names as defined in the profile query band. The NOT DEFAULT profile query band causes the SET QUERY_BAND statement to discard any name-value pair matching the names in the profile query band.

IGNORE QUERY_BAND VALUES

Keywords to introduce query band options to ignore.

pair_name

Name component of a query band specification.

The maximum size for each *pair_name* is 128 UNICODE characters. For more information about database object names, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

pair_value

Value component of a query band specification.

pair_value can be up to 256 UNICODE characters in length.

TRANSFORM

Adds UDT transform groups to the profile.

You can specify up to 256 *data_type-transform_group* pairs for a profile.

If the UDT has a default transform group, the default transform group is used unless the user specifies a transform group.

data_type

Name of user-defined data type (UDT).

group_name

Name of transform group for the data type.

COST PROFILE = *cost_profile_name*

Optimizer cost profile to be associated with *profile_name*.

The default is NULL, meaning use the default system cost profile.

Note:

Optimizer cost profiles are not intended for use on production systems. The COST PROFILE parameter is for use only under the direction of Teradata Support Center personnel.

CONSTRAINT***rls_constraint_name***

Name of one or more row-level security constraints, each followed by a list of the hierarchical levels or non-hierarchical categories, valid for the constraint, which are being assigned to the *profile_name*.

For more information about row-level security constraints, see [CREATE CONSTRAINT](#).

level_name

List of the hierarchical levels. Example level (hierarchical) constraint assignment:

```
CONSTRAINT = Classification_Level (Secret, Unclassified DEFAULT)
```

category_name

List of non-hierarchical categories. Example category (non-hierarchical) constraint assignment:

```
CONSTRAINT = Classification_Country (US, UK, GER)
```

There is a limit of 6 hierarchical constraints and 2 non-hierarchical constraints that can be assigned per profile.

DEFAULT

DEFAULT can be specified for 1 classification level from the specified list of levels for a hierarchical constraint. The specified level becomes the default value for the constraint when a session is established for the user to which the profile is assigned.

If you do not specify DEFAULT for any of the listed classification levels, then the first level listed becomes the default for the profile.

DEFAULT is not valid for non-hierarchical constraints.

Usage Notes

Effects of Profile-Based Password Controls

The effects of password controls specified in a profile definition vary from those controls applied directly to a user in a CREATE USER or MODIFY USER statement.

Password Control Attribute	Effects
EXPIRE	Applies to first level user to which the profile is assigned, and is effective immediately
MINCHAR and MAXCHAR	<p>Does not apply to first level user to which the profile is assigned, rather it applies only to the children of that user.</p> <p>As a result of these second-level effects, if the user submitting the CREATE USER statement is a member of a profile that specifies values for password control attributes, the password the creator assigns in the CREATE USER statement must comply with the password controls in the creator profile.</p> <p>Controls take effect at the first child user logon after the child is created or modified.</p> <p>To apply password controls to first-level users not affected by profile-based controls, you can Allow the users to defer to the global controls in DBC.SysSecDefaults.</p>
DIGITS	
REUSE	
SPECCHAR	
RESTRICTWORDS	
MAXLOGONATTEMPTS	Applies to first level users to which the profile is assigned, at the next logon attempt.
LOCKEDUSEREXPIRE	

Working with the SPECCHAR Password Control

The Password SpecChar control defines the use of special characters in user passwords.

Vantage has a simplified set up for defining special character options, which uses a single character to represent each valid combination of the following four options:

- Special characters are allowed in a password.
- The username for the user is allowed in the password string.
- Alpha characters are allowed but not required.
- Mixed upper and lower case characters are allowed but not required.

The values are not case sensitive.

When you specify a single letter SPECCHAR setting, the system sets the status of each of the four options using the status indicators shown in the following table. Each status indicator defines how the system enforces a password parameter that has that status.

Status Indicator	Description
Y	Allowed but not required
N	Not Allowed
R	Required

Specifying the Single-Letter Code to Define SpecChar Rules

Specify the SPECCHAR single-letter option code with the set of status indicators that represents how you want to enforce the four SPECCHAR options.

SPECCHAR Option Code	Included Special Character Option Rules			
	UserName	Mixture of Upper and Lower Case letters	At least One Alpha Character	Special Characters
N, n	Y	Y	Y	N
Y, y	Y	Y	Y	Y
A, a	Y	Y	Y	R
B, b	Y	Y	R	N
C, c	Y	Y	R	Y
D, d	Y	Y	R	R
E, e	Y	R	R	N
F, f	Y	R	R	Y
G, g	Y	R	R	R
H, h	N	Y	Y	N
I, i	N	Y	Y	Y
J, j	N	Y	Y	R
K, k	N	Y	R	N
L, l	N	Y	R	Y
M, m	N	Y	R	R

Included Special Character Option Rules				
SPECCHAR Option Code	UserName	Mixture of Upper and Lower Case letters	At least One Alpha Character	Special Characters
O, o	N	R	R	N
P, p	N	R	R	Y
R, r	N	R	R	R

Default Value

The default value of the SPECCHAR parameter is Y.

Effects of Object Naming on SPECCHAR Controls

The Password SpecChar control includes some additional character effects:

- Upper includes all UNICODE characters in General Category Class Lu.
- Lower includes all UNICODE characters in General Category Class Ll.
- Alpha includes all UNICODE characters that are in either Upper (Lu) or Lower (Ll)
- Special indicates characters that are neither Alpha nor any of the characters 0 through 9 (U+0030 – U+0039). By this definition, uncased letters (General Category Class Lo) are considered Special (for example, Arabic, Chinese or Hebrew characters). Likewise numeric digits (General Category Class Nd) other than the characters 0 through 9 (U+0030 – U+0039) are classified as Special.

For consistency in handling characters, the system checks Password SpecChar based on the NFC (Normalization Form C) representation of the password string. For example, the character 'é' (U+00E9) is a Special, a Lower, and an Alpha character.

For details, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

CREATE PROFILE Examples

Example: Creating Profile that Defines Default Database

The following request creates a profile called *human_resources* that defines the default database to be *personnel*.

```
CREATE PROFILE human_resources AS
  DEFAULT DATABASE = personnel;
```

The following request assigns the *human_resources* profile to a new user.

```
CREATE USER marks AS
  PROFILE = human_resources;
```

Example: Using Constant Expression to Specify SPOOL Space for Profile

The following request creates SPOOL with a size based on the constant expression 2,000,000 (HASHAMP()+1). The expression calculates the number of AMPs in the current system and scales the SPOOL space for the *research_and_development* profile to that size.

```
CREATE PROFILE research_and_development AS
  DEFAULT DATABASE = it_dev,
  PASSWORD = (EXPIRE = 0),
  SPOOL = 2000000*(HASHAMP()+1);
```

Example: Using Constant Expression to Specify TEMPORARY Space for Profile

The following statement creates a profile with the amount of TEMPORARY space based on the constant expression 2,000,000 * (HASHAMP() +1). The expression calculates the number of AMPs in the current system and scales the TEMPORARY space for the *research_and_development* profile to that size.

```
CREATE PROFILE research_and_development AS
  DEFAULT DATABASE = it_dev,
  PASSWORD = (EXPIRE = 0),
  TEMPORARY = 2000000*(HASHAMP()+1);
```

Example: Creating Profile that Defines User-Level Password Security Attribute

The following request defines user-level password security attributes for a profile called *human_resources*.

```
CREATE PROFILE human_resources AS
  DEFAULT DATABASE = personnel,
  PASSWORD =
  (EXPIRE =          90,
   MAXLOGONATTEMPTS = 10,
   LOCKEDUSEREXPIRE = -1);
```

The following request assigns the *human_resources* profile to user *marks*.

```
MODIFY USER marks AS
  PROFILE = human_resources;
```

Example: Set Profile Query Band Pairs as Default

This example sets the query band as the default.

```
CREATE PROFILE testprofile AS
  QUERY_BAND = 'IMPORTANCE=batch;' (DEFAULT);
```

If a user with the profile *testprofile* issues the following statement, it will be accepted and the name-value pair used by Teradata Active System Management is IMPORTANCE=OnlineStrategic.

```
SET QUERY_BAND = 'IMPORTANCE=OnlineStrategic;' FOR SESSION;
```

Example: Profile Query Band Pairs Are Not Default

Because this CREATE PROFILE statement does not include the DEFAULT clause, the profile query band is set to NOT DEFAULT.

```
CREATE PROFILE salesprofile AS,
  SPOOL = 2e6*(HASHAMP()+1),
  QUERY_BAND = 'GROUP=sales;AREA=retail;';
```

If a user with the profile *salesprofile* issues the following statement, the GROUP=production pair is removed from the query band because the profile query band pairs cannot be overwritten.

```
SET QUERY_BAND = 'GROUP=production;' FOR SESSION;
```

Example: Creating Profile to Set Default Query Band Value and Query Band Value to Ignore

In this example, the user's profile query band is set to TVSTemperature=COLD. The user can issue a SET QUERY_BAND statement setting a higher precedence name-value pair of TVSTemperature=WARM. However, if the user sets the query band to TVSTemperature=HOT, the name-value pair is discarded from the query band.


```
CREATE PROFILE salesprofile AS,
  QUERY_BAND = 'TVSTemperature=COLD;' DEFAULT,
  IGNORE QUERY_BAND VALUES ='TVSTemperature=HOT';
```

Example: Creating Profile with System Default Query Band Value and Query Band Value to Ignore

In this example, the profile QUERY_BAND does not contain a TVSTemperature name-value pair. With this profile, suppose the session has the system default TVSTemperature setting as defined in the related DBS Control parameter. The user can execute an application that sets the QUERY_BAND TVSTemperature to WARM or COLD.

```
CREATE PROFILE salesprofile AS,
  QUERY_BAND = 'GROUP=WestCoast;' DEFAULT,
  IGNORE QUERY_BAND='TVSTemperature=HOT';
```

Example: Creating Profile that Ignores *pair-name* with Any *pair-value*

This example specifies the name IMPORTANCE without a value, that is, an empty string. This causes a SET QUERY_BAND statement to discard the name-value with the matching name and any value.

```
CREATE PROFILE testprofile AS
  IGNORE QUERY_BAND VALUES = 'IMPORTANCE=;' ;
```

Example: Adding UDT Transform Groups to Profile

This example creates a profile that specifies the XMLD_STRUCT1INT transform group for the XMLD_STRUCT1 data type and the TD_JSON_VARCHAR transform group for the JSON CHARACTER SET LATIN data type.

```
CREATE PROFILE DR_PROF AS TRANSFORM (XMLD_STRUCT1 = XMLD_STRUCT1INT, JSON
CHARACTER SET LATIN = TD_JSON_VARCHAR);
```

Example: Assigning Row-Level Security Constraint Classifications to Profile

The following example assigns a subset of classifications for two different row-level security constraints to a profile. One of the level classifications is defined as the default. The capabilities of profile members to access row-level security tables with these constraints are defined by the profile constraint assignments.

```
CREATE PROFILE profile_name AS
... ,
CONSTRAINT = Classification_Level (Secret, Unclassified DEFAULT),
CONSTRAINT = Classification_Country (US, UK, GER);
```

Related Information

- GRANT (SQL Form) in *Teradata Vantage™ - SQL Data Control Language*, B035-1149
- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100

MODIFY PROFILE

Changes the parameters for the specified profile.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Other SQL dialects support similar non-ANSI standard statements with names such as ALTER PROFILE.

Required Privileges

To use MODIFY PROFILE, you must have the DROP PROFILE privilege.

To modify a profile to assign a row-level security constraint to it, you must have the CONSTRAINT ASSIGNMENT privilege.

New users do not implicitly have the DROP PROFILE privilege. User *DBC* or a user who has the DROP PROFILE privilege WITH GRANT OPTION can grant this privilege to another user.

Usage Rules and Restrictions for MODIFY PROFILE

Any or all options can be changed using the same MODIFY request. Changes take effect at the next profile member logon after the profile definition is modified.

If an existing option is not changed by a MODIFY PROFILE request, the existing value for that option remains in effect.

If necessary, the system may slightly change the values defined for the PERM, SPOOL, or TEMPORARY space for a profile member to the next higher value that is an even multiple of the number of AMPs on the system.

MODIFY PROFILE Syntax

```
MODIFY PROFILE profile_name AS profile [;]
```

profile

```

{ ACCOUNT = { 'account_string' | ('account_string' [,...] ) | NULL } |

  DEFAULT MAP = { map_name | NULL } [ OVERRIDE [NOT] ON ERROR ] |

  DEFAULT DATABASE = { database_name | NULL } |

  SPOOL = { { n | constant_expression } [BYTES] | NULL } |

  TEMPORARY = { { n | constant_expression } [BYTES] | NULL } |

  PASSWORD [ATTRIBUTES] = { (attribute [,...]) | NULL } |

  QUERY_BAND = { 'pair [...]' [ ([NOT] DEFAULT) ] | ( [NOT] DEFAULT )
| NULL }

  IGNORE QUERY_BAND VALUES = 'pair [...]' |

  TRANSFORM ( [ transformation [,...] ] ) |

  COST PROFILE = { cost_profile_name | NULL } |

  CONSTRAINT = constraint_specification [,...]
}

```

attribute

```
attribute_name = { value | NULL }
```

pair

```
pair_name = pair_value;
```

transformation

```
data_type = group_name
```

constraint_specification

```
row_level_security_constraint_name ({ level_specification [, ...] |
category_name [, ...] | NULL })
```

level_specification

```
level_name [DEFAULT]
```

MODIFY PROFILE Syntax Elements***profile_name***

Name of the profile to be modified.

A profile name must be unique among all profiles.

ACCOUNT

Add one or more accounts to any accounts already specified for the profile.

account_string

Each position in an account string has a specific meaning and format requirement. See *Teradata Vantage™ - Database Administration*, B035-1093. Each *account_string* must follow the Vantage object naming rules. See *Teradata Vantage™ - SQL Fundamentals*, B035-1141. You must enclose each account string with APOSTROPHE characters. You must separate each entry in a list of account strings with a COMMA character and enclose the list with LEFT PARENTHESIS and RIGHT PARENTHESIS characters.

Accounts do not exist independently in the database. Accounts only exist as specified in user, database, and profile definitions. You can specify the same account string in multiple object definitions, if needed.

Account strings defined in the profile supersede the accounts defined in the user definitions of profile member users. Newly added accounts are immediately available.

The first account string specified for the profile is the default account for profile members.

If you do not specify an account string for a profile, the system defaults to the accounts specified for individual users. If a user has no account in either the user or profile definitions, the user inherits the first account string that is defined for the immediate owner of the user.

Users with multiple accounts can access a non-default account by specifying the account in a:

- Logon string.
- SET SESSION ACCOUNT statement.

For a user to specify an account in the logon string or in a SET SESSION ACCOUNT statement, the account must be assigned in the user or profile definition.

DEFAULT MAP

You can specify an existing contiguous or sparse map as the default map for the profile.

You must have been granted the specified map unless the map is the same as the default map of the profile modifier.

map_name

Name of an existing contiguous or sparse map.

You cannot specify TD_DataDictionaryMap or TD_GlobalMap.

To specify a sparse map as the default map, you must be in the same secure zone as the sparse map and the sparse map must be in the same secure zone as the profile.

OVERRIDE ON ERROR

Use the default map if an error occurs when the MAP clause is specified for a CREATE TABLE, CREATE JOIN INDEX, or CREATE HASH INDEX statement. For example, if the MAP clause specifies a nonexistent contiguous map or a sparse map in another secure zone, a warning message displays. The message indicates that the specified map was not used and lists the name of the default map used instead.

This is the default.

OVERRIDE NOT ON ERROR

Do not use the default map if an error occurs when the MAP clause is specified for a CREATE TABLE, CREATE JOIN INDEX, or CREATE HASH INDEX statement.

NULL

A default map is not associated with the database. You can specify NULL to remove the default map that is currently set for the profile.

DEFAULT DATABASE

The default database established each time a user with this profile logs onto Vantage. A change to the default database takes place at the next user logon.

Vantage returns an error if a profile member user tries to create or reference an object within a nonexistent database.

If the default database is NULL or is not defined in the profile assigned to a user, Vantage uses the setting defined for the individual user.

database_name

Name of the default database.

SPOOL

Maximum number of bytes allowed for spool files in *profile_name*. The default is null, which uses the setting defined for the individual user assigned to the profile.

Changes to spool space limits in a profile take effect immediately upon submitting the MODIFY PROFILE request.

n

You can enter the number of bytes as an integer, decimal, or floating point value or as a constant expression whose evaluation determines the number of bytes. You can also enter the value using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

n cannot exceed the spool space parameter in the profile of the creator. If no spool space is defined for that profile, then Vantage uses the spool space limit defined for the individual user-creator.

constant_expression

A constant expression is any SQL expression that does not make any column references. Specifying an appropriate constant expression for the SPOOL size of a database enables Vantage to assign an optimal quantity of SPOOL that scales to the size of your system.

When you specify a SPOOL size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

Optional keyword that redundantly specifies the unit for the amount of space allowed.

TEMPORARY

The number of bytes to allocate for global temporary table space.

If no temporary space is defined in the profile assigned to a user, Vantage uses the setting defined for the individual user. Temporary space is reserved prior to spool space for any user defined with this

characteristic. Disk usage for a materialized global temporary table is charged to the temporary space allocation of the user who referenced the table.

The default is NULL, which uses the setting defined for the individual user. Changes to the temporary space allocation in a profile take effect immediately upon submitting the MODIFY PROFILE request.

n

n can be an integer, a decimal value, or a floating point value.

n cannot exceed the temporary space parameter in the profile of the creator. If no temporary space limit is defined for that profile, then Vantage uses the temporary space limit defined for the individual user-creator.

n refers to bytes, whether or not the optional BYTES keyword is specified.

constant_expression

Any SQL expression that does not make any column references. Specifying an appropriate constant expression for the TEMPORARY space size of a user enables Vantage to assign an optimal quantity of TEMPORARY space that scales to the size of your system by allocating TEMPORARY space on a per AMP basis.

When you specify a TEMPORARY space size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

Optional keyword.

PASSWORD

If you specify the PASSWORD or PASSWORD ATTRIBUTES phrase, you must list one or more password control option and specify either a value or NULL (the default) for each.

Password security attributes in a modified profile take effect the next time a profile member user logs on after the modification.

Password controls only affect users authenticated by the database. Externally authenticated users are unaffected.

For detailed description of password control methods, options, and strategies, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

Note:

Character-related password format controls do not apply to multibyte client character sets on systems enabled with Japanese language support.

Note:

Some password control attribute values are not applicable to first-level profile members, but only to children of profile member users. See [Effects of Profile-Based Password Controls](#).

ATTRIBUTES

Keywords that introduces a set of password control attributes. Following are brief descriptions of the password controls.

attribute_name = value

If you specify a value in a profile for a password control, the value supersedes the global value for profile members.

attribute_name = NULL

If the value for an attribute is NULL or if you do not specify the PASSWORD or PASSWORD ATTRIBUTES phrase, the profile defaults to the global password control settings in DBC.SysSecDefaults.

EXPIRE=*n*

The number of days to elapse before the password expires.

A value of 0 for *n* indicates the password never expires.

NULL indicates that the EXPIRE option is not set for the profile.

You can specify any non-zero value to cause new users who are profile members to replace the temporary password specified in their user definitions with a permanent private password at first log on. Users must use the MODIFY USER statement to change their password.

MINCHAR=*n*

The minimum number of characters in a password string.

The valid range for *n* is 1-127 UNICODE characters.

NULL indicates that the MINCHAR option is not set for the profile.

MAXCHAR=*n*

The maximum number of characters in a password string.

The valid range for *n* is 1-127 UNICODE characters.

NULL indicates that the MAXCHAR option is not set for the profile.

DIGITS=*c*

Specifies whether at least one digit must appear in a password string.

- If *c* is N or n, digits are not permitted in a password string.
- If *c* is R or r, *At least* one digit is required in a password string.
- If *c* is Y or y, digits are permitted in a password string, but not required.
- NULL indicates that the DIGITS option is not set for the profile.

SPECCHAR=*c*

Specifies whether various characters or the user name are allowed, not allowed, or required in a user password string.

The value for SPECCHAR must be one of the single letter option codes shown in [Working with the SPECCHAR Password Control](#), where the letter code you specify for the SPECCHAR option represents a unique set of possible SPECCHAR rules.

MAXLOGONATTEMPTS=*n*

Number of incorrect logon attempts allowed before locking the user from further attempts, where *n* is a value from 0 to 127.

A value of 0 for *n* indicates never to lock the user.

NULL indicates that the MAXLOGONATTEMPTS option is not set for the profile.

LOCKEDUSEREXPIRE=*n*

Number of minutes to elapse before unlocking a locked user.

- If *n* is 0, Vantage unlocks the user immediately.
- If *n* is -1, Vantage locks the user indefinitely.
- NULL indicates that the LOCKEDUSEREXPIRE option is not set for the profile.

REUSE=*n*

Number of days to elapse before a password can be reused.

A value of 0 for *n* allows the password to be reused immediately.

NULL indicates that the REUSE option is not set for the profile.

RESTRICTWORDS=*c*

Specifies whether certain words are restricted from use within a password string.

The valid values for *c* are listed below.

- If *c* is Y or y, any words listed in DBC.PasswordRestrictions cannot be used in password strings.
- If *c* is N or n, use of words listed in DBC.PasswordRestrictions in password strings is allowed.
- NULL indicates that the RESTRICTWORDS option is not set for the profile.

For details, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100 and *Teradata Vantage™ - Database Administration*, B035-1093.

QUERY_BAND

Modify the default query band defined in the profile. The name-value pairs you specify replace any existing query band settings. Changes to the profile query band of currently logged on users take effect on the next request.

You must enclose the set of *name=value* pairs for the query band with APOSTROPHE characters (').

The number of *name=value* pairs is limited to the maximum length of the string, which is 4,096 UNICODE characters, including pad characters.

Names and values cannot contain any of the following characters:

- SEMICOLON (;)
- a null

If an APOSTROPHE character is embedded within a pair name or value, you must type it twice, to escape it. Otherwise, the system interprets it as a *pair_name=pair_value* string terminator.

Note:

Do not specify reserved pair names or values. For a list of reserved query band names and values, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

QUERY_BAND

Keyword to introduce query band options you specify.

pair_name

Name component of a query band specification.

The maximum size for each *pair_name* is 128 UNICODE characters. For more information about database object names, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

pair_value

Value component of a query band specification.

pair_value can be up to 256 UNICODE characters in length.

NULL

Removes the entire query band from the profile.

DEFAULT

The values are set as default values that can be overwritten using the SET QUERY_BAND statement for the session or transaction. See [SET QUERY_BAND](#).

NOT DEFAULT

Overwriting the pairs in the profile query band is not permitted. Any pair in a SET QUERY_BAND statement matching a name in the profile query band will be discarded from the query band.

IGNORE QUERY_BAND VALUES

Defines the set of name-value pairs to ignore if specified in a SET QUERY_BAND statement for all users assigned the profile. When a user with the profile issues a SET QUERY_BAND statement containing a name-value pair matching a IGNORE QUERY_BAND VALUES name-value pair, the system performs the following actions:

- The matching name-value pair is ignored.
- The query band is set with the remaining name-value pairs.
- A warning message displays.

You must enclose the set of *name=value* pairs for the query band with APOSTROPHE characters (').

The number of *name=value* pairs is limited to the maximum length of the string, which is 4,096 UNICODE characters, including pad characters.

Pair names and values cannot contain any of the following characters:

- SEMICOLON (;)
- a null

Setting a *pair_name* to an empty string value indicates that a name-value pair with the *pair_name* and any value will be discarded. For example:

```
pair_name=;
```

To specify multiple values for the same *pair_name*, each name-value pair must be specified separately. For example:

```
IGNORE QUERY_BAND VALUES =  
'TVSTemperature=HOT;TVSTemperature=WARM;'
```

If an APOSTROPHE character is embedded within a pair name or value, you must type it twice, to escape it. Otherwise, the system interprets it as a *pair_name=pair_value* string terminator.

If a profile has a query band that is defined as NOT DEFAULT, it is redundant to define the IGNORE QUERY_BAND VALUES with name-value pairs having the same names as defined in the profile query

band. The NOT DEFAULT profile query band causes the SET QUERY_BAND statement to discard any name-value pair matching the names in the profile query band.

Changes to the profile IGNORE QUERY_BAND VALUES of currently logged on users do not affect existing query bands in the session. Subsequent SET QUERY_BAND statements in the session are validated according to the updated profile IGNORE QUERY_BAND VALUES.

IGNORE QUERY_BAND VALUES

Keywords to introduce query band options to ignore.

pair_name

Name component of a query band specification.

The maximum size for each *pair_name* is 128 UNICODE characters. For more information about database object names, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

pair_value

Value component of a query band specification.

pair_value can be up to 256 UNICODE characters in length.

NULL

Removes the entire ignore query band setting from the profile.

TRANSFORM

Adds UDT transform groups to the profile or removes UDT transform groups from the profile.

You can specify up to 256 *data_type-transform_group* pairs for a profile.

You can only add, replace, or remove entire transform group settings. You cannot add or remove individual *data_type-transform_group* pairs.

After transform group settings are changed, the user must log out and log in again to use the new settings.

data_type

Name of the user-defined data type (UDT).

group_name

Name of transform group for the data type.

COST PROFILE

Cost profile to be associated with *profile_name*.

cost_profile_name

Name of cost profile.

NULL

The default is NULL, meaning use the default system profile.

Note:

Optimizer cost profiles are not intended for use on production systems. The COST PROFILE parameter is for use only under the direction of Teradata Support Center personnel.

CONSTRAINT

Up to 6 hierarchical constraints and 2 non-hierarchical constraints can be assigned per profile.

For more information about row-level security constraints, see [CREATE CONSTRAINT](#).

Changes take effect at the next logon by profile members after the profile definition is modified.

row_level_security_constraint_name

Name of one or more row-level security constraints, each followed by a list of the hierarchical levels or non-hierarchical categories, valid for the constraint, which are being assigned to the *profile_name*.

level_name

List of hierarchical levels. Example of hierarchical level constraint assignment:

```
CONSTRAINT = Classification_Level (Secret, Unclassified DEFAULT)
```

DEFAULT

DEFAULT can be specified for 1 classification level from the specified list of levels for a hierarchical constraint. The specified level becomes the default value for the constraint when a session is established for the user to which the profile is assigned.

If you do not specify DEFAULT for any of the listed classification levels, then the first level listed becomes the default for the profile.

DEFAULT is not valid for non-hierarchical constraints.

category_name

List of non-hierarchical categories. Example non-hierarchical category constraint assignment:

```
CONSTRAINT = Classification_Country (US, UK, GER)
```

If the constraint is already assigned to the profile, the level and category names you specify for the constraint replace existing level and constraint specifications.

NULL

If you specify NULL for a constraint previously assigned to the profile, the constraint is dropped from the profile definition.

MODIFY PROFILE Examples

Example: Modify profile spool space

Assume user *anderson* has a spool setting of 500,000 bytes and user *brewster* has a spool setting of 800,000 bytes.

User *anderson* can create a profile named *finance* with a spool setting of 500,000 bytes.

```
CREATE PROFILE finance AS
SPOOL = 500000;
```

User *anderson* can assign this profile to user *brewster*, effectively reducing the spool limit from 800,000 to 500,000 bytes.

```
MODIFY USER brewster AS
PROFILE = finance;
```

If the owner of *anderson* later increases the spool allotment to 1,000,000, *anderson* can also increase the *finance* profile spool setting up to 1,000,000 bytes. All users assigned that profile, including *brewster*, automatically get 1,000,000 bytes of spool space.

```
MODIFY PROFILE finance AS
SPOOL = 1000000;
```

If the owner of *anderson* later decreases the spool allotment back to 500,000 bytes, the *finance* profile and the allotment for *brewster* are not affected.

Example: Using a Constant Expression to Specify the SPOOL Space for a Profile

The following request modifies the SPOOL space for the profile *research_and_development* to a size based on the constant expression 3,000,000 (HASHAMP()+1). The expression calculates the number of AMPs in the current system and scales the SPOOL space for the *research_and_development* profile to that size.

This is the original definition of profile *research_and_development*.

```
CREATE PROFILE research_and_development AS
  DEFAULT DATABASE = it_dev,
  PASSWORD = (EXPIRE = 0),
  SPOOL = 2000000*(HASHAMP()+1);
```

The following MODIFY PROFILE request alters the SPOOL space allocation for this profile as follows.

```
MODIFY PROFILE research_and_development AS
  DEFAULT DATABASE = it_dev,
  PASSWORD = (EXPIRE = 0),
  SPOOL = 3000000*(HASHAMP()+1);
```

Example: Using Constant Expression to Specify TEMPORARY Space for Profile

This is the original definition of the *research_and_development*.profile.

```
CREATE PROFILE research_and_development AS
  DEFAULT DATABASE = it_dev,
  PASSWORD = secret,
  TEMPORARY = 2000000*(HASHAMP()+1);
```

The following statement changes the TEMPORARY space for the profile *research_and_development* to a size based on the constant expression 3,000,000 (HASHAMP()+1). The expression calculates the number of AMPs in the current system and scales the TEMPORARY space for the *research_and_development* profile to that size.

```
MODIFY PROFILE research_and_development AS
  DEFAULT DATABASE = it_dev,
  PASSWORD = secret,
  TEMPORARY = 3000000*(HASHAMP()+1);
```

Examples: QUERY_BAND

```
MODIFY PROFILE salesprofile AS
  QUERY_BAND = (NOT DEFAULT);
```

```
MODIFY PROFILE testprofile AS
  QUERY_BAND = NULL;
```

```
MODIFY PROFILE research_and_development AS
  QUERY_BAND = 'jobtype=test;' (DEFAULT);
```

Example: Modifying Profile to Set Default Query Band Value and Query Band Value to Ignore

In this example, the user's profile query band is set to TVSTemperature=WARM. If the user sets the query band to TVSTemperature=HOT, the name-value pair is discarded from the query band.

```
MODIFY PROFILE salesprofile AS,
  QUERY_BAND = 'TVSTemperature=WARM;' DEFAULT,
  IGNORE QUERY_BAND VALUES ='TVSTemperature=HOT';
```

Example: Adding UDT Transform Groups to Profile

This example modifies a profile to specify the XMLD_STRUCT1INT transform group for the XMLD_STRUCT1 data type and the TD_JSON_CLOB transform group for the JSON CHARACTER SET LATIN data type.

```
MODIFY PROFILE DR_PROF AS TRANSFORM (XMLD_STRUCT1 = XMLD_STRUCT1INT, JSON
CHARACTER SET LATIN = TD_JSON_CLOB);
```

Example: Removing UDT Transform Groups from Profile

This example removes the UDT transform groups from the profile.

```
MODIFY PROFILE DR_PROF AS TRANSFORM ();
```


Related Information

- [CREATE USER](#)
- [MODIFY USER](#)
- “GRANT (SQL Form)” topic in *Teradata Vantage™ - SQL Data Control Language*, B035-1149
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100

CREATE ROLE

Creates a role for managing user access privileges on database objects. A role is a shell database object to which sets of privileges can be granted using GRANT requests.

You can use a role to grant a set of privileges that are commonly needed by a group of users, for example, users in the Finance department.

After creating a role and granting database privileges to the role, you can grant role membership to the users that require those privileges.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Required Privileges

You must have the CREATE ROLE privilege to create a standard database role or EXTERNAL role.

New users do not implicitly have the CREATE ROLE privilege.

User DBC or a user who has the CREATE ROLE WITH GRANT OPTION privilege can grant this privilege to another user, but you cannot grant the WITH GRANT OPTION privilege to a role.

Type of Role	WITH ADMIN OPTION Privilege
Database	Granted implicitly to its creator. This permits the creator to grant the role to other users and roles.
External	Not granted to its creator, because you cannot grant an external role to database users or roles. For information on assigning database external roles to directory users, see Security Administration

Privileges Granted Automatically

None.

CREATE ROLE Syntax

```
CREATE [EXTERNAL] ROLE role_name [;]
```

CREATE ROLE Syntax Elements

EXTERNAL

The role named by *role_name* is an external role.

External roles exist in the database, but can only be assigned to users in an LDAP-compliant directory.

For information about how to create directory objects that correspond to database external roles and map the objects to directory users (using group objects), see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

role_name

Name of the role.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Database and external roles share the same name space.

Databases and external roles can have the same name as a profile, table, column, view, macro, trigger, table UDF, external SQL procedure, method, UDT, or SQL procedure. However, role names must be unique among users and databases.

Example: Creating an External Role

The following request creates an external role named *rh* that is managed by the Vantage, for use in assigning privileges to users in a directory:

```
CREATE EXTERNAL ROLE rh;
```

Related Information

- Role and SQL forms of GRANT in *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

CREATE USER

Creates a permanent database user object, in which other database objects may be created, with a mandatory permanent space allocation and a mandatory password plus optional attributes.

Optional attributes can include:

- Startup string.
- Maximum spool space assignment.
- Maximum temporary space assignment.
- Default database.
- Default collation sequence.
- Default account.
- Default map.
- Default level of fallback protection.
- Before journaling.
- After journaling.
- Default journal table.
- Default time zone.
- Default date form type.
- Default server character set.
- Default role.
- Profile.
- Transform group.
- One or more row-level security constraints.

For information about definition and administration of external (directory-based) users, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

Note:

Where a specific setting exists in the user definition and the assigned user profile, the profile setting takes precedence.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

CREATE USER is functionally similar to the ANSI SQL:2011 CREATE SCHEMA statement.

Required Privileges

To create a user, you must have the CREATE USER privilege on the immediate owner database or user.

To include a profile specification, you must have the DROP PROFILE privilege.

To include the CONSTRAINT = *row_level_security_constraint_name* option in a CREATE USER statement, you must also have the CONSTRAINT ASSIGNMENT privilege.

For more information, see [Privileges Received by the Creator](#).

Privileges Granted Automatically

See [Privileges Granted Automatically to a Created User](#) and [Privileges Not Granted Automatically](#).

CREATE USER Syntax

```
CREATE USER user_name [ FROM database_name ] AS
  { PERMANENT | PERM } = n [BYTES] [ skew_specification ] [,]
  PASSWORD = temp_password [,]
  [ user_attribute [,]... ] [;]
```

skew_specification

```
SKEW = { constant_expression | DEFAULT } [PERCENT]
```

user_attribute

```
{ STARTUP = 'string;' |

  { TEMPORARY | SPOOL } = { n | constant_expression } [BYTES] [
skew_specification ] |

  DEFAULT DATABASE = database_name |

  COLLATION = collation_sequence |

  ACCOUNT = { 'account_string' | ( 'account_string' [,...] ) | NULL } |

  DEFAULT MAP = { map_name | NULL } [ OVERRIDE [NOT] ON ERROR ] |

  [NO] FALLBACK [PROTECTION] |

  [ NO | DUAL ] [BEFORE] JOURNAL |

  [ NO | DUAL | [NOT] LOCAL ] AFTER JOURNAL |

  DEFAULT JOURNAL TABLE = [ database_name. ] table_name |

  TIME ZONE = { LOCAL | [ sign ] 'quotestring' | 'time_zone_string' |
  NULL } |

  DATEFORM = { INTEGERDATE | ANSIDATE | NULL } |
```

```

DEFAULT CHARACTER SET server_character_set |

DEFAULT ROLE = { role_name | NONE | NULL | ALL } |

PROFILE = { profile_name | NULL } |

TRANSFORM ( transform_specification [,...]) |

DBA |

CONSTRAINT = constraint [,...] |

EXPORTWIDTH { 'export_definition_name' | DEFAULT }
}

```

transform_specification

```
data_type = group_name
```

constraint

```

row_level_security_constraint_column_name {
  ( { level_name [DEFAULT] | category_name } [,...]) |
  (NULL)
}

```

CREATE USER Syntax Elements***user_name***

Name of the user being created.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

database_name

Name of the immediate owner database.

The default is the current creator of this user.

user_name

Name of the immediate owner user.

The default is the current creator of this user.

AS

An introduction to the first user options.

Subsequent options can be entered in any order.

PERMANENT

Number of bytes to be reserved for permanent storage for this user. The space is taken from unallocated space in the database or user of the immediate owner.

You must specify a value for this option. There is no default.

A global temporary table requires a minimum of 512 bytes from the PERM space of its containing user or database. This space is used for its table header.

You cannot specify any other options for the new user until after you specify the PERMANENT and PASSWORD options.

Note:

The only significant difference between a user and a database is that you can specify a password for a user, but not for a database.

n

You can enter the number of bytes as an integer, decimal, or floating point value or as a constant expression whose evaluation determines the number of bytes. You can also enter the value using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

To create a user without database component, specify a value of 0 bytes for PERMANENT space. You must still specify a SPOOL space value for the user.

constant_expression

Any SQL expression that does not make any column references. Specifying an appropriate constant expression for the PERM space size of a user enables Vantage to assign an optimal quantity of PERM space that scales to the size of your system by allocating PERM space on a per AMP basis.

When you specify a PERM space size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

Optional keyword that redundantly specifies the unit for the amount of space allowed.

SKEW

Keyword that you use to specify a skew limit for PERMANENT space. You can specify a skew limit percentage that allows the maximum AMP space usage to be above the per-AMP quota, that is, the system maximum space limit divided by the number of AMPs.

constant_expression

Constant expression or constant from 0 through 10000. Specify a value from 1 to 9999 to indicate an AMP-level limit, which is the per-AMP quota * $(1 + \text{permskewlimit}/100)$. Specify 0 to set space accounting to the per-AMP level, that is, no skew. A value of 10000 indicates unlimited skew, up to the system maximum space limit.

DEFAULT

Use the value of DBS Control DefaultPermSkewLimitPercent.

PERCENT

Optional keyword that you can include for readability to indicate that the *constant_expression* or DEFAULT keyword specifies a percentage of allowable skew.

PASSWORD

A password associated with the new user.

You must specify a password.

You cannot specify any other options for the new user until after you specify the PERMANENT and PASSWORD options.

Note:

If the user submitting the CREATE USER statement is a member of a profile that specifies values for password control attributes, the password the creator assigns in the CREATE USER statement must comply with the password controls in the creator profile. Applicability of profile-based password control attributes to first-level members of a profile vary. See [Effects of Profile-Based Password Controls](#).

For detailed information on password format requirements and using password controls, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

password

Best security practice suggests that you not assign permanent passwords using CREATE USER requests. Instead, the recommended practice is to assign a temporary password when you create a user, then set the EXPIRE password control attribute to a non-zero value to require users to create private passwords when they first log onto Vantage. See the PASSWORD ATTRIBUTES option in [CREATE PROFILE](#).

STARTUP

One or more optional SQL requests, separated by SEMICOLON characters, that are performed to establish the session environment when the user logs on.

string

Strings can be up to 255 characters, must be terminated by a SEMICOLON character, and must be enclosed by APOSTROPHE characters. The default is null, meaning no startup string.

A startup string can perform a macro. However, the USING request modifier is not supported in a startup string. If a string includes a DDL statement, no other statement is allowed in the string.

You can also include a SET SESSION CALENDAR request in a startup string to set the calendar for a user to something other than the default Teradata calendar. The other available system-defined calendars are named ISO and COMPATIBLE. See SET SESSION CALENDAR in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for more information about this.

The startup string is performed either when you log on to Vantage through BTEQ or when you establish a JDBC connection using the Teradata JDBC driver and have specified the connection parameter RUNSTARTUP=ON. BTEQ and JDBC are the only client applications that support startup strings.

All other Teradata client APIs ignore the string when Vantage returns it to them.

TEMPORARY

Temporary space allowed for creating materialized global temporary tables by this user. Temporary space is reserved prior to pool space for any user defined with this characteristic.

If no temporary space is defined for a user, then the space allocated for any materialized global temporary tables referenced by that user is set to the maximum temporary space allocated for the immediate owner.

Note that each materialized global temporary table requires a minimum of 512 bytes of PERM space to contain its table header.

Disk usage for a materialized temporary table is charged to the temporary space allocation of the user who referenced the table.

n

Number of bytes allowed for creating materialized global temporary tables by this user.

constant_expression

Any SQL expression that does not make any column references. Specifying an appropriate constant expression for the TEMPORARY space size of a user enables Vantage to assign an optimal quantity of TEMPORARY space that scales to the size of your system by allocating TEMPORARY space on a per AMP basis.

When you specify a TEMPORARY space size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

Optional keyword that redundantly specifies the unit for the amount of TEMPORARY space allowed.

SKEW

Keyword that you use to specify a skew limit for TEMPORARY space. You can specify a skew limit percentage that allows the maximum AMP space usage to be above the per-AMP quota, that is, the system maximum space limit divided by the number of AMPs.

constant_expression

Constant expression or constant from 0 through 10000. Specify a value from 1 to 9999 to indicate an AMP-level limit, which is the per-AMP quota * $(1 + \text{temp_skew_limit}/100)$. Specify 0 to set space accounting to the per-AMP level, that is, no skew. A value of 10000 indicates unlimited skew, up to the system maximum space limit.

DEFAULT

Use the value of DBS Control DefaultTempSkewLimitPercent.

PERCENT

Optional keyword that you can include for readability to indicate that the *constant_expression* or DEFAULT keyword specifies a percentage of allowable skew.

SPOOL

Number of bytes allowed for spool files.

The default is the largest value that is not greater than the owner spool space, and that is a multiple of the number of AMPs on the system.

The default is the amount of spool space allocated to the owner of this user. As a general guideline, specify a minimum of 20 percent of the permanent space allocated for this user. See *Teradata Vantage™ - Database Design*, B035-1094 for details.

Neither *n* nor the evaluation of *constant_expression* can exceed the size of the owner spool space.

Note:

If you specify a PERMANENT space value of 0 bytes for a user, you must still specify some minimum number of bytes for its SPOOL space.

n

You can enter the number of bytes as an integer, decimal, or floating point value or as a constant expression whose evaluation determines the number of bytes. You can also enter the value using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

n cannot exceed the spool space parameter in the profile of the creator. If no spool space is defined for that profile, then Vantage uses the spool space limit defined for the individual user-creator.

constant_expression

A constant expression is any SQL expression that does not make any column references. Specifying an appropriate constant expression for the SPOOL space size of a user enables Vantage to assign an optimal quantity of spool that scales to the size of your system by allocating SPOOL space on a per AMP basis.

When you specify a SPOOL space size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

Optional keyword that redundantly specifies the unit for the amount of space allowed.

SKEW

Keyword that you use to specify a skew limit for SPOOL space. You can specify a skew limit percentage that allows the maximum AMP space usage to be above the per-AMP quota, which is the system maximum space limit divided by the number of AMPs.

constant_expression

Constant expression or constant from 0 through 10000. Specify a value from 1 to 9999 to indicate an AMP-level limit, which is the per-AMP quota * (1+*spoolskewlimit*/100). Specify

0 to set space accounting to the per-AMP level, that is, no skew. A value of 10000 indicates unlimited skew, up to the system maximum space limit.

DEFAULT

Use the value of DBS Control DefaultSpoolSkewLimitPercent.

PERCENT

Optional keyword that you can include for readability to indicate that the *constant_expression* or DEFAULT keyword specifies a percentage of allowable skew.

DEFAULT DATABASE

Default database established each time this user logs onto Vantage.

database_name

Name of a default database.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

The default is the user name associated with the current session.

database_name need not be an existing database. However, Vantage returns an error when you try to create or reference an object within a nonexistent database.

COLLATION

Collation to be used as the default for this user.

collation_sequence

Name of the collation.

The maximum length of *collation_sequence* is 30 characters.

The default is COLLATION = HOST.

The following keywords represent the collation sequences supported by Vantage.

- ASCII
- CHARSET_COLL
- EBCDIC
- HOST
- JIS_COLL
- MULTINATIONAL

For details, see [Collation Option Usage](#) and [MULTINATIONAL Collation](#).

ACCOUNT

One or more accounts to be assigned to the user. See *Teradata Vantage™ - Database Administration*, B035-1093.

Account strings defined in the user profile supersede the accounts defined in the user definition.

If you do not specify an account string for the user or the user profile, the user inherits the first account string that is defined for the immediate owner of the user, that is, another user or a database.

Users with multiple accounts can use one of the following methods to access a non-default account:

- Logon string.
- SET SESSION ACCOUNT statement.

For a user to specify an account in the logon string or in a SET SESSION ACCOUNT statement, the account must be assigned in the user or profile definition for the user.

account_string

Each position in an account string has a specific meaning and format requirement. See *Teradata Vantage™ - Database Administration*, B035-1093. Each *account_string* must follow the Vantage object naming rules. See *Teradata Vantage™ - SQL Fundamentals*, B035-1141. You must enclose each account string with APOSTROPHE characters. You must separate each entry in a list of account strings with a COMMA character and enclose the list with LEFT PARENTHESIS and RIGHT PARENTHESIS characters.

Accounts do not exist independently in the database. Accounts only exist as specified in user, database, and profile definitions. You can specify the same account string in multiple object definitions, if needed.

The first account string you specify becomes the default account for the user, but you can specify multiple accounts if needed.

DEFAULT MAP

You can specify an existing contiguous or sparse map as the default map for the user.

You must have been granted the specified map unless the map is the same as the default map of the user creator.

To specify a sparse map as the default, you must be in the same secure zone as the sparse map and the sparse map must be in the same secure zone as the user.

map_name

Name of an existing contiguous or sparse map.

You cannot specify TD_DataDictionaryMap or TD_GlobalMap.

OVERRIDE ON ERROR

Use the default map if an error occurs when the MAP clause is specified for a CREATE TABLE, CREATE JOIN INDEX, or CREATE HASH INDEX statement. For example, if the MAP clause specifies a nonexistent contiguous map or a sparse map in another secure zone, a warning message displays. The message indicates that the specified map was not used and lists the name of the default map used instead.

This is the default.

OVERRIDE NOT ON ERROR

Do not use the default map if an error occurs when the MAP clause is specified for a CREATE TABLE, CREATE JOIN INDEX, or CREATE HASH INDEX statement.

NULL

A default map is not associated with the user.

FALLBACK

Specifies whether to create and store a duplicate copy of each table created in the new user.

PROTECTION

Optional keyword.

NO

Do not create and store a duplicate copy of each table created in the new user.

You can override this setting for a particular table when you create the table. See [CREATE TABLE](#) and [CREATE TABLE AS](#).

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

JOURNAL

Change images are maintained for each data table created in the new user.

Specifying the JOURNAL keyword without also specifying either NO or DUAL implies single copy journaling.

If journaling is specified, a DUAL journal is maintained for data tables with FALLBACK protection.

The JOURNAL keyword without BEFORE implies both types (BEFORE and AFTER) of images.

Journal options are not supported for tables with row sizes greater than 64KB.

NO

Change images are not maintained for each data table created in the new user.

DUAL

Dual change images are maintained by default for each data table created in the new user.

BEFORE

Number of BEFORE change images to be maintained by default for each data table created in the new user.

AFTER JOURNAL

After-image journal rows are maintained by default for data tables created in the new user.

A user can contain only one journal table.

The default is no journaling.

If only AFTER JOURNAL is specified, then a before change image is not maintained. If both types are specified, the two specifications must not conflict.

This setting can be overridden for a particular data table when the table is created (see [CREATE TABLE](#) and [CREATE TABLE AS](#)).

Specifying the JOURNAL keyword without also specifying either NO or DUAL implies single copy journaling.

If journaling is specified, a DUAL journal is maintained for data tables with FALLBACK protection.

The JOURNAL keyword without AFTER implies both types (BEFORE and AFTER) of images.

NO

After-image journal rows are not maintained by default for data tables created in the new user.

DUAL

Dual after-image journal rows for non-fallback data tables are maintained.

LOCAL

Single after-image journal rows for non-fallback data tables are written on the same virtual AMP as the changed data rows.

NOT LOCAL

Single after-image journal rows for non-fallback data tables are written on another virtual AMP in the cluster.

DEFAULT JOURNAL TABLE

Default table that is to receive the journal images of data tables created in the new user.

table_name

table_name must be defined if journaling is requested, but it need not be contained within the new user.

table_name is automatically created in the new database if *database_name* is not specified.

database_name

If a different database is specified, then that database must exist and *table_name* must have been defined as its default journal table.

user_name

If a different user is specified, then that user must exist and *table_name* must have been defined as its default journal table.

TIME ZONE

Default time zone displacement for the specified user.

You can specify these options:

- LOCAL
- NULL
- \pm 'quotestring'
- 'time_zone_string'

See CREATE USER in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for the definitions of these options and for a comprehensive list of the time zone strings supported by Vantage.

DATEFORM

Default format for importing and exporting DATE values for the user.

DATEFORM = INTEGERDATE is the default.

For further information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

ANSIDATE

Sets the DATEFORM option to import and export DATE values as CHARACTER(10).

Results in a 'YYYY-MM-DD' date format for date columns created and for date constants in character form.

INTEGERDATE

Sets the DATEFORM option to import and export DATE values as encoded integers.

INTEGERDATE results in a default DATE format in field mode of 'YY/MM/DD' for date columns created and for date constants in character form. The 'YY/MM/DD' default DATE format can be changed by using the tdlocaledef utility. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

INTEGERDATE is the default.

NULL

Sets DATEFORM null.

When DATEFORM is null, the default format for a date in field mode, and the default format for importing and exporting DATE values, is the same as the system default defined by the DATEFORM parameter in the DBS Control record.

DEFAULT CHARACTER SET

User default for server character set.

If you do not assign a default server character set, Vantage uses the setting of the DBS Control parameter Default Character Set.

LATIN

Fixed 8-bit Latin. This is the default on Standard Language Support mode systems.

UNICODE

Fixed 16-bit encoding of virtually all characters for all languages currently in use throughout the world. This is the default on Japanese Language Support mode systems.

KANJISJIS

Mixed single and multibyte characters defined by KanjiSJIS.

You cannot specify a server character set of KANJI1 or GRAPHIC. Otherwise, the system returns an error.

For details about server character set options, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

DEFAULT ROLE

Name of a role (which may include nested roles) to assign as the default role for the user. The role must already be granted to the user.

The assignment of a default role provides a user with an enabled role at logon. If you do not assign a default role to the user, or if the user wants to enable a different assigned role, the user must submit a SET ROLE request to enable the needed role for the session. See [SET ROLE](#).

ALL

Enables all roles currently granted to the user, including any nested roles.

NONE

Do not assign a default role to the user.

NULL

Do not assign a default role to the user.

PROFILE

Assign a profile to the user. See [CREATE PROFILE](#) for details.

Note:

For attributes that appear in both the user definition and the profile definition, the profile-based attributes supersede and take precedence over those in the user definition.

profile_name

Name of a profile to assign to the user.

NULL

No profile exists for the user.

TRANSFORM

Associates UDT transform groups with the user.

You can specify up to 256 *data_type-transform_group* pairs for a user.

If the user has a profile, the transform group for the UDT in the user's profile is used.

If the UDT has a default transform group, the default transform group for the UDT is used unless the user specifies a transform group.

data_type

Name of user-defined data type (UDT).

group_name

Name of transform group for the UDT.

DBA

Database Administrator (DBA) option.

DBA users can perform DML operations on objects they create in their own user-space.

DBA users can perform DML operations on objects created by other users if the DBA user has been granted the required discretionary access control (DAC) privileges.

DBA users cannot perform DML operations on objects they create in another user's space.

CONSTRAINT

Assign one or more row-level security constraints to the *user_name*.

There is a limit of 6 hierarchical constraints and 2 non-hierarchical constraints that can be assigned per user.

For more information about row-level security constraints, see [CREATE CONSTRAINT](#).

row_level_security_constraint_column_name

Name of one or more row-level security constraints, each one followed by the list of the hierarchical levels or non-hierarchical categories, valid for the constraint, which are being assigned to the *user_name*.

- Example hierarchical constraint assignment:

```
CONSTRAINT = Classification_Level (Secret, Unclassified DEFAULT),
```

- Example non-hierarchical constraint assignment:

```
CONSTRAINT = Classification_Country (US, UK, GER)
```

level_name

Hierarchical levels, valid for the constraint, which are being assigned to the *user_name*.

DEFAULT

DEFAULT can be specified for 1 classification level from the specified list of level names for a hierarchical constraint. The specified level becomes the default value for the constraint when a session is established for the user.

If you do not specify DEFAULT for any of the listed classification levels, then the first level listed becomes the default for the user.

DEFAULT is not valid for non-hierarchical category constraints.

category_name

Non-hierarchical categories, valid for the constraint, which are being assigned to the *user_name*.

NULL

EXPORTWIDTH

The export width for this user, enclosed in SINGLE QUOTATION MARK characters.

Vantage uses the value you assign to a user as a NUPI to access *DBC.ExportWidth*, which defines the rules associated with the different export width definitions for Latin, Unicode, KanjiSJIS, and Graphic strings. See *Teradata Vantage™ - Data Dictionary*, B035-1092 for more information about *DBC.ExportWidth*.

export_definition_name

Vantage supports the keyword DEFAULT as an export width definition string as well as the following predefined export width definition strings:

- COMPATIBILITY
- EXPECTED
- MAXIMUM

For a detailed description of the valid export width specifications, see [Valid Export Width Specifications](#).

Also see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125 for detailed information about export widths and how to define your own export width table.

DEFAULT

If you do not specify an EXPORTWIDTH value for a user or specify the keyword DEFAULT, Vantage uses the export width currently specified by the DBSControl Export Width Table ID parameter. See Export Width Table ID in *Teradata Vantage™ - Database Utilities*, B035-1102.

Usage Notes

Privileges Received by the Creator

The creator receives the following privileges on the newly created user, WITH GRANT OPTION:

- CHECKPOINT
- CREATE DATABASE
- CREATE MACRO
- CREATE TABLE
- CREATE TRIGGER
- CREATE USER
- CREATE VIEW
- DELETE
- DROP DATABASE
- DROP
- DROP MACRO
- DROP TABLE
- DROP TRIGGER
- DROP USER
- DROP VIEW
- DUMP
- EXECUTE
- INSERT
- REVOKE
- RESTORE
- UPDATE

Privileges Granted Automatically to a Created User

Newly created users are automatically granted the privileges needed to create and maintain objects in their space, but must be explicitly granted the privileges needed to CREATE USER/MODIFY USER or CREATE DATABASE (see also GRANT (SQL Form) in *Teradata Vantage™ - SQL Data Control Language*, B035-1149).

Note:

Newly created users do not receive WITH GRANT OPTION privileges for any of the automatically granted privileges.

The following privileges are automatically granted to a user when it is created.

- CHECKPOINT
- CREATE AUTHORIZATION
- CREATE MACRO
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DELETE
- DROP AUTHORIZATION
- DROP FUNCTION
- DROP MACRO
- DROP PROCEDURE
- DROP TABLE
- DROP TRIGGER
- DROP VIEW
- DUMP
- EXECUTE
- INSERT
- RESTORE
- SELECT

Privileges Not Granted Automatically

The following privileges on itself are not automatically granted to a user when it is created.

- ALTER EXTERNAL PROCEDURE
- ALTER FUNCTION
- ALTER PROCEDURE
- CREATE DATABASE
- CREATE EXTERNAL PROCEDURE
- CREATE FUNCTION
- CREATE PROCEDURE
- CREATE USER
- DROP DATABASE
- DROP USER
- EXECUTE FUNCTION
- EXECUTE PROCEDURE

Collation Option Usage

The COLLATION option defines the name for a collation sequence that determines the ordering of data characters during comparison operations and when sorting data in response to a SELECT query that includes an ORDER BY or WITH ... BY clause.

You can override the COLLATION attribute during any session by using a SET SESSION COLLATION statement. See [SET SESSION COLLATION](#).

The following table defines the collation sequences supported by Vantage.

Keyword	Definition
ASCII	HOST. The collation is ASCII for configurations other than an IBM mainframe-attached host. In this case, ASCII refers to the Teradata extension to the ASCII standard.
CHARSET_COLL	HOST. The collation is binary in the order of the current client character set. Strings are compared character-by-character and padded with the pad character for the appropriate character data type where necessary. When not CASE SPECIFIC, lowercase characters are mapped to their uppercase counterparts.
EBCDIC	HOST. The collation is EBCDIC for an IBM mainframe-attached host.
HOST	The collation sequence used by the logon client system. This is the default.
JIS_COLL	Japanese Industrial Standards (JIS). The collation is as follows. Characters and symbols from the JIS X 0201 standard (in JIS X 0201 order) Ideographs, characters, and symbols from the JIS X 0208 standard (in JIS X 0208 order) Ideographs, characters, and symbols from the JIS 0212 standard (in JIS X 0212 order) IBM Kanji ideographs not present in JIS X 0208 and JIS X 0212 (in KanjiEBCDIC order) User-defined ideographs (in U+ order) Any remaining characters in Unicode (in U+ order) Treatment of KANJISJIS, GRAPHIC, and LATIN character types is as if the data were first converted to Unicode and then the appropriate JIS_COLL ordering is applied.
MULTINATIONAL	One of the European or Japanese language sort sequences.

MULTINATIONAL Collation

The MULTINATIONAL keyword defines the name of the default collation sequence for the user to be one of the International sort orders. For more information, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

The MULTINATIONAL parameter determines that the International sort order is to be used. This parameter returns an error unless the hashing algorithm is set to recognize diacritical characters.

On International/European sites MULTINATIONAL collation is two-level.

On Japanese language sites, collation is single-level and is always available.

Valid Export Width Specifications

Vantage supports the following default export width specifications:

- Compatibility Mode
- Expected Mode
- Maximum Mode
- Default Mode

Or, you can create a custom export width table to handle special problems that your site has with character export widths.

Vantage always treats the export width string you specify as if it were defined as NOTCASESPECIFIC ASCII.

See *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125 for detailed information about export widths and how to create a custom export width table.

Compatibility Mode

Enables Unicode column data to be processed by applications written to process Latin or Kanji1 character data. The expected export width multiplier for this case is 1 with an adjustment of 0.

Assume that you have changed your Kanji1(*n*) columns to Unicode(*n*) columns, so you always want your exported character strings to have *n* bytes. Because of this, Compatibility mode does not add any bytes for Shift Out/Shift In support for mainframe systems.

Your Kanji1 column was already designed to store Shift Out/Shift In characters, and the corresponding new Unicode definition enables data exported to a mainframe client to have enough room to include those Shift Out and Shift In characters.

Expected Mode

Provides reasonable default widths for a character data type and client form-of-use. The expected export width multiplier for this case is 3 with an adjustment of 2.

In Expected mode, Vantage multiplies Unicode characters by 2 to calculate the number of bytes required to export Unicode strings to KanjiSJIS_OS because n Unicode characters translate to $2*n$ bytes in KanjiSJIS_OS.

For another example, suppose you want to export Unicode characters to KanjiEBCDIC5035_OI in Expected mode. In this case, Vantage multiplies the Unicode strings by 2 to determine the number of bytes to export, and, if the client is a mainframe, adds an additional 2 bytes to account for the Shift Out and Shift In characters needed for KanjiEBCDIC5035_OI to express any double byte characters if the client is a mainframe. This is because n Unicode characters are expected to take $2*n + 2$ bytes in KanjiEBCDIC5035_OI.

Maximum Mode

Provide the maximum default export width for a character data type and client form-of-use. The expected width multiplier for this case is 2 with an adjustment of 1.

Maximum mode is designed to handle cases where there can be multiple Shift Out and Shift In characters in the same string for a mainframe client. Because determining a true maximum is a complex computation, Vantage simplifies the calculation to just multiply $n * 3$ to determine the number of bytes exported based on the number of Unicode characters.

This works except for the case where the length of the character string is 1. The maximum number it could take is two bytes for a double byte character plus 1 byte each for the Shift Out and Shift In characters (assuming the client is a mainframe), making four bytes. Vantage adds 1 byte to handle this case.

DEFAULT Mode

Unlike Compatibility, Expected, and Maximum modes, which are literal strings that represent predefined export width definitions, DEFAULT is an SQL keyword that does not have a predefined meaning and is not delimited by SINGLE QUOTATION MARK characters.

If you specify DEFAULT as the Export Width for a user, Vantage assigns the export definition that is currently defined in the Export Width Table ID parameter of DBS Control to that user. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Effects of Shift Out and Shift In Characters on Export Widths

Shift Out and Shift In are control characters used to switch to a different character set and back for data exported to mainframes. Workstation-attached systems do not use the Shift Out and Shift In characters.

- Shift Out shifts to an alternative character encoding, which for Teradata is always a double-byte encoding.
- Shift In returns to the normal character encoding, which for Teradata is always a single-byte encoding.

Vantage assumes that all character strings (with the exception of CHARACTER SET GRAPHIC columns outside of Field mode) are in single-byte mode initially, and must be returned to single-byte mode before the string ends.

The following session character sets are affected significantly by Shift Out/Shift In considerations for export widths.

- KatakanaEBCDIC
- Any session character set that ends in the string '_OI'
- Any site-defined session character set with STATEMACHINE SOSIOEOF.

For more information about these character sets in the context of session export widths, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Journal Tables and the Default Map

If CREATE USER statement specifies a journal table and a default map, the journal table is created using this default map. The journal table map must be a contiguous map that has the same AMPs as TD_GlobalMap.

If CREATE USER statement specifies a journal table and without specifying a default map, the system determines the map for the journal table in the following order of precedence:

- Default map for the profile of the user creator, if defined.
- Default map for the user creator, if defined.
- System-default map.

CREATE USER Examples

Example: Creating an Administrative User

Assume that a Teradata system has just been installed. The DBA has logged on as user *DBC* and has submitted a CREATE USER request to create the administrative user called *SYSADMIN*. As user *SYSADMIN*, the DBA submits CREATE DATABASE requests to create such required databases as *finance*, *executive*, *engineering*, *manufacturing*, and *marketing*.

Still logged on as user *SYSADMIN*, the DBA then adds a new user to the *finance* database using a CREATE USER request, as follows:

```
CREATE USER marks
FROM finance AS
```

```

PERMANENT = 1000000,
PASSWORD = finan1,
SPOOL = 1200000,
FALLBACK PROTECTION,
DUAL AFTER JOURNAL,
DEFAULT JOURNAL TABLE = finance.journals,
DEFAULT DATABASE = finance,
STARTUP = 'EXEC setpf;'
ACCOUNT = '$mfinance','$hfinance';

```

After user *marks* is created, the DBA submits the following request to grant *finance* privileges WITH GRANT OPTION to *marks*.

```

GRANT ALL ON finance
TO marks
WITH GRANT OPTION;

```

Example: Using Constant Expression to Specify PERM, SPOOL, and TEMPORARY Space for User

The following request creates PERM space for a user with a size based on the constant expression 2,000,000 (HASHAMP()+1), SPOOL with a size based on the same constant expression 2,000,000 (HASHAMP()+1), and TEMPORARY with a size based on the constant expression 2,000,000 (HASHAMP()+1). The expressions calculate the number of AMPs in the current system and scales the PERM, SPOOL, and TEMPORARY space for the sls120639 user to that size.

```

CREATE USER sls120639 AS
  DEFAULT DATABASE = it_dev,
  PASSWORD = (EXPIRE = 0),
  PERM = 2000000*(HASHAMP()+1),
  SPOOL = 2000000*(HASHAMP()+1),
  TEMPORARY = 2000000*(HASHAMP()+1);

```

Example: Creating User with Permanent Space Skew Limit

On a system with 4 AMPs, you create a user with 1 gigabyte of permanent space. The per-AMP quota of permanent space is 250 megabytes. If you set the permanent space skew limit to 10%, any AMP is allowed a skew limit of 25 megabytes over the per-AMP quota of permanent space. An AMP can use up to 275 megabytes of permanent space as long as total permanent space used does not exceed the 1 gigabyte global limit for permanent space.

```
CREATE USER user1 AS
  PASSWORD = (EXPIRE = 0),
  PERM = 1e9 SKEW = 10 PERCENT;
```

Example: Creating User with Temporary Space Skew Limit

For a system with 4 AMPs, you create a user with 1 gigabyte of temporary space. The per-AMP quota of temporary space is 250 megabytes. If you set the temporary skew limit to 20%, any AMP is allowed a skew limit of 50MB above the per-AMP quota of temporary space. An AMP can use up to 300 megabytes of temporary space as long as total temporary space used does not exceed the 1 gigabyte global limit for temporary space.

```
CREATE USER user1 AS
  PASSWORD = (EXPIRE = 0),
  PERM = 1e9,
  TEMPORARY = 1e9 SKEW = 20 PERCENT;
```

Example: Creating User with Spool Space Skew Limit

On a system with 4 AMPs, you create a user with 1 gigabyte of spool space. The per-AMP quota of spool space is 250 megabytes. If you set the spool space skew limit to 20%, any AMP is allowed a skew limit of 50 megabytes over the per-AMP quota of spool space. An AMP can use up to 300 megabytes of spool space as long as total spool space used does not exceed the 1 gigabyte global limit for spool space.

```
CREATE USER user1 AS
  PASSWORD = (EXPIRE = 0),
  PERM = 1e9
  SPOOL = 1e9 SKEW = 20 PERCENT;
```

Example: Creating User Using TIME_ZONE='time_zone_string'

This example creates *user_name* with a default time zone displacement defined by the text string 'America Pacific'. When *user_name* is created, Vantage passes the time zone string to a system-defined UDF that validates it. The UDF is also called to retrieve the rules associated with the time zone string whenever *user_name* logs on.

If the value of the *time_zone_string* you specify is not valid, the system returns an error to the requestor.

```
CREATE USER user_name
  FROM r_n_d AS
  PERMANENT = 1000000,
  PASSWORD = abbub279,
```

```

SPOOL = 1200000,
FALLBACK PROTECTION,
DUAL AFTER JOURNAL,
DEFAULT JOURNAL TABLE = r_n_d.journals,
DEFAULT DATABASE = r_n_d,
STARTUP = 'EXEC setpf;'
ACCOUNT = '$mr_n_d', '$hr_n_d',
TIME ZONE = 'America Pacific';

```

Example: Create User that Includes UDT Transforms

This example creates a user that specifies the XMLD_STRUCT2INT transform group for the XMLD_STRUCT2 data type, the TD_JSON_VARCHAR transform group for the JSON CHARACTER SET LATIN data type, and the TD_JSON_VARCHAR transform group for the JSON CHARACTER SET UNICODE data type.

```

CREATE USER TEST_XFORM AS PERM=4E7 PASSWORD=TEST_XFORM TRANSFORM (XMLD_STRUCT2
= XMLD_STRUCT2INT, JSON CHARACTER SET LATIN = TD_JSON_VARCHAR, JSON CHARACTER
SET UNICODE = TD_JSON_VARCHAR);

```

Example: Creating Users With Row-Level Security Constraints

The following SQL text creates users *user_name*, *pls*, and *ArnAnderson*, each with one or more row-level security constraints.

```

CREATE USER  user_name
AS
  PERMANENT = 1e6,
  PASSWORD=my_pwd
,
  CONSTRAINT = classification_level (TopSecret),
  CONSTRAINT = classification_category (UnitedStates);
CREATE USER  pls
AS
  PERMANENT = 1e6,
  PASSWORD=secret
,
  CONSTRAINT = classification_level (Secret, Unclassified DEFAULT),
  CONSTRAINT = classification_category (UnitedStates);
CREATE USER  ArnAnderson
AS
  PERMANENT = 1e6,

```

```

        PASSWORD=hidden
    ,
        CONSTRAINT = classification_category (Norway);

```

The following set of CREATE USER requests creates a set of users are all created with the *group_membership* constraint, but each has a different *value_name* for that constraint.

User *sally_jones* is the only user who is defined as a personnel clerk.

```

CREATE USER sally_jones
AS
    PERMANENT = 1E6,
    PASSWORD=Sal3446Jones
,
    CONSTRAINT = group_membership (clerk),
    DEFAULT ROLE=personnel_clerk;

```

User *big_guy* is an executive.

```

CREATE USER big_guy
AS
    PERMANENT = 1E6,
    PASSWORD=Big9999Guy
,
    CONSTRAINT = group_membership (executive),
    DEFAULT ROLE=exec_role;

```

User *al_manager* is a manager.

```

CREATE USER al_manager
AS
    PERMANENT = 1E6,
    PASSWORD=Al9999Manager
,
    CONSTRAINT = group_membership (manager),
    DEFAULT ROLE=mgr_role;

```

User *tom_smith* is an auditor and needs to be able to read the *emprecords* table. However, he should not have any other access to the table.

```

CREATE USER tom_smith
AS
    PERMANENT = 1E6,
    PASSWORD=Tom1111Smith

```

```
,
    CONSTRAINT = group_membership (executive),
    DEFAULT ROLE=peon;
```

The following set of CREATE USER requests creates a set of users are all created with the *group_membership* constraint, but each has a different *value_name* for that constraint.

User *sally_jones* is the only user who is defined as a personnel clerk.

```
CREATE USER sally_jones
AS
    PERMANENT = 1E6,
    PASSWORD=Sal3446Jones
,
    CONSTRAINT = group_membership (clerk),
    DEFAULT ROLE=personnel_clerk;
```

User *big_guy* is an executive.

```
CREATE USER big_guy
AS
    PERMANENT = 1E6,
    PASSWORD=Big9999Guy
,
    CONSTRAINT = group_membership (executive),
    DEFAULT ROLE=exec_role;
```

User *al_manager* is a manager.

```
CREATE USER al_manager
AS
    PERMANENT = 1E6,
    PASSWORD=Al9999Manager
,
    CONSTRAINT = group_membership (manager),
    DEFAULT ROLE=mgr_role;
```

User *tom_smith* is an auditor and needs to be able to read the *emprecords* table. However, he should not have any other access to the table.

```
CREATE USER tom_smith
AS
    PERMANENT = 1E6,
    PASSWORD=Tom1111Smith
```

```
,
  CONSTRAINT = group_membership (executive),
  DEFAULT ROLE=staff;
```

Example: Creating New User With EXPECTED Export Width Table Name

This example creates a new user who is defined using the EXPECTED export width definition.

```
CREATE USER fc1
FROM r_n_d AS
PERMANENT = 1000000,
PASSWORD = bub279,
SPOOL = 2200000,
FALLBACK PROTECTION,
DUAL AFTER JOURNAL,
DEFAULT JOURNAL TABLE = r_n_d.journals,
DEFAULT DATABASE = r_n_d,
STARTUP = 'EXEC setpf;',
ACCOUNT = '$mr_n_d', '$hr_n_d',
TIME ZONE = 'Africa Egypt'
EXPORTWIDTH = EXPECTED;
```

Related Information

- *Teradata Vantage™ - Data Dictionary*, B035-1092
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100

MODIFY USER

Changes system parameters assigned to the specified user.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Other SQL dialects support similar non-ANSI standard statements with names such as ALTER USER.

Required Privileges

User Performing Modification	DROP USER Privilege Required For
Not the user being modified.	Referenced user.
User being modified.	You, if you are changing any of the following parameters:

User Performing Modification	DROP USER Privilege Required For
	<ul style="list-style-type: none"> • PERMANENT • SPOOL • ACCOUNT • PROFILE <p>To submit a MODIFY USER request that assigns or changes a profile, you must have the DROP PROFILE privilege.</p> <p>The DROP USER privilege on yourself is not required to change other parameters.</p>

To assign, change, or remove a row-level security constraint for a user, you must have the CONSTRAINT ASSIGNMENT privilege.

You cannot use MODIFY USER to modify a database. See [MODIFY DATABASE](#).

Privileges Granted Automatically

If you submit a successful MODIFY USER DEFAULT JOURNAL TABLE request, the following privileges are granted automatically to the user on the journal table.

- DROP TABLE
- INSERT

MODIFY USER Syntax

```
MODIFY USER user_name AS user_attribute [[,]...] [;]
```

user_attribute

```
{ { PERMANENT | PERM | TEMPORARY | SPOOL } = { n | constant_expression } [BYTES]
  [ SKEW { constant_expression | DEFAULT } [PERCENT]] |

STARTUP = { NULL | quotestring } |

PASSWORD = password [FOR USER] |

RELEASE PASSWORD LOCK |

ACCOUNT = { 'account_string' | ('account_string' [,...]) | NULL } |

DEFAULT MAP = { map_name | NULL } [ OVERRIDE [NOT] ON ERROR ] |

DEFAULT DATABASE = database_name |
```



```

COLLATION = collation_sequence |

[NO] FALLBACK [PROTECTION] |

[ NO | DUAL ] [BEFORE] JOURNAL |

[ NO | DUAL | [NOT] LOCAL ] AFTER JOURNAL |

DEFAULT JOURNAL TABLE = [ database_name. ] table_name |

DROP DEFAULT JOURNAL TABLE [ = table_name ] |

TIME_ZONE = { LOCAL | [sign] 'quotestring' | 'time_zone_string' |
NULL } |

DATEFORM = { INTEGERDATE | ANSIDATE | NULL } |

DEFAULT CHARACTER SET server_character_set |

DEFAULT ROLE = { role_name | NONE | NULL | ALL } |

PROFILE = { profile_name | NULL } |

TRANSFORM ( [ transform_specification [,...] ] ) |

[NOT] DBA |

EXPORTWIDTH { 'export_definition_name' | DEFAULT } |

CONSTRAINT = constraint [,...]
}

```

transform_specification

```
data_type = group_name
```

constraint

```

row_level_security_constraint_column_name {
  ( { level_name [DEFAULT] | category_name } [,...]) |

```

```
(NULL)
}
```

MODIFY USER Syntax Elements

user_name

Name of the user to be modified.

PERMANENT

A revised value for PERM space allocation for the specified user, in bytes.

n

You can enter the number of bytes as an integer, decimal, or floating point value or as a constant expression whose evaluation determines the number of bytes. You can also enter the value using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

The value of *n* or the number of bytes determined from the evaluation of *constant_expression* cannot exceed the permanent space of the owner.

constant_expression

A constant expression is any SQL expression that does not make any column references. Specifying an appropriate constant expression for the PERM space size of a database enables Vantage to assign an optimal quantity of PERM space that scales to the size of your system by allocating PERM space on a per AMP basis.

When you specify a PERM space size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

Optional keyword that redundantly specifies the unit for the amount of space allowed.

SKEW

Keyword that you use to specify a skew limit for PERMANENT space. You can specify a skew limit percentage that allows the maximum AMP space usage to be above the per-AMP quota, that is, the system maximum space limit divided by the number of AMPs.

constant_expression

Constant expression or constant from 0 through 10000. Specify a value from 1 to 9999 to indicate an AMP-level limit, which is the per-AMP quota * $(1 + \text{permskewlimit}/100)$. Specify 0 to set space accounting to the per-AMP level, that is, no skew. A value of 10000 indicates unlimited skew, up to the system maximum space limit.

DEFAULT

Use the value of DBS Control DefaultPermSkewLimitPercent.

PERCENT

Optional keyword that you can include for readability to indicate that the *constant_expression* or DEFAULT keyword specifies a percentage of allowable skew.

STARTUP

The startup string is performed either when you log on to Vantage through BTEQ or when you establish a JDBC connection using the Teradata JDBC driver and have specified the connection parameter RUNSTARTUP=ON. BTEQ and JDBC are the only client applications that support startup strings.

You can include a SET SESSION CALENDAR request in a startup string to set the calendar for a user to something other than the default Teradata calendar. The other available system-defined calendars are named ISO and COMPATIBLE. For more information, see SET SESSION CALENDAR in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

quotestring

One or more SQL requests, separated by SEMICOLON characters, that are performed to establish the session environment when the user logs on.

NULL

Set the startup string to NULL, that is, no startup string.

PASSWORD

New password for the specified user.

All users have the MODIFY USER privilege on their own user definition. This enables users to:

- Create a private password at the system prompt after attempting a first logon with a temporary password. For details, see the explanation of the PASSWORD keyword in [CREATE USER](#).
- Create a new password when their existing user password expires.

password

Passwords must follow required formatting rules. See *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

FOR USER

[Optional] Removes the existing password for the specified user immediately and sets the password expiration date to zero, depending on the value of the Expire option in the profile assigned to the user.

If user is not assigned a profile or if the Expire option in the assigned profile is not set, then the expiration date is set to the value of the Expire Password column specified in DBC.SysSecDefaults as shown in the following table.

- If the Expire value in the user profile or the value for ExpirePassword in DBC.SysSecDefaults is 0, then the current password expires immediately and the new temporary password becomes valid indefinitely.

NOTICE
This is a potential security breach. You must ensure that it does not happen.

- If the Expire value in the user profile or the value for ExpirePassword in DBC.SysSecDefaults is > 0, then the current password expires immediately and the new temporary password becomes valid.

The value for ExpirePassword is reset to 0.

The new temporary password expires at first logon, at which time the user must create a new, permanent password.

The password attribute settings in a user profile override the corresponding system-level password settings.

For more information about DBC.SysSecDefaults and ExpirePassword, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100 and *Teradata Vantage™ - Data Dictionary*, B035-1092. For more information about profiles, see [CREATE ROLE](#).

RELEASE PASSWORD LOCK

Keywords to release a user lock.

A user row in DBC.Dbbase becomes locked if the number of successive erroneous logon attempts exceeds the current user default specified in the MaxLogonAttempts password control attribute.

The user row remains locked to logon attempts until the site-defined time has passed, or until a user with MODIFY privileges on the locked user submits a MODIFY USER request to release the lock.

TEMPORARY

Temporary space reserved for creating materialized global temporary tables by the specified user.

Temporary space is reserved prior to spool space for any user defined with this characteristic.

Disk usage for a materialized global temporary table is charged to the user who referenced the table.

n

Number of bytes allowed for creating global temporary tables by this user.

constant_expression

Any SQL expression that does not make any column references. Specifying an appropriate constant expression for the TEMPORARY space size of a user enables Vantage to assign an optimal quantity of TEMPORARY space that scales to the size of your system by allocating TEMPORARY space on a per AMP basis.

When you specify a TEMPORARY space size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

Optional keyword that redundantly specifies the unit for the amount of space allowed.

SKEW

Keyword that you use to specify a skew limit for TEMPORARY space. You can specify a skew limit percentage that allows the maximum AMP space usage to be above the per-AMP quota, that is, the system maximum space limit divided by the number of AMPs.

constant_expression

Constant expression or constant from 0 through 10000. Specify a value from 1 to 9999 to indicate an AMP-level limit, which is the per-AMP quota * (1+*temp skew limit*/100). Specify 0 to set space accounting to the per-AMP level, that is, no skew. A value of 10000 indicates unlimited skew, up to the system maximum space limit.

DEFAULT

Use the value of DBS Control DefaultTempSkewLimitPercent.

PERCENT

Optional keyword that you can include for readability to indicate that the *constant_expression* or DEFAULT keyword specifies a percentage of allowable skew.

SPOOL

Maximum space in bytes allowed for spool files for the specified user.

n

You can enter the number of bytes as an integer, decimal, or floating point value or as a constant expression whose evaluation determines the number of bytes. You can also enter the value using exponential notation. For example, you can write one thousand as either 1000 or 1E3.

n cannot exceed the spool space parameter in the profile of the creator. If no spool space is defined for that profile, then Vantage uses the spool space limit defined for the individual user-creator.

constant_expression

A constant expression is any SQL expression that does not make any column references. Specifying an appropriate constant expression for the SPOOL space size of a user enables Vantage to assign an optimal quantity of spool that scales to the size of your system by allocating SPOOL space on a per AMP basis.

When you specify a SPOOL space size based on a constant expression, the assigned size does not automatically scale if, for example, you add more AMPs to your system.

BYTES

Optional keyword that redundantly specifies the unit for the amount of space allowed.

SKEW

Keyword that you use to specify a skew limit for SPOOL space. You can specify a skew limit percentage that allows the maximum AMP space usage to be above the per-AMP quota, which is the system maximum space limit divided by the number of AMPs.

constant_expression

Constant expression or constant from 0 through 10000. Specify a value from 1 to 9999 to indicate an AMP-level limit, which is the per-AMP quota * (1+*spoolskewlimit*/100). Specify 0 to set space accounting to the per-AMP level, that is, no skew. A value of 10000 indicates unlimited skew, up to the system maximum space limit.

DEFAULT

Use the value of DBS Control DefaultSpoolSkewLimitPercent.

PERCENT

Optional keyword that you can include for readability to indicate that the *constant_expression* or DEFAULT keyword specifies a percentage of allowable skew.

ACCOUNT

One or more accounts to be added to any accounts that already exist in the user definition.

The first account string you specify becomes the default account for the user, but you can specify multiple accounts if needed.

Users with multiple accounts can access a non-default account by specifying the account in a:

- Logon string.
- SET SESSION ACCOUNT statement.

For a user to specify an account in the logon string or in a SET SESSION ACCOUNT statement, the account must be assigned in the user or profile definition for the user.

Accounts do not exist independently in the database. Accounts only exist as specified in user, database, and profile definitions. You can specify the same account string in multiple object definitions, if needed.

Account strings defined in the user profile supersede the accounts defined in the user definition.

account_string

Each position in an account string has a specific meaning and format requirement. See *Teradata Vantage™ - Database Administration*, B035-1093. Each *account_string* must follow the Vantage object naming rules. See *Teradata Vantage™ - SQL Fundamentals*, B035-1141. You must enclose each account string with APOSTROPHE characters. You must separate each entry in a list of account strings with a COMMA character and enclose the list with LEFT PARENTHESIS and RIGHT PARENTHESIS characters.

If you do not specify an account string for the user or the user profile, the user inherits the first account string that is defined for the immediate owner of the user.

DEFAULT MAP

You can specify an existing contiguous or sparse map as the default map for the user. You can also remove a default map that is defined for the user by setting the default map to null.

You must have been granted the specified map unless the map is the same as the default map of the database modifier.

map_name

Name of an existing contiguous or sparse map.

You cannot specify TD_DataDictionaryMap or TD_GlobalMap.

To specify a sparse map as the default map, you must be in the same secure zone as the sparse map and the sparse map must be in the same secure zone as the database.

OVERRIDE ON ERROR

Use the default map if an error occurs when the MAP clause is specified for a CREATE TABLE, CREATE JOIN INDEX, or CREATE HASH INDEX statement. For example, if the MAP clause specifies a nonexistent contiguous map or a sparse map in another secure zone, a warning message displays. The message indicates that the specified map was not used and lists the name of the default map used instead.

This is the default.

OVERRIDE NOT ON ERROR

Do not use the default map if an error occurs when the MAP clause is specified for a CREATE TABLE, CREATE JOIN INDEX, or CREATE HASH INDEX statement.

NULL

A default map is not associated with the user. You can specify NULL to remove the default map that is currently set for the user.

DEFAULT DATABASE

Default database established each time this user logs onto Vantage.

The default is the user name associated with the current session.

database_name

Name of the default database.

database_name need not be an existing database. However, Vantage returns an error when you try to create or reference an object within a nonexistent database or if the default database is null.

COLLATION

The default collation sequence for the specified user.

The maximum length of *collation_sequence* is 30 characters.

The COLLATION option defines the name for a collation sequence that determines the ordering of data characters during comparison operations and when sorting data in response to a SELECT query that includes an ORDER BY or WITH ... BY clause.

You can override the COLLATION attribute during any session by running an appropriate SET SESSION COLLATION request. See [SET SESSION COLLATION](#).

ASCII

The collation is ASCII for configurations other than an IBM mainframe-attached host.

In this case, ASCII refers to the Teradata extension to the ASCII standard.

CHARSET_COLL

The collation is binary in the order of the current client character set.

Strings are compared character-by-character and padded with the pad character for the appropriate character data type where necessary.

When not CASE SPECIFIC, lowercase characters are mapped to their uppercase counterparts.

EBCDIC

The collation is EBCDIC for an IBM mainframe-attached host.

HOST

The collation sequence used by the logon client system.

This is the default.

JIS_COLL

Japanese Industrial Standards (JIS).

The collation is as follows.

Characters and symbols from the JIS X 0201 standard (in JIS X 0201 order)

Ideographs, characters, and symbols from the JIS X 0208 standard (in JIS X 0208 order)

Ideographs, characters, and symbols from the JIS 0212 standard (in JIS X 0212 order)

IBM Kanji ideographs not present in JIS X 0208 and JIS X 0212 (in KanjiEBCDIC order)

User-defined ideographs (in U+ order)

Any remaining characters in Unicode (in U+ order)

KANJISJIS, GRAPHIC, and LATIN character types are treated as if the data were first converted to Unicode and then the appropriate JIS_COLL ordering is applied.

MULTINATIONAL

One of the European or Japanese language sort sequences.

The MULTINATIONAL keyword defines the name of the default collation sequence for the user to be one of the International sort orders (see SQL Data Manipulation Language).

The MULTINATIONAL parameter determines that the International sort order is to be used. This parameter returns an error unless the hashing algorithm is set to recognize diacritical characters.

On International/European sites MULTINATIONAL collation is two-level. On Japanese language sites, collation is single-level and is always available.

FALLBACK

Default for duplicate copy protection of each data table subsequently created in the database. The current fallback setting for existing data tables remains unchanged.

PROTECTION

Optional keyword that you can specify with FALLBACK. The FALLBACK keyword used alone implies PROTECTION.

NO

Sets the default to not provide duplicate copy protection for data tables created in the database.

Note:

You cannot use the NO FALLBACK option and the NO FALLBACK default on platforms optimized for fallback.

JOURNAL

The JOURNAL keyword without AFTER or BEFORE indicates that both types of images are to be maintained. In this case, the current default for either or both types is modified accordingly.

For example, if only AFTER JOURNAL is specified, the current default for before change images remains in effect.

If the JOURNAL keyword is specified without NO or DUAL, single copy journaling is implied. If journaling is specified, a DUAL journal is maintained for data tables with FALLBACK protection.

Existing images are not affected until the corresponding table is updated.

This option can appear twice in the same request: once to specify a BEFORE or AFTER image, and again to specify the alternate type.

If only one type is specified, then the current default is modified only for that type.

If BEFORE and AFTER are specified, the two must not conflict.

Journal options are not supported for tables with row sizes greater than 64KB.

BEFORE

A new default for the number of before change images to be maintained for data tables subsequently created in the database by the specified user.

DUAL

Dual journal images are maintained for the table.

NO

Terminates any current journaling default.

AFTER JOURNAL

A change to the type of image to be maintained for the table. Any existing images are not affected until the table is updated.

NO

After-change images are not maintained for tables created by the specified user.

DUAL

Dual after-change images are maintained for tables created by the specified user.

LOCAL

Single after-image journal rows for non-fallback data tables created by the specified user are written on the same virtual AMP as the changed data rows.

NOT LOCAL

Single after-image journal rows for non-fallback data tables created by the specified user are written on another virtual AMP in the cluster.

DEFAULT JOURNAL TABLE

The current journal table or removes its status as the default for the specified user.

table_name

Name of table.

database_name

Containing database.

DROP DEFAULT JOURNAL TABLE

Removal of the default status of the journal table currently defined as the default for the specified user.

If the journal table is contained by the user being modified, DROP also drops the table from the system. An error message is returned if the journal table is being used by active data tables.

Specifying this option does not change the status of existing data tables for the modified user.

table_name

The *table_name* parameter is required if this clause is specified without the DROP keyword.

database_name

If you do not specify a database, then Vantage assumes the current database by default.

If the database being modified does not have a journal table, *table_name* is created.

If you specify a different database, then it must already exist and *table_name* must already be defined as its default journal table.

user_name

If you do not specify a user, then Vantage assumes the current user by default.

If the user being modified does not have a journal table, *table_name* is created.

If you specify a different user, then it must already exist and *table_name* must already be defined as its default journal table.

TIME ZONE

Default time zone displacement for the specified user.

For descriptions of these options, see MODIFY USER in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

You can specify:

- LOCAL
- NULL
- \pm 'quotestring'
- 'time_zone_string'

DATEFORM

Default format for importing and exporting DATE values for the specified user.

For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

INTEGERDATE

Sets the DATEFORM option to import and export DATE values as encoded integers.

INTEGERDATE results in a default DATE format in field mode of 'YY/MM/DD' for date columns created and for date constants in character form. You can use the `tdlocaledef` utility to change the 'YY/MM/DD' default DATE format. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

INTEGERDATE is the default.

ANSIDATE

Sets the DATEFORM option to import and export DATE values as CHARACTER(10).

Results in a 'YYYY-MM-DD' date format for date columns created and for date constants in character form.

NULL

Sets DATEFORM null.

When DATEFORM is null, the default format for a date in field mode, and the default format for importing and exporting DATE values, is the same as the system default defined by the DATEFORM parameter in the DBS Control record.

DEFAULT CHARACTER SET

Server character set for the specified user.

For details about server character set options, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

If you change the default character set for a user, you must recompile any UDFs the user previously created.

All changes affect only those columns created after the user definition has been modified. Columns created previously by this user retain their original server character set.

If you do not assign a default server character set, Vantage assigns one automatically using the default character set it retrieves from the DBS Control parameter `DefaultCharacterSet`.

server_character_set

Most systems can use either:

- LATIN
- UNICODE (contains all languages)

You cannot specify a server character set of KANJI1. Otherwise, the system returns an error to the requestor.

DEFAULT ROLE

The assignment of a default role provides a user with an enabled role at logon. If you do not assign a default role to the user, or if the user wants to enable a different assigned role, the user must submit a SET ROLE request to enable the needed role for a session. See [SET ROLE](#).

role_name

Name of a role (which may include nested roles) to assign as the default role for the user, replacing the previously existing default role if it exists. The specified role must already be granted to the user.

ALL

Indicates that all roles currently granted to the user, including any nested roles, are enabled by default.

NONE

No default role is assigned to the user.

NULL

Removes any previous default role assignment and does not assign a default role to the user.

PROFILE

Profile for the specified user. Replaces any previous profile specification.

Effects of changing the user profile assignment:

- Password attribute settings in the new profile take effect the next time the user logs on.
- Spool and temporary space settings in the new profile take effect immediately.
- The current account and database settings in the new profile do not take effect until the next time the user logs on or the user uses the SET SESSION statement to explicitly change them.
- The password for the modifying user must be compliant with any password control rules specified in the new profile being assigned to the modified user. Some password controls may only affect the children of the user being modified. See *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.
- Query band settings in the new profile take effect on the next request in the session.

profile_name

The name of the profile assigned to the user. If a profile is already assigned to the user, the newly specified profile replaces the existing profile.

NULL

No profile is assigned to the user. Any existing profile assignment is removed.

TRANSFORM

Adds UDT transform groups to the user or removes UDT transform groups from the user.

You can specify up to 256 *data_type-transform_group* pairs for a user.

If the user has a profile, the transform group for the UDT in the user's profile is used.

You can only add, replace, or remove entire transform group settings. You cannot add or remove individual *data_type-transform_group* pairs.

After transform settings are changed, the user must log out and log in again to use the new settings.

data_type

Name of the user-defined data type (UDT).

group_name

Name of transform group for the data type.

DBA

Database Administrator (DBA) option.

DBA users can perform DML operations on objects they create in their own user-space.

DBA users can perform DML operations on objects created by other users if the DBA user has been granted the required discretionary access control (DAC) privileges.

DBA users cannot perform DML operations on objects they create in another user's space.

NOT

Removes the DBA option from the user.

EXPORTWIDTH

Vantage uses the value you assign to a user as a NUPI to access DBC.ExportWidth, which defines the rules associated with the different export width definitions for Latin, Unicode, KanjiSJIS, and

Graphic strings. For more information about DBC.ExportWidth, see *Teradata Vantage™ - Data Dictionary*, B035-1092.

export_definition_name

Export width for this user, enclosed in SINGLE QUOTATION MARK characters.

Vantage supports the keyword DEFAULT as an export width definition string as well as the following predefined export width definition strings.

- COMPATIBILITY
- EXPECTED
- MAXIMUM

For a detailed description of the valid export width specifications, see [Valid Export Width Specifications](#).

For detailed information about export widths and how to define your own export width table, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

DEFAULT

If you do not specify an EXPORTWIDTH value for a user or specify the keyword DEFAULT, Vantage uses the export width currently specified by the DBSControl Export Width Table ID parameter. See Export Width Table ID in *Teradata Vantage™ - Database Utilities*, B035-1102.

CONSTRAINT

There is a limit of 6 hierarchical constraints and 2 non-hierarchical constraints that can be assigned per profile.

If the constraint is already assigned to the user, the level or category names you specify for the constraint replace all existing specifications.

Changes take effect at the next user logon after the user definition is modified.

For more information about row-level security constraints, see [CREATE CONSTRAINT](#).

row_level_security_constraint_name

Name of an existing row-level security constraint followed by the list of the hierarchical levels or non-hierarchical categories, valid for the constraint, which are being assigned to the *user_name*.

level_name

List of the hierarchical levels.

category_name

List of non-hierarchical categories

DEFAULT

DEFAULT can be specified for 1 classification level from the specified list of levels for a hierarchical constraint. The specified level becomes the default value for the constraint when a session is established for the user to which the profile is assigned.

If you do not specify DEFAULT for any of the listed classification levels, then the first level listed becomes the default for the profile.

DEFAULT is not valid for non-hierarchical constraints.

NULL

If you specify NULL for a constraint previously assigned to the user, the constraint is dropped from the user definition.

Usage Notes

Valid Export Width Specifications

Vantage supports the following default export width specifications:

- Compatibility Mode
- Expected Mode
- Maximum Mode
- Default Mode

Or, you can create a custom export width table to handle special problems that your site has with character export widths.

For detailed information about export widths and how to create a custom export width table, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Compatibility Mode

Enable Unicode column data to be processed by applications written to process Latin or Kanji1 character data. The expected export width multiplier for this case is 1 with an adjustment of 0.

Assume that you have changed your Kanji1(*n*) columns to Unicode(*n*) columns, so you always want your exported character strings to have *n* bytes. Because of this, Compatibility mode does not add any bytes for Shift Out/Shift In support for mainframe systems.

Your Kanji1 column was already designed to store Shift Out/Shift In characters, and the corresponding new Unicode definition enables data exported to a mainframe client to have enough room to include those Shift Out and Shift In characters.

Expected Mode

Provide reasonable default widths for a character data type and client form-of-use. The expected export width multiplier for this case is 3 with an adjustment of 2.

In Expected mode, Vantage multiplies Unicode characters by 2 to calculate the number of bytes required to export Unicode strings to KanjiSJIS_0S because n Unicode characters translate to $2*n$ bytes in KanjiSJIS_0S.

For another example, suppose you want to export Unicode characters to KanjiEBCDIC5035_0I in Expected mode. In this case, Vantage multiplies the Unicode strings by 2 to determine the number of bytes to export, and, if the client is a mainframe, adds an additional 2 bytes to account for the Shift Out and Shift In characters needed for KanjiEBCDIC5035_0I to express any double byte characters if the client is a mainframe. This is because n Unicode characters are expected to take $2*n + 2$ bytes in KanjiEBCDIC5035_0I.

Maximum Mode

Provide the maximum default export width for a character data type and client form-of-use. The expected width multiplier for this case is 2 with an adjustment of 1.

Maximum mode is designed to handle cases where there can be multiple Shift Out and Shift In characters in the same string for a mainframe client. Because determining a true maximum is a complex computation, Vantage simplifies the calculation to just multiply $n * 3$ to determine the number of bytes exported based on the number of Unicode characters.

This works except for the case where the length of the character string is 1. The maximum number it could take is two bytes for a double byte character plus 1 byte each for the Shift Out and Shift In characters (assuming the client is a mainframe), making four bytes. Vantage adds 1 byte to handle this case.

DEFAULT Mode

Unlike Compatibility, Expected, and Maximum modes, which are literal strings that represent predefined export width definitions, DEFAULT is an SQL keyword that does not have a predefined meaning and is not delimited by SINGLE QUOTATION MARK characters.

If you specify DEFAULT as the Export Width for a user, Vantage assigns the export definition that is currently defined in the Export Width Table ID parameter of DBS Control to that user (see *Teradata Vantage™ - Database Utilities*, B035-1102 for further information).

Effects of Shift Out and Shift In Characters on Export Widths

Shift Out and Shift In are control characters used to switch to a different character set and back for data exported to mainframes. Workstation-attached systems do not use the Shift Out and Shift In characters.

- Shift Out shifts to an alternative character encoding, which for Teradata is always a double-byte encoding.
- Shift In returns to the normal character encoding, which for Teradata is always a single-byte encoding.

Vantage assumes that all character strings (with the exception of CHARACTER SET GRAPHIC columns outside of Field mode) are in single-byte mode initially, and must be returned to single-byte mode before the string ends.

The following session character sets are affected significantly by Shift Out/Shift In considerations for export widths.

- KatakanaEBCDIC
- Any session character set that ends in the string '_OI'
- Any site-defined session character set with STATEMACHINE SOSI0E0F.

For more information about these character sets in the context of session export widths, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Journal Tables and the Default Map

Modifying the default map of the user does not change the map being used for the journal table.

If the MODIFY USER statement specifies a journal table and a default map, the journal table is created using this default map. The journal table map must be a contiguous map with the same AMPs as TD_GlobalMap.

If the MODIFY USER statement specifies a journal table without specifying a default map, the system determines the map for the journal table in the following order of precedence:

- Default map, if defined, for the profile of the database creator. See the DEFAULT MAP option of CREATE PROFILE.
- Default map, if defined, for the database creator. See the DEFAULT MAP option of CREATE USER.
- System-default map.

MODIFY USER Examples

Example: Modifying Permanent Space Allocation for User

This statement changes the permanent space allocation for user peterson.

```
MODIFY USER peterson AS
PERMANENT = 6000000 BYTES;
```

Example: Modifying PERM, SPOOL, and TEMPORARY Space Using Constant Expression

This example creates PERM space for a user with a size based on the constant expression 2,000,000 (HASHAMP()+1), SPOOL space with a size based on the same constant expression, 2,000,000 (HASHAMP()+1), and TEMPORARY space with a size based on the constant expression 2,000,000 (HASHAMP()+1). The expressions calculate the number of AMPs in the current system and scale the PERM, SPOOL, and TEMPORARY space for the sls120639 user to that size.

This is the original definition of user sls120639.

```
CREATE USER sls120639 AS
  DEFAULT DATABASE = it_dev,
  PASSWORD = (EXPIRE = 0),
  PERM = 2000000*(HASHAMP()+1),
  SPOOL = 2000000*(HASHAMP()+1),
  TEMPORARY = 2000000*(HASHAMP()+1);
```

The following MODIFY USER statement changes the PERM, SPOOL, and TEMPORARY space allocations for this user as follows.

```
MODIFY USER sls120639 AS
  DEFAULT DATABASE = it_dev,
  PASSWORD = (EXPIRE = 0),
  PERM = 3000000*(HASHAMP()+1),
  SPOOL = 3000000*(HASHAMP()+1),
  TEMPORARY = 3000000*(HASHAMP()+1);
```

Example: Modifying User to Add Permanent Space Skew Limit

Assume a system with 4 AMPs and a user with 1 gigabyte of permanent space. The per-AMP quota of permanent space is 250 megabytes. If you set the permanent space skew limit to 10%, any AMP is allowed a skew limit of 25 megabytes over the per-AMP quota of permanent space. An AMP can use up to 275 megabytes of permanent space as long as total permanent space used does not exceed the 1 gigabyte global limit for permanent space.

```
MODIFY USER user1 AS
  PASSWORD = (EXPIRE = 0),
  PERM = 1e9 SKEW = 10 PERCENT;
```

Example: Modifying Password, Spool Allocation, Startup String, and Default Database

You do not need access privileges to change your own PASSWORD, STARTUP, FALLBACK, or DEFAULT DATABASE definitions. Otherwise, the user submitting a MODIFY request must have the DROP privilege on the user being modified.

For example, although user Marks is not the creator of his own space and does not own himself, he can submit a MODIFY USER request to change his password, modify his startup string, or redirect his default database. Marks can resize *finance*, as well.

Change the password and other options on user *marks*.

```
MODIFY USER marks AS
PASSWORD = design,
SPOOL = 1500000,
STARTUP = 'EXEC paystat;',
DEFAULT DATABASE = payroll;
```

Example: Modifying User to Add Temporary Space Skew Limit

Assume a system with 4 AMPs and a user with 1 gigabyte of temporary space. The per-AMP quota of temporary space is 250 megabytes. If you set the temporary skew limit to 20%, any AMP is allowed a skew limit of 50MB above the per-AMP quota of temporary space. An AMP can use up to 300 megabytes of temporary space as long as total temporary space used does not exceed the 1 gigabyte global limit for temporary space.

```
MODIFY USER user1 AS
PASSWORD = (EXPIRE = 0),
PERM = 1e9,
TEMPORARY = 1e9 SKEW = 20 PERCENT;
```

Example: Modifying User to Add Spool Space Skew Limit

Assume a system with 4 AMPs and a user with 1 gigabyte of spool space. The per-AMP quota of spool space is 250 megabytes. If you set the spool space skew limit to 20%, any AMP is allowed a skew limit of 50 megabytes over the per-AMP quota of spool space. An AMP can use up to 300 megabytes of spool space as long as total spool space used does not exceed the 1 gigabyte global limit for spool space.

```
MODIFY USER user1 AS
PASSWORD = (EXPIRE = 0),
```

```

    PERM = 1e9
    SPOOL = 1e9 SKEW = 20 PERCENT;

```

Example: Modifying Default Database and Password for User

Change user the default database and password for *chin*.

```

MODIFY USER chin AS
DEFAULT DATABASE = personnel,
PASSWORD = nitram;

```

Example: Modifying Default Journal Table for User

Two MODIFY requests are needed to change the default journal table if it resides in the user being modified.

For example, suppose the journal table *fincopy* is contained by user *jones*. To change the default journal table from *fincopy* to *jrn1*, the present default journal table must be dropped and then a new journal table must be created.

The first request removes *fincopy* as the default and drops it from the system. If there are any existing tables that use it as their journal table, the request returns an error.

```

MODIFY USER jones AS
DROP DEFAULT JOURNAL TABLE;

```

After the current journal table has been dropped, the second request creates a new default journal table.

```

MODIFY USER jones AS
DEFAULT JOURNAL TABLE = jrn1;

```

Example: Changing Default Journal Table

If the current journal table does not reside in the hierarchy beneath the user being modified, the following request could be used to change the default journal table from *fincopy* to *jrn1*.

```

MODIFY USER peterson AS
DEFAULT JOURNAL TABLE = jrn1;

```

Example: Modifying Time Zone Displacement with TIME ZONE LOCAL Option

This example changes the time zone displacement for *user_name* from whatever it had been to the time zone defined as the system default.

```
MODIFY USER user_name AS
TIME ZONE = LOCAL;
```

Example: Modifying Time Zone Displacement with TIME ZONE *time_zone_string* Option

This statement changes the time zone displacement for user *pa* from whatever it had been to the time zone displacement defined by the string 'America Pacific' using the TIME ZONE = '*time_zone_string*' option.

Vantage passes the specified time zone string to a system-defined UDF that validates it. The UDF is also called to retrieve the rules associated with the time zone string whenever *pa* logs on.

```
MODIFY USER pa AS
TIME ZONE = 'America Pacific';
```

If the value of the *time_zone_string* you specify is not valid, Vantage returns an error to the requestor.

Example: Assigning Profile to User with PROFILE Option

This statement creates a profile called *human_resources* that defines spool space and default database system parameters.

```
CREATE PROFILE human_resources AS
SPOOL = 1500000,
DEFAULT DATABASE = personnel;
```

This statement assigns the *human_resources* profile to user *ortega*.

```
MODIFY USER ortega AS
PROFILE = human_resources;
```

Example: Adding UDT Transform Groups to User

This example modifies a user to specify the XMLD_STRUCT2INT transform group for the XMLD_STRUCT2 data type, the TD_JSON_VARCHAR transform group for the JSON CHARACTER SET LATIN data type, and the TD_JSON_VARCHAR transform group for the JSON CHARACTER SET UNICODE data type.

```
MODIFY USER TEST_XFORM AS TRANSFORM (XMLD_STRUCT2 = XMLD_STRUCT2INT,
JSON CHARACTER SET LATIN = TD_JSON_VARCHAR, JSON CHARACTER SET UNICODE
= TD_JSON_VARCHAR);
```

Example: Removing UDT Transform Groups from User

This example removes the UDT transform groups from the user.

```
MODIFY USER TEST_XFORM AS TRANSFORM ();
```

Example: Adding or Dropping Security Constraint Assignments in MODIFY USER Statement

You can change security constraint assignments for a user in a MODIFY USER statement. When you specify a security constraint that is:

- Not currently assigned to the user, the constraint is added to any existing constraints.
- Already assigned to the user, the new specifications replace the existing specifications.
- Already assigned to the user, followed by the keyword NULL, the constraint assignment is dropped from the user.

A new or changed security constraint assignment takes effect at the next user login.

Example

```
MODIFY USER
  Joe_Smith AS
  CONSTRAINT = Classification_Level (TopSecret, Unclassified DEFAULT),
  CONSTRAINT = Classification_Country (NULL)
  CONSTRAINT = Classification_Job (Analyst);
```


Classification_Level (TopSecret, Unclassified DEFAULT)

Raises the user classification level by replacing a previously assigned Secret clearance with TopSecret, while retaining the DEFAULT Unclassified level.

Classification_Country (NULL)

Drops the Classification_Country constraint assignment that was previously assigned to the user.

Classification_Job (Analyst)

Adds the new hierarchical category Classification_Job, and assigns the Analyst classification level.

Related Information

- [CREATE USER](#)
- [DROP USER](#)
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

SET ROLE

Sets the current role for a session. It does not distinguish between directory- and database-managed roles.

You cannot use SET ROLE in a procedure definition.

SET ROLE ALL enables all roles granted to the user submitting the SET ROLE statement, including nested roles.

Use of SET ROLE EXTERNAL requires that the user is a directory user and is mapped to the external role object in the directory. For details, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

The use of SET ROLE is not permitted in a trusted session. The active roles for trusted sessions must be defined using the SET QUERY_BAND request. See [SET QUERY_BAND](#).

ANSI Compliance

SET ROLE is ANSI SQL:2011-compliant, except for the NULL, EXTERNAL, and ALL options, which are Teradata extensions.

Required Privileges

Statement	Result
SET ROLE NONE SET ROLE NULL	No privilege is needed to disable the current role for a session.
SET ROLE <i>role_name</i>	The role must already be granted to you.

Statement	Result
SET ROLE ALL	No privilege is needed and you are not required to have any roles granted to you.
SET ROLE EXTERNAL	At least one role must already be assigned (mapped) to you in the directory. See <i>Security Administration</i> .

SET ROLE Syntax

```
SET ROLE { role_name | EXTERNAL | NONE | NULL | ALL } [;]
```

SET ROLE Syntax Elements

role_name

Name of the role to set as the current role for a session.

You can use the SET ROLE *role_name* syntax to enable an externally-assigned role plus any of its nested database roles within a session while disabling all other roles.

EXTERNAL

All external roles mapped to the user are to be made active for the session.

The EXTERNAL keyword is useful only for directory users who are mapped to both a user object and one or more external role objects in the directory, where the system defaults to the privileges for the mapped user.

If the directory user is mapped only to external role objects and not mapped to a database user object, all the external roles are active by default.

NONE

Disables the current role, whether directory- or database-assigned, for a session.

NULL

Null role is to be made current for the session.

ALL

All the roles that have been directly and indirectly granted to a user, whether directory- or database-assigned, become current and active.

SET ROLE Examples

Example: Changing Current Role

This request changes the current role for a session to administration.

```
SET ROLE administration;
```

Example: Enabling All Roles

This request specifies that all roles and their nested database roles that have been granted to the user become current and active for a session.

```
SET ROLE ALL;
```

Example: Disabling Current Role SQL Data Control Language

These requests disable the current role for a session.

```
SET ROLE NONE;
SET ROLE NULL;
```

Example: Enabling EXTERNAL Role

This request resets the current set of roles for the session to the set of all directory roles assigned (mapped) to the external user making the request.

```
SET ROLE EXTERNAL;
```

Related Information

- GRANT (Role Form) in *Teradata Vantage™ - SQL Data Control Language*, B035-1149
- GRANT (SQL Form) in *Teradata Vantage™ - SQL Data Control Language*, B035-1149
- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100

DELETE USER

Deletes all data tables, views, triggers, SQL procedures, macros, and user-installed files (UIFs) from a user.

This statement does not delete the definition for a user from the dictionary. See [DROP USER](#).

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the DROP privilege on the specified user.

DELETE USER Syntax

```
{ DELETE | DEL } user_name [ALL] [;]
```

DELETE USER Syntax Elements

user_name

Name of the user from which all database objects are to be deleted.

ALL

Specifies that all objects, including the materialized global temporary tables contained by the specified user, are to be dropped.

If you do not specify ALL and the specified user contains materialized global temporary tables, Vantage aborts the request.

Example: Deleting Database Objects from User

This request can be used to delete all tables, views, triggers, SQL procedures, user-defined functions, and macros from the user named *user_name*.

```
DELETE USER user_name;
```

Related Information

- [CREATE ERROR TABLE](#)
- [CREATE TABLE and CREATE TABLE AS](#)
- [CREATE TABLE \(Queue Table Form\)](#)
- [DROP USER](#)
- *Teradata Vantage™ - Data Dictionary*, B035-1092

DROP PROFILE

Drops a specified profile.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

The following users can drop a profile.

- User *DBC*.
- A user who has the DROP PROFILE privilege.

The profile creator does not have the implicit privilege to drop a profile. A profile creator who does not have the DROP PROFILE privilege cannot drop the profile.

DROP PROFILE Syntax

```
DROP PROFILE profile_name [;]
```

Usage Notes

Effects of Dropping a Profile

Dropping an active profile may have undesirable effects on profile member users. Teradata recommends that you use assign a new profile to the affected users before dropping their current profile. See [MODIFY USER](#).

If you drop an active profile:

- Profile members continue to have the profile defined in their user definitions. If you later recreate a profile with the same name, the profile is again active for profile members.
- Space, account, and default database settings for the dropped profile immediately default to the corresponding settings in the user definition. If the user definition does not have a specific setting, the setting for the owning database or user becomes the default.
- Password control attributes default to the corresponding global settings defined in DBC.SysSecDefaults.
- If you drop a profile used to assign security policy, the policy no longer applies to profile member users. For information on how to assign security policy to database profile objects in a directory, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

Example: Dropping Profile

This request drops the profile named *finance*.

```
DROP PROFILE finance;
```

Related Information

- “GRANT (SQL Form)” topic in *Teradata Vantage™ - SQL Data Control Language*, B035-1149

- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100

DROP ROLE

Drops a specified role.

A DROP ROLE statement removes the privileges on database objects that were granted to the dropped role.

The effect of a role drop (including EXTERNAL roles) is immediate. Users who are logged on with the dropped role as an active role lose the access privileges that were granted to the role.

A DROP ROLE statement only eliminates the specified role. If you no longer need roles that were nested within a dropped role, you must drop them individually.

Default role settings for all users with the dropped role as their default role do not reset to NULL. Affected users receive no warnings or errors the next time they log on. The system does not use the obsolete default role for privileges validation.

If a dropped default role is later recreated, it automatically becomes the default role again, but it has a different role ID number than it had before being dropped.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Required Privileges

The following users can drop a specified role, whether it is a database role or an external role.

- User *DBC*.
- A user who has the DROP ROLE privilege.
- A user who has been granted the specified role WITH ADMIN OPTION.
- A user whose current role has the specified role as a nested role, and the nested role was granted to the current role WITH ADMIN OPTION.

The role creator does not have the implicit privilege to drop a role. A role creator who does not have the DROP ROLE privilege and who loses WITH ADMIN OPTION on a role cannot drop the role.

DROP ROLE Syntax

```
DROP [EXTERNAL] ROLE [ database_name. ] role_name [;]
```

DROP ROLE Syntax Elements

EXTERNAL

Specifies that the role to be dropped is an external role:

- For a database role, you cannot specify EXTERNAL.

- For an external role, you must specify `EXTERNAL`.

database_name

Containing database for *role_name*.

role_name

Name of the role to drop.

DROP ROLE Examples

Example: Dropping Role

This request drops the role named *temp_admin*.

```
DROP ROLE temp_admin;
```

Example: Dropping External Role

This request drops the external role named *rh*.

```
DROP EXTERNAL ROLE rh;
```

Related Information

- GRANT (Role Form) in *Teradata Vantage™ - SQL Data Control Language*, B035-1149
- GRANT (SQL Form) in *Teradata Vantage™ - SQL Data Control Language*, B035-1149
- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100

DROP USER

Drops the definition for an empty user from the Data Dictionary.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

DROP USER is functionally similar to the ANSI SQL:2011 statement DROP SCHEMA.

Required Privileges

You must have the DROP USER privilege on the user. The user that is being dropped cannot own other databases or users, and must be at the bottom of the database hierarchy.

DROP USER Syntax

```
DROP USER user_name [;]
```

DROP USER Syntax Elements

user_name

Name of the user that is to be dropped.

Usage Notes

Effects of Dropping a User on Database Privileges

When you drop a user, the system deletes individual privileges granted directly to the dropped user, and privileges obtained through role assignment, from dictionary tables.

Dropping database objects that exist in the user space prior to the DROP USER statement (required) removes any privileges on the objects from roles and other users or databases.

Any privileges granted by the dropped user on objects outside the user space remain in the system.

Effects of Dropping a User on User-Created Roles and Profiles

Roles and profiles created by a dropped user remain in the system after the user is dropped.

DROP USER Examples

Example: Dropping User, DBQL Case

Suppose the following DBQL rules are in place.

- Rule 1: ALL users, any account string, default logging only.
- Rule 2: ALL users with account string '*finance*', object logging.
- Rule 3: *user_1*, account string '*marketing*', step logging.
- Rule 4: *user_1*, account string '*HR*', default logging only.

You need to drop *user_1*. Before you can do so, you must first stop query logging explicitly for *user_1* so you can remove Rules 3 and 4.

To do this, submit the following requests in the order indicated.

```
END QUERY LOGGING ON user_1 ACCOUNT = ('marketing','HR');
DROP USER user_1;
```


Example: Dropping User

This request drops user *Jones*.

```
DROP USER Jones;
```

Related Information

- [DELETE USER](#)

HELP USER

Displays the attributes, sorted by object name, for all tables, views, join indexes, hash indexes, SQL procedures, user-defined functions, and macros contained by the specified user.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must either own the user or have at least one privilege on that user with one exception: HELP USER DBC does not require any privileges on user DBC.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against a specified user.

HELP USER Syntax

```
HELP USER user_name [;]
```

HELP USER Syntax Elements***user_name***

Name of the user about whom information is required.

HELP USER Example

The HELP USER statement returns the following information for *user_name*.

```
HELP USER  user_name
;
Table/View/Macro Name Kind Comment  Protection  Creator Name
-----
Department           T              F Administration
```

.	.	.	.
.	.	.	.

Related Information

- [SHOW object](#)
- *Teradata Vantage™ - Database Administration, B035-1093*
- *Teradata Vantage™ - Advanced SQL Engine Security Administration, B035-1100*

Map Statements

CREATE MAP

Creates a sparse map.

Sparse maps are intended for small tables, join indexes, and hash indexes. You can colocate tables, join indexes, or hash indexes on the same AMPs. Tables that have the same sparse map and colocation name can be joined using the same primary index or primary AMP index columns without redistributing or duplicating rows. For example, you can colocate two tables, then join the tables on the primary index or primary AMP index columns. See the CREATE TABLE MAP option, [ALTER TABLE \(Map and Colocation Form\)](#), the CREATE JOIN INDEX [MAP](#) option, and the CREATE HASH INDEX MAP option.

You can also specify a sparse map for a table operator when a table operator is included in the FROM clause of a SELECT statement. See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Required Privileges

You must have the CREATE MAP privilege. You must have been granted the contiguous map that you specify in this statement. See GRANT MAP in *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

Privileges Automatically Granted

The map creator is granted the map WITH GRANT OPTION that allows the creator to grant or revoke the created sparse map to users, PUBLIC, and roles.

CREATE MAP Syntax

```
CREATE MAP sparse_map_name FROM contiguous_map_name SPARSE AMPCOUNT = n [;]
```

CREATE MAP Syntax Elements

sparse_map_name

Name of the sparse map.

You cannot specify a name of an existing map or a name that begins with TD_.

For best practice, include "map" in the name, for example OneAMPMap or SmallTable_Map.

FROM *contiguous_map_name*

Name of the parent contiguous map for the sparse map. You must specify the name of an existing contiguous map.

You cannot specify `TD_DataDictionaryMap` or `TD_GlobalMap`.

SPARSE

Required keyword to indicate that the map is a sparse map.

AMPCOUNT = *n*

Specifies the number of AMPs for the map, where *n* is a value from 1 to the number of AMPs in the parent contiguous map, up to 1024.

Usage Notes

A sparse map is specific to the secure zone of the creator.

You cannot use the `CREATE MAP` statement to create a contiguous map.

CREATE MAP Example

This statement creates a 4-AMP sparse map for small tables.

```
CREATE MAP Small_Table_Map FROM TD_Map1 SPARSE AMPCOUNT=4;
```

DROP MAP

Drop a contiguous or sparse map.

When a map is dropped, the map is revoked from `PUBLIC`, users, and roles.

Required Privileges

You must have the `DROP MAP` privilege.

Secure Zones and Dropping a Map

Users and users with roles within a secure zone cannot drop a sparse map created in another secure zone, cannot drop a sparse map not specific to a secure zone, and cannot drop a contiguous map.

Only user `DBC` can drop a sparse map created in another secure zone.

Dropping a Map that Contains Tables

You cannot drop a map that contains tables. You must move the tables to another map before you can drop the map. See [ALTER TABLE \(Map and Colocation Form\)](#).

Dropping a Default Map

You cannot drop the default map for a user, database, or profile. You must set the default map to null or to another map before you can drop the map. See the [DEFAULT MAP](#) option for MODIFY USER, the [DEFAULT MAP](#) option for MODIFY DATABASE, or the [DEFAULT MAP](#) option for MODIFY PROFILE, as appropriate.

Dropping a Map Used by a Table Operator

You cannot drop a map that is being used by a table operator. See the EXECUTE MAP option of the Table operator in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

DROP MAP Syntax

```
DROP MAP map_name [;]
```

DROP MAP Syntax Elements

map_name

Name of the existing contiguous or sparse map to drop.

You cannot drop a contiguous map on which sparse maps are based.

You cannot drop TD_DataDictionaryMap, TD_GlobalMap, or the system-default map.

DROP MAP Example

This statement drops the map, SmallTableMap.

```
DROP MAP SmallTableMap;
```

SHOW MAP

Returns the equivalent CREATE MAP statement in SQL or, optionally, XML.

Required Privileges

None.

Secure Zones and Sparse Maps

For a sparse map, you must be in the same secure zone as the sparse map.

A zone guest can use SHOW MAP for a sparse map in the same secure zone or sparse maps not contained in the secure zone.

SHOW MAP Syntax

```
SHOW [ IN XML ] MAP map_name [ ; ]
```

SHOW MAP Syntax Elements

map_name

Name of an existing contiguous or sparse map to show.

You can show TD_DataDictionaryMap or TD_GlobalMap.

IN XML

Return the MAP object definition in XML format.

SHOW MAP Examples

Following is the sample output for a contiguous map.

```
CREATE MAP TD_Map1  
CONTIGUOUS  
AMP BETWEEN <AmpNum1> AND <AmpNum20>  
;
```

Here is the sample output for a sparse map.

```
CREATE MAP SmallTableMap FROM TD_Map1 SPARSE AMPCOUNT = 1;
```

Load Isolation Statements

BEGIN ISOLATED LOADING

Starts an explicit concurrent isolated load operation on a load isolated (LDI) table. You can perform concurrent read operations on committed rows while the table is being loaded.

After you perform the BEGIN ISOLATED LOADING statement, you must use SET QUERY_BAND to set the LDILoadGroup reserved query band to the load group value for the session before you can begin a load operation. See [LDILoadGroup Query Band](#).

A load isolated table can either be in an explicit load state or implicit load state, but not both. After the explicit load starts, multiple sessions can load the table. Any session performing a load on the table is considered a load session. The table remains in a pending commit state until the load is committed. The table continues to be in load state until you perform an END ISOLATED LOADING statement.

For information on creating a load isolated table, see the CREATE TABLE option, WITH ISOLATED LOADING. You can also alter a table to specify the load isolation option. See the ALTER TABLE option, WITH ISOLATED LOADING.

Required Privileges

You must have the INSERT, UPDATE, or DELETE privilege on the base LDI table.

BEGIN ISOLATED LOADING Syntax

```
BEGIN [CONCURRENT] ISOLATED LOADING ON table_specification [, ...]
  USING QUERY BAND 'LDILoadGroup = value'
  [ [IN] { SINGLE | MULTIPLE } SESSION ] [;]
```

BEGIN ISOLATED LOADING Syntax Elements

table_specification

```
[ database_name. ] table_name
```

CONCURRENT

Keyword you can include for readability.

database_name

The name of the database containing the table, if different from the current default database.

user_name

The name of the user containing the table, if different from the current default user.

table_name

The name of the load isolated table on which to perform the isolated load operation.

LDILoadGroup= *value*

For *value*, you specify a load group to use for the isolated load operation.

IN

Optional keyword to precede the single or multiple session specification.

SINGLE SESSION

Load the specified tables in a single load session using the current session.

MULTIPLE SESSION

Enable multiple sessions to load the specified tables.

To begin a multi-session load operation, the master session performs the BEGIN ISOLATED LOADING statement with the MULTIPLE SESSION option.

Usage Notes

Before you perform the BEGIN ISOLATED LOADING statement, the LDILoadGroup Query_Band value must not be set or must be set to NONE for the session.

You cannot issue a BEGIN ISOLATED LOADING statement from a session already performing a load operation. The current load operation must end and you must set the LDILoadGroup reserved query band to NONE before beginning another load operation.

The BEGIN ISOLATED LOADING statement must be the last statement in an explicit transaction.

BEGIN ISOLATED LOADING Examples

The following statement starts a multi-session load on ldi_table1 and ldi_table2:

```
BEGIN ISOLATED LOADING ON ldi_table1, ldi_table2 USING QUERY_BAND
'LDILoadGroup=Grp1;' IN MULTIPLE SESSION;
```

This statement sets the session as a load session:


```
SET QUERY_BAND='LDILoadGroup=Grp1;' FOR SESSION;
```

LDILoadGroup Query Band

You set the LDILoadGroup reserved query band to control multi-session isolated loading.

You must first perform a BEGIN ISOLATED LOADING statement with the specified load group value before you set this query band. See [BEGIN ISOLATED LOADING](#).

LDILoadGroup Query Band Examples

The following example sets the session as a load session for the tables associated with Grp1:

```
SET QUERY_BAND= 'LDILoadGroup = Grp1;' FOR SESSION;
```

This example removes the LDILoadGroup query band from the session:

```
SET QUERY_BAND= 'LDILoadGroup=NONE;' FOR SESSION;
```

CHECKPOINT ISOLATED LOADING

Sets a checkpoint for the explicit isolated load operation and commits the data that has been loaded up to that point.

Required Privileges

You must have the INSERT, UPDATE, or DELETE privilege on the base table associated with the load operation.

CHECKPOINT ISOLATED LOADING Syntax

```
CHECKPOINT [CONCURRENT] ISOLATED LOADING FOR QUERY BAND 'LDILoadGroup =  
value' [;]
```

CHECKPOINT ISOLATED LOADING Syntax Elements

CONCURRENT

Keyword you can include for readability.

LDILoadGroup= *value*

For *value*, you specify the load group in use for the isolated load operation.

Usage Notes

The CHECKPOINT ISOLATED LOADING statement must be issued by the same session as the BEGIN ISOLATED LOADING statement.

When you issue the CHECKPOINT ISOLATED LOADING statement, the data that has been loaded up to that point by the session, or by all sessions in a multi-session load operation, is committed and becomes available for concurrent read operations. You can use the CHECKPOINT ISOLATED LOADING statement to enable recently loaded data to be available for concurrent read operations while the load operation continues.

The CHECKPOINT ISOLATED LOADING statement must be the last statement in an explicit transaction.

CHECKPOINT ISOLATED LOADING Example

This statement sets a checkpoint for the load operation:

```
CHECKPOINT ISOLATED LOADING FOR QUERY_BAND 'LDILoadGroup=Grp1';
```

END ISOLATED LOADING

Ends the explicit concurrent isolated load operation for the specified load group value.

Required Privileges

You must have the INSERT, UPDATE, or DELETE privilege on the base table associated with the load operation.

END ISOLATED LOADING Syntax

```
END [CONCURRENT] ISOLATED LOADING FOR
  QUERY BAND 'LDILoadGroup = value' [ OVERRIDE SESSION ] [;]
```

END ISOLATED LOADING Syntax Elements

CONCURRENT

Keyword you can include for readability.

LDILoadGroup= *value*

For *value*, you specify the load group in use for the isolated load operation.

OVERRIDE SESSION

Enables you to end the load operation from a different session than the session that issued the BEGIN ISOLATED LOADING statement.

Usage Notes

For a single session load, the END ISOLATED LOADING statement must be issued by the same session as the BEGIN ISOLATED LOADING statement, unless you specify the OVERRIDE SESSION option. In a multiple session load, the END ISOLATED LOADING statement can be issued from any load session.

The END ISOLATED LOADING statement must be the last statement in an explicit transaction.

After you issue the END ISOLATED LOADING statement, you must use SET QUERY_BAND statement to set the LDILoadGroup query band to NONE.

END ISOLATED LOADING Example

This statement ends concurrent isolated loading for the load operation:

```
END ISOLATED LOADING FOR QUERY_BAND 'LDILoadGroup=Grp1';
```

Secure Zones Statements

CREATE ZONE

Defines a Teradata Secure Zone. You can also specify an existing database or user as the zone root.

A secure zone is an area which restricts unauthorized outside users from accessing data and objects within the zone.

The zone root is a database or a user on which a zone is created. You create a secure zone and associate a database or a user as the root. Before assigning the database or user as the root of a zone, the database or user should not contain any objects, users, databases, roles, or profiles.

For information on adding guests to a zone, see GRANT ZONE in *Teradata Vantage™ - SQL Data Control Language*, B035-1149

For information on assigning a primary Database Administrator to a zone, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100

Required Privileges

To create a secure zone, you must have the CREATE ZONE privilege.

To define a user as a root, the zone creator must have DROP USER privilege on the user that becomes a root.

To define a database as root, the zone creator must have CREATE USER privilege on the database that becomes a root.

CREATE ZONE Syntax

```
CREATE ZONE zone_name [ ROOT { database_name | user_name } ] [;]
```

CREATE ZONE Syntax Elements

zone_name

Specify a name for the secure zone.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

ROOT

The zone root is a database or a user on which a secure zone is created. You create a secure zone and associate a database or a user as the root. Before assigning the database or user

as the root of a secure zone, the database or user should not contain any associated objects, users, databases, roles, or profiles.

Root should be an existing user or database. You should create the user or database before you specify the user or database as root.

The user or database that you specify cannot be a guest, creator, or member of another secure zone and cannot contain objects, descendants, roles, profiles, or constraints.

Only the zone creator, owner, and creator of root can have privileges on the root. Other users cannot have privileges on the root. Root cannot have any privileges on other users.

Root cannot be a proxy user.

Root should not be granted with any roles or constraints.

The user should not be logged in while adding the user as a zone root.

Root cannot have a database as default database.

The user or database that you specify cannot have any Access logging or Query logging rules enabled on them or created by them.

database_name

Name of database.

user_name

Name of user.

CREATE ZONE Examples

Example: Create a secure zone without a root

```
CREATE ZONE zn_01;
```

Example: Create a secure zone with a predefined user or database as root

```
CREATE ZONE zn_01 ROOT z_root01;
```

ALTER ZONE

Add a database or user as the root to a secure zone or de-link the root database or user from a secure zone created using Teradata Secure Zones.

Required Privileges

To alter a secure zone, you must have the DROP ZONE privilege.

To define a user as a root, the zone creator must have DROP USER privilege on the user that becomes a root.

To define a database as root, the zone creator must have CREATE USER privilege on the database that becomes a root.

ALTER ZONE Syntax

```
ALTER ZONE zone_name [ ADD ROOT { database_name | user_name } | DROP ROOT ] [;]
```

ALTER ZONE Syntax Elements

zone_name

Name of an existing secure zone.

ADD ROOT

Add a root database or user to a secure zone.

Only the secure zone creator can ADD a root.

The root cannot be a part of another zone.

Root should be an existing user or database. You should create the user or database before you specify the user or database as root.

The user or database that you specify cannot be a guest, creator, or member of another zone and cannot contain objects, descendants, roles, profiles, or constraints.

Only the zone creator, owner and creator of root can have privileges on the root. Other users cannot have privileges on the root. Root cannot have any privileges on other users.

Root cannot be a proxy user.

Root should not be granted with any roles or constraints.

The user should not be logged in while adding him as a zone root.

Root cannot have a database as default database.

The user or database that you specify cannot have any Access logging or Query logging rules enabled on them or created by them.

DROP ROOT

De-link the root database or user from the secure zone.

Only the zone creator can DROP a root.

The root cannot contain any objects, descendants, roles, constraints, or profiles.

The root database or user cannot have any Access logging or Query logging rules enabled on them or created by them.

All the zone guests must be removed from the zone.

The root user should not be logged in while dropping zone root.

ALTER ZONE Examples

Example: Add Root

```
ALTER ZONE zn_01 ADD ROOT zn_rt01;
```

Example: Drop Root

```
ALTER ZONE zn_01 DROP ROOT;
```

DROP ZONE

Remove a secure zone.

Only the zone creator can drop a zone.

To drop a zone, the zone cannot be associated with any root, Primary Database Administrator, or Zone Guest.

Required Privileges

To drop a zone, you must have the DROP ZONE privilege.

DROP ZONE Syntax

```
DROP ZONE zone_name [;]
```

DROP ZONE Syntax Elements

zone_name

Name of an existing zone.

DROP ZONE Example

```
DROP ZONE zn_01;
```

Session Statements

SET SESSION

Allows the setting of various session parameters for the entire session or individual requests within the session.

For more information, see the individual SET SESSION commands:

- [SET SESSION ACCOUNT](#)
- [SET SESSION CALENDAR](#)
- [SET SESSION TRANSACTION ISOLATION LEVEL](#)
- [SET SESSION COLLATION](#)
- [SET SESSION CONSTRAINT](#)
- [SET SESSION DATABASE](#)
- [SET SESSION DATEFORM](#)
- [SET SESSION DEBUG FUNCTION](#)
- [SET SESSION DOT NOTATION](#)
- [SET SESSION FOR ISOLATED LOADING](#)
- [SET SESSION FUNCTION TRACE](#)
- SET SESSION *temporal_qualifier* in *Teradata Vantage™ - Temporal Table Support*, B035-1182
- SET SESSION TTGRANULARITY in *Teradata Vantage™ - Temporal Table Support*, B035-1182
- [SET SESSION UDFSEARCHPATH](#)
- [SET QUERY_BAND](#)
- [SET TIME_ZONE](#) functions in a similar way, but does not require the keyword SESSION.
- [HELP SESSION](#)

SET SESSION Syntax

```
{ SET SESSION | SS } KEYWORD [=] value [;]
```

Usage Notes

Using SET SESSION Statements with Connection Pooling

Connection pooling is a technique for sharing server resources among end users of a requesting client application. It enables multiple end users to share a cached set of connection objects that provides

access to Vantage. Connection pooling improves performance by eliminating the overhead associated with establishing a new connection to Vantage for each request submitted by a client application end user.

Because it is not possible to reset a connection to Vantage, you should avoid changing any of the following session parameters during a pooled session (including trusted sessions), because these parameter changes are inherited by the next user of the pooled connection.

- Character set
- Collation
- Database
- Dateform
- Default Date Format
- Time zone
- Transaction Isolation Level

SET SESSION ACCOUNT

Dynamically changes your account or account priorities for the duration of a session or for one SQL request only.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Although SET SESSION ACCOUNT is technically a DCL statement, Vantage treats it as a DDL statement for transaction semantics.

Required Privileges

There are no privileges required to execute a SET SESSION ACCOUNT request, however, the specified account must be assigned to the executing user.

SET SESSION ACCOUNT Syntax

```
{ SET SESSION | SS } ACCOUNT = 'account_string' FOR { SESSION | REQUEST } [;]
```

SET SESSION ACCOUNT Syntax Elements

account_string

An account string assigned in the user or profile definition for the logged on user. If the account string is not valid for the user, an error occurs.

Changing the account string changes the access level for the session or request only if different workload management rules apply to the new account.

A user cannot use the SQL version of the SET SESSION ACCOUNT command to change the account for any other user.

To change accounts for another user, you can use the Performance Manager SET SESSION ACCOUNT command. See *Teradata Vantage™ - Application Programming Reference*, B035-1090.

For information about account string options and session priorities, see *Teradata Vantage™ - Database Administration*, B035-1093.

For information about creating accounts, see [CREATE USER](#) or [CREATE PROFILE](#).

SESSION

Specifies that the account is changed for the remainder of the session.

You cannot undo this request within the current session. To revoke a session-level account change, a user must log off the session, then log back on, at which point the system assigns the user default priority.

REQUEST

Specifies that the account is changed for the next request only (next being defined as the first request this user performs after the current SET SESSION ACCOUNT request).

After that, the previous account resumes.

The specified account information is kept in volatile memory, not in DBC.SessionTbl, so a subsequent transparent system reset and recovery effectively voids this request. To be sure the request performs correctly, set SET CRASH to NOWAIT_TELL or verify the result.

Example: Setting the Account to Change Resource Charges for a Session

User A, employed in the Marketing Group, has the default account '\$M0+MKTG&S&D&H'. The user wants to change to the FIN1 account to charge the system resources used to generate the monthly finance report to the Finance Group.

```
SET SESSION ACCOUNT='M0+FIN1&S&D&H' FOR SESSION
```

Example: Setting the Account to Change the Priority for a Request

User B is submitting a series of large data mining queries using the user default account '\$L0+DMIN&S&D&H'. The user wants to change to an account with a higher priority to run an emergency report for his manager.

```
SET SESSION ACCOUNT='H0+EMRG&S&D&H' FOR REQUEST
```

After running the emergency report, the account reverts to the default for User B, who continues with low priority data mining.

Changing the account string changes the access level for the session or request only if different workload management rules apply to the new account.

Related Information

For more information about Vantage accounts, see *Teradata Vantage™ - Database Administration*, B035-1093.

SET SESSION CALENDAR

Sets the default calendar for the session to a system-defined calendar.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Although SET SESSION CALENDAR is technically a DCL statement, Vantage treats it as a DDL statement for transaction semantics.

Required Privileges

None.

SET SESSION CALENDAR Syntax

```
{ SET SESSION | SS } CALENDAR = calendar_name [;]
```

SET SESSION CALENDAR Syntax Elements

calendar_name

Name of the system-defined calendar to be used as the default calendar for the session.

The calendar name you specify cannot be user-defined.

The default calendar name is Teradata.

The other valid system-defined calendar names are ISO and COMPATIBLE.

See *Teradata Vantage™ - Data Types and Literals*, B035-1143 for more information about these calendars.

Example: Changing the Default Session Calendar Using SET SESSION CALENDAR

This example sets the default calendar for the session from Teradata to the system-defined calendar ISO.

```
SET SESSION CALENDAR=ISO;
```

This example sets the default calendar for the session from Teradata to the system-defined calendar COMPATIBLE.

```
SET SESSION CALENDAR=COMPATIBLE;
```

Example: Changing the Default Session Calendar Using CREATE USER

You can specify a different default session calendar for a user by specifying the SET SESSION CALENDAR SQL text string as the startup string for the user by submitting a CREATE USER or MODIFY USER statement.

Assuming that the default business calendar for the site is Teradata, the following CREATE USER statement creates a user named john_smith and uses the STARTUP string for john_smith to set his session calendar to COMPATIBLE instead of Teradata each time he starts a new session.

```
CREATE USER john_smith AS
  PERM=10E6,
  PASSWORD=john_smith,
  SPOOL = 1200000,
  FALLBACK PROTECTION,
  STARTUP='SET SESSION CALENDAR=COMPATIBLE';
```

Related Information

- SET SESSION CALENDAR in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - Data Types and Literals*, B035-1143
- *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211

SET SESSION CHARACTER SET UNICODE PASS THROUGH

Enables or disables Unicode Pass Through processing for the session. Pass Through Characters (PTCs) include the Unicode characters that are not currently supported and character codes reserved for future use.

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

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

None.

SET SESSION CHARACTER SET UNICODE PASS THROUGH Syntax

```
{ SET SESSION | SS } CHARACTER SET UNICODE PASS THROUGH [ ON | OFF ] [;]
```

SET SESSION CHARACTER SET UNICODE PASS THROUGH Syntax Elements

ON

Enables Unicode Pass Through processing.

OFF

Disables Unicode Pass Through processing. This is the default.

Example: Enabling Unicode Pass Through for the Session

```
SET SESSION CHARACTER SET UNICODE PASS THROUGH ON;
```

SET SESSION COLLATION

Changes the collation sequence for the current session.

Note:

You can use the HELP SESSION request to find out the collation that is currently in effect.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Although SET SESSION COLLATION is technically a DCL statement, Vantage treats it as a DDL statement for transaction semantics.

Required Privileges

None.

SET SESSION COLLATION Syntax

```
{ SET SESSION | SS } COLLATION collation_sequence [;]
```

SET SESSION COLLATION Syntax Elements

COLLATION

Specifies that the collation sequence in effect for the session is to be changed.

The COLLATION option defines the name for a collation sequence that determines the ordering of data characters during comparison operations and when sorting data in response to a SELECT query that includes an ORDER BY or WITH ... BY clause.

You can override the COLLATION attribute during any session by running an appropriate SET SESSION COLLATION request (see [SET SESSION COLLATION](#)).

collation_sequence

Name of the collation sequence to be established for the session as one of the following:

- ASCII
- CHARSET_COLL
- EBCDIC
- HOST
- JIS_COLL
- MULTINATIONAL

For details about collation, see [Collation Option Usage](#) and [MULTINATIONAL Collation](#).

ASCII

HOST.

The collation is ASCII for configurations other than an IBM mainframe-attached host.

In this case, ASCII refers to the Teradata extension to the ASCII standard.

CHARSET_COLL

HOST.

The CHARSET_COLL collation performs binary ordering based on the current client character set. Strings are compared byte-by-byte.

When one comparison string is shorter than the other, it is padded with the pad character defined for the character data type before the comparison is made.

When comparisons are not case specific, the following process occurs.

1. Lowercase letters are mapped to their uppercase counterparts.
2. The strings are compared.

IF the strings are ...	THEN the ...
identical	equality relation holds.
not identical	first pair of bytes that is not equal determines the collating sequence.

When string comparisons involve one or more characters outside the current client character set, then the following behavior occurs.

1. The strings are compared.
2. Characters are checked to determine if they are all within the repertoire of the current client character set.

IF the characters compared have this relationship to the client character set ...	THEN ...
both are in	the binary ordering of the two characters in the client form-of-use becomes the ordering of the two strings.
one is not in	the error character for the character set is used as the collation point for that character.
neither is in	the binary ordering of the characters, either case blind or case specific, as appropriate) in the UNICODE form-of-use becomes the ordering of the two strings.

CHARSET_COLL string sorts by character data type behave as follows.

FOR this character data type ...	CHARSET_COLL collation orders characters as follows ...
Kanji1	Single-byte characters based on the current character set. Multibyte characters based on their internal value. KANJI1 character data types can contain mixed single-byte/multibyte character sets.

FOR this character data type ...	CHARSET_COLL collation orders characters as follows ...
	Single-byte characters in KANJI1 are translated into the form-of-use. Multibyte characters in KANJI1 are not translated.
KanjiEBCDIC KanjiShift-JIS	As a binary sort on the client would be ordered.
KanjiEUC	As a Kanji Phase I ASCII collation. The difference with a binary sort on the client is that JIS X 0208 characters sort before, rather than after, JIS X 0212 characters.

You can specify CHARSET_COLL as the default user collation in CREATE USER (see [CREATE USER](#)) or in MODIFY USER ([MODIFY USER](#)) if the user definition already exists and you wish to change it.

SET SESSION COLLATION CHARSET_COLL overrides any such user default definitions for the duration of the session in which it is invoked.

EBCDIC

The collation is EBCDIC for an IBM mainframe-attached host.

HOST

The HOST collation orders characters using:

- EBCDIC encoding for IBM mainframe-attached clients.
- ASCII encoding for all other clients

JIS_COLL

Japanese Industrial Standards (JIS).

The collation is as follows:

1. Characters and symbols from the JIS X 0201 standard (in JIS X 0201 order).
2. Ideographs, characters, and symbols from the JIS X 0208 standard (in JIS X 0208 order).
3. Ideographs, characters, and symbols from the JIS 0212 standard (in JIS X 0212 order).
4. IBM Kanji ideographs not present in JIS X 0208 and JIS X 0212 (in KanjiEBCDIC order).
5. User-defined ideographs (in U+ order).
6. Any remaining characters in Unicode (in U+ order).

Treatment of KANJISJIS, GRAPHIC, and LATIN character types is as if the data were first converted to Unicode and then the appropriate JIS_COLL ordering is applied.

Do not specify JIS_COLL collation for the KANJI1 character set.

MULTINATIONAL Collation

The MULTINATIONAL keyword defines the name of the default collation sequence for the user to be one of the International sort orders (see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146).

The MULTINATIONAL parameter determines that the International sort order is to be used. This parameter returns an error unless the hashing algorithm is set to recognize diacritical characters. The MULTINATIONAL option is useful only on a European Feature Site or a Japanese language site.

On International/European sites MULTINATIONAL collation is two-level.

On Japanese language sites, collation is single-level and is always available.

MULTINATIONAL sets the collation for the session to an international sort sequence that is compatible with either a diacritical character set or a Japanese character set, as described in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Each character in this collation is assigned two integer values. The first indicates the equivalence class of the character and the second indicates the ordering of the character within that class. For example, the letters LATIN CAPITAL LETTER A (Latin A) and LATIN CAPITAL LETTER A WITH GRAVE (Latin Å) might belong to the same class, but are ordered differently.

The general ordering follows UNICODE ordering: Latin A precedes Latin B, and so on.

Single byte MULTINATIONAL collations, such as the Scandinavian languages and other user installed collations, are extended to include the UNICODE repertoire. The remainder of the UNICODE character repertoire collates the same as the default MULTINATIONAL collation.

When you apply MULTINATIONAL to KANJISJIS, GRAPHIC, or LATIN character data types, the result is as if the data were first converted to UNICODE and the appropriate MULTINATIONAL ordering applied.

Do not specify MULTINATIONAL collation for the KANJI1 character set. MULTINATIONAL collates KANJI1 data with only a single-level comparison rather than the proper two-level comparison.

Do not specify MULTINATIONAL collation for the Katakana EBCDIC, KanjiEBCDIC5035_I, or KanjiEBCDIC5026_OI character data types either because the definitions of MULTINATIONAL collation designed for KANJI1 data also apply to non-KANJI1 data, and the results are unpredictable. Instead, use CHARSET_COLL for sessions with these character sets.

Each international sort sequence is defined by the database administrator (and no check is made to ensure that collation is compatible with the character set of the current session) but multibyte character collation sequences cannot be changed.

Usage Notes

SET SESSION COLLATION is not valid in 2PC sessions.

Character data is sorted in ascending order unless the DESC (descending) option is included in the SQL request.

Vantage converts data characters to their uppercase values for comparison and sorting operations unless the CASESPECIFIC option is included in the SQL request, or was defined at creation time for the column being queried. The results of CASESPECIFIC on sorted results is described in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

SET SESSION COLLATION Examples

Example: Setting a Session to Use the ASCII Collation Sequence

A session can be set to ASCII collation as follows.

```
SET SESSION COLLATION ASCII;
```

Example: Effect of the ASCII Collation Sequence on SQL Report Output

The session collation is set to ASCII. Assuming that *job_rept* contains alphanumeric data, a SELECT that requests a sorted return, as follows.

```
SELECT emp_no, jobid
FROM job_rept
ORDER BY job_id;
```

This request produces the following report, sorting *job_id* in ascending order.

EmpNo	JobId
-----	-----
10201	1001-AP2
10201	1032-AR3
10201	1031-AR2
10004	ENG-0002
10016	ENG-0002
10004	ENG-0003
10003	OE1-0001
10001	PAY-0002

Example: Setting a Session to Use the EBCDIC Collation Sequence

If the same session from [Example: Effect of the ASCII Collation Sequence on SQL Report Output](#) is then set to EBCDIC collation as follows.

```
SET SESSION COLLATION EBCDIC;
```

The same SQL request produces the following report, sorting *job_id* in the ascending order defined for EBCDIC.

EmpNo	JobId
-----	-----
10004	ENG-000
10016	ENG-0002
10004	ENG-0003
10003	OE1-0001
10001	PAY-0002
10201	1001-AP2
10201	1032-AR3
10201	1031-AR2

Example: Effect of the EBCDIC Collation Sequence on SQL Report Output

If the descending option is specified for the query in [Example: Setting a Session to Use the EBCDIC Collation Sequence](#) in place of the default sort direction, which is ascending, the following report is produced, sorting *job_id* in descending EBCDIC order.

```
SELECT emp_no, job_id
FROM job_rept
ORDER BY job_id DESC;
emp_no      job_id
-----
10201       1031-AR2
10201       1032-AR3
10201       1001-AP2
10001       PAY-0002
10003       OE1-0001
10004       ENG-0003
10004       ENG-0002
10016       ENG-0002
```

Related Information

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

SET SESSION CONSTRAINT

Overrides the default constraints assigned to a user for the current session.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Although SET SESSION CONSTRAINT is technically a DCL statement, Vantage treats it as a DDL statement for transaction semantics.

Required Privileges

None.

SET SESSION CONSTRAINT Syntax

```
{ SET SESSION | SS } constraint [,...] [;]
```

constraint

```
CONSTRAINT= row_level_security_constraint_name {  
  ( level_name | category_name [,...] | NULL )  
}
```

SET SESSION CONSTRAINT Syntax Elements

CONSTRAINT

One or more security constraint values (levels and categories) are being changed for the session.

row_level_security_constraint_name

Name of an existing constraint.

The specified *constraint_name* must be currently assigned to the user.

You can specify a maximum of 6 hierarchical constraints and 2 non-hierarchical constraints per SET SESSION CONSTRAINT statement.

level_name

Name of a hierarchical level, valid for the *constraint_name*, that is to replace the default level.

The specified *level_name* must be currently assigned to the user. Otherwise, Vantage returns an error to the requestor.

category_name

A set of one or more existing non-hierarchical category names valid for the *constraint_name*.

Because all assigned category (non-hierarchical) constraint values assigned to a user are automatically active, SET CONSTRAINT is only useful to specify a subset of the assigned categories for the constraint.

For example, assume that User BOB has 3 country codes, and wants to load a table with data that is to be made available to User CARL who only has rights to see data for his own country. User BOB can use SET SESSION CONSTRAINT to specify only the country code for User CARL when loading the data so Carl can access the data later.

NULL

If you specify NULL and then update a table, only users with OVERRIDE privileges can subsequently access the affected table rows.

Usage Notes

Rules for Determining Whether Profile Defaults or User Defaults Are Assigned as Session Row-Level Security Defaults

When you establish a session, the default value for row-level security constraints assigned to the profile for the user or those directly assigned to the user become current for the session. Those row-level security constraints that are assigned to a profile take precedence over those assigned to the user. The rules for determining the initial assignment of constraint values for a session are as follows.

- If a user has been assigned a profile and that profile has been assigned row-level security constraint values, those profile constraints become current for the session.

In this case, Vantage does not consider any constraints that are assigned directly to the user.

- If a user has been assigned a profile and that profile has not been assigned any row-level security constraints, then Vantage assigns any constraints for the session from the user definition.

If there are no row-level security constraints assigned to the user, all constraints for the session are null.

- If a user has not been assigned a profile, then Vantage allocates the row-level security constraints assigned directly to the user for the session.

If no row-level security constraints are assigned to the user, all constraints for the session are set null.

- If a hierarchical constraint with multiple assigned values is assigned to a session, then Vantage only assigns the default values for the source to the session.
- If a non-hierarchical constraint with multiple assigned values is assigned to a session, then Vantage applies all assigned values for the source to the session.

Using SET SESSION CONSTRAINT To Modify the Constraints Assigned to You for the Current Session

Several possible uses exist for using a SET SESSION CONSTRAINT request to alter the row-level security constraints in effect for the session. The noted changes can be applied whether the default row-level security constraint set for the session was assigned to a user or to a profile. The following points indicate some of the possible uses for a SET SESSION CONSTRAINT request.

- To change the default level (hierarchical) constraint for a session.
- To change the active category (non-hierarchical) constraint set for a session to a subset of those defined for the constraint.

Using the NULL Option for SET SESSION CONSTRAINT

The purpose of the NULL option for SET SESSION CONSTRAINT is similar to the NULL option for SET ROLE, where the request removes the active role for a session.

If the session is set to NULL for a constraint in place of one or more constraint names, the assigned constraint set must be currently assigned to the session. If you do not have the OVERRIDE privilege, Vantage returns an error. Constraint UDFs can accept a null from data rows, but not from a user session.

Determining the Session Constraint Values

The security constraint values used by the system for a session vary based on several factors. In some cases, the user must manually supply the values.

Session Constraint Values for Permanent Database Users

When a permanent database user accesses a row protected by a security constraint, the system determines the session constraint value(s) as follows:

1. If the user profile has a corresponding security constraint assignment, the session uses the constraint value(s) specified in the profile definition.
2. If the user profile does not have a security constraint specification, the system derives the session constraint value(s) from the user definition.
3. Whether user constraint values are defined in a profile or user definition, if multiple constraint values are defined, the system determines the session value(s) as follows:
 - For hierarchical (non-set) constraints, the system uses the DEFAULT value specified in the profile, or if there is no profile, in the user definition. If no DEFAULT is specified, the system uses first value listed for the constraint.
 - For non-hierarchical (set) constraints, the session uses all constraint values in the profile, or if there is no profile, in the user definition. No DEFAULT can be specified.

Note:

Requesting users can use SET SESSION CONSTRAINT to change the session constraint value to any constraint value specified in the profile or user definition.

4. If neither the user profile nor the user definition contains a security constraint assignment, the constraint value for the session is NULL, and the user can access rows controlled by a security constraint only if assigned the necessary OVERRIDE privileges.
5. If a user has OVERRIDE privileges on the object and the operation being performed, the system ignores constraints assigned in the profile or user definition. The session derives security constraint values in one of the following ways:
 - For simple inserts, the user must supply the constraint value(s).
 - For compound statements, for example, INSERT ... SELECT or MERGE, the system derives constraint value(s) from the security constraint columns in the source table.
 - The user can use SET SESSION CONSTRAINT to specify constraint values.

Session Constraint Values for Directory Users

- A directory user who is mapped to a permanent user inherits the permanent user privileges and constraint assignments.
- A directory user who is not mapped to a permanent database user, but who uses Sign-On As to log on as a permanent user, inherits the user privileges and constraint assignments.
- Directory users who are mapped to multiple constraint value sources (users or profiles) must use the `user=user_name` or `profile=profile_name` option in the logon string to specify the source for default constraint values and OVERRIDE privileges.
- Directory users can use the SET SESSION CONSTRAINT option to access any mapped or inherited constraint assignments.
- For directory users with no mappings and only PUBLIC or EXTUSER privileges, the session constraint value is NULL, and the user cannot access row level security tables.

Session Constraint Values for Application Pooled Users

For mainframe and middle-tier application pooled sessions (not trusted sessions), users inherit the row level security privileges for the mainframe or application logon user, or its profile, if used. System processing of constraint values is the same as for any other permanent database user.

Session Constraint Values for Trusted User Applications and Proxy Users

All users logging on through a middle-tier trusted user application inherit the security constraint assignments and access privileges of the application logon user (trusted user), and can use the SET SESSION CONSTRAINT statement to switch from the default constraint values to other values available to the trusted user.

See [Using SET SESSION to Change the Session Security Constraint Value](#).

Initial SET QUERY_BAND Processing

If the trusted user application submits an initial SET QUERY_BAND statement (a standard operation for trusted user applications) the system resets the session constraint values to an empty set, and takes the following actions:

- If the statement does not set a proxy user, the session uses the default constraint value for the trusted user according to the normal permanent user constraint hierarchy. See [Session Constraint Values for Permanent Database Users](#).
- If the statement sets a proxy user, and the user is also a permanent database user, the system uses the default constraint value for the permanent user, according to the normal permanent user constraint hierarchy, including profile. See [Session Constraint Values for Permanent Database Users](#).
- If the statement sets a proxy user, and the user is not also a permanent database user, the system is unable to set a session constraint value and also will not accept a SET SESSION CONSTRAINT statement. As a result, proxy users that are not also permanent database users cannot access row level security tables.

Subsequent SET QUERY_BAND Processing without Update Option.

If the trusted user application submits a subsequent SET QUERY_BAND statement without an update option, the statement causes the constraint values currently active for the session to reset to an empty set. The new constraint values for the session are determined according to the rules shown in Initial SET QUERY_BAND Processing.

Subsequent SET QUERY_BAND Processing with Update Option

If the trusted user application submits a subsequent SET QUERY_BAND statement with an update option, the resulting constraint values for the update depend on whether the session currently has a proxy user assigned:

- If the session did not previously define a proxy user and a SET QUERY_BAND update adds a proxy user that is a permanent database user, the system determines session constraint values using the normal constraint hierarchy for the permanent user. The user can change the value to other permanent user constraint values using SET SESSION CONSTRAINT.
- If the session did not previously define a proxy user and the SET QUERY_BAND update adds a proxy user that is not a permanent database user, there are no constraint values allocated to the session. The user cannot access a row level security table.
- If the session previously defined a proxy user and the SET QUERY_BAND update defines a new proxy user, the constraint values for the session depend on whether the new proxy user is also a permanent user:
 - If the user is a permanent user, the system determines session constraint values using the normal constraint hierarchy for the permanent user. The user can use a SET SESSION CONSTRAINT statement to change constraint values active for the session to other available user or profile values.

- If the user is not also a permanent user, no constraint values are allocated to the session. A SET SESSION CONSTRAINT statement to change constraint values active for the session will be rejected with an error.

END TRANSACTION Processing

If the user executes a statement terminating a transaction during a trusted session, the constraint values for the session depend on whether the user is a proxy user.

- If the proxy user is assigned to the session only for the current transaction, the constraint values assigned to the session revert to those set at the initial connect of the session, that is, those for the application logon user.
- If the proxy user is assigned to the session for all transactions then the constraint values are not changed, and are the values assigned to the proxy user.

Session Constraint Values in OVERRIDE Sessions

The user conducting a session that accesses an RLS table may need OVERRIDE privileges if:

- The CONSTRAINT object associated with a constraint column in the table does not define a UDF for the SQL operation.
- Accessing table rows or setting row level constraint values cannot be done effectively with the assigned user defaults.

Note:

For additional information on OVERRIDE privileges, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

When an OVERRIDE is in effect for a session:

- The OVERRIDE privileges bypasses enforcement of the affected UDFs.
- If the operation is an INSERT or UPDATE, the user SQL must supply the value(s) to be entered into the named constraint column in the table, either using data from the source table or values specified in a SET SESSION CONSTRAINT statement.

Note:

Loading tables with non-hierarchical constraints in an OVERRIDE session requires special handling. See [Example: Loading Tables with User OVERRIDE Privileges](#).

Specifying Non-Hierarchical Constraint Values when Loading Tables

To provide the capacity for the largest number of values in the smallest space, non-hierarchical constraint values are defined in RLS tables as bit positions rather than as numeric values. The bit positions are represented in the table as hex code.

The method by which hex-encoded constraint values are loaded into an RLS table varies depending on the presence or absence of the `OVERWRITE` privilege.

Using `SET SESSION` to Change the Session Security Constraint Value

Users assigned more than one value for a security constraint can use `SET SESSION CONSTRAINT` to replace the default value with another assigned value, for example:

```
SET SESSION
CONSTRAINT = constraint_name
  {(value_name
    ... [,value_name
  ]|(NULL)} ... [,CONSTRAINT = constraint_name
  {(value_name ... [,value_name)]|(NULL)}
```

CONSTRAINT = *constraint_name*

A security constraint for which the user wants to reset the session default value, which must be:

- An existing `CONSTRAINT` object
- Currently assigned to the user

A `SET SESSION` statement can specify maximum of 6 non-set and 2 set constraints.

value_name

One or more value names that are valid for the specified constraint name and executing user.

The *value_name* specification(s) replace the current session constraint value.

If values for a constraint are already assigned to a session and the constraint is not named in the `SET SESSION` request, the values for that constraint remain unchanged.

Value_name specifications are subject to the following limitations:

- For non-hierarchical (set) constraints, you can specify as many of the values assigned to the user (in the profile or user definition) as you need. The system uses the specified value(s) as the session value(s).
- For hierarchical (non-set) constraints, you can only specify one alternate value from either the profile or user definition.

[*value_name*]|(NULL)

If the constraint name is followed by the `NULL` option, the constraint is removed from the session label.

Using HELP SESSION to Investigate Session Constraint Values

A user who is unsure of the session constraint value(s) can use the `HELP SESSION CONSTRAINT` command to get a list of the security constraints and associated constraint values for the session.

SET SESSION CONSTRAINT Examples

Constraint Definitions for the Examples

The following constraint and user definitions are used for the examples that follow.

The following SQL text creates a hierarchical constraint named *classification_level* that enforces all four of the possible statement actions for a constraint.

```
CREATE CONSTRAINT classification_level SMALLINT, NOT NULL,
VALUES (top_secret:4, secret:3, confidential:2, unclassified:1),
INSERT SYSLIB.insert_level,
UPDATE SYSLIB.update_level,
DELETE SYSLIB.delete_level,
SELECT SYSLIB.read_level;
```

The following SQL text creates a non-hierarchical constraint named *classification_category* that enforces all four of the possible statement actions for a constraint.

```
CREATE CONSTRAINT classification_category BYTE(8),
VALUES (nato:1, united_states:2, canada:3, united_kingdom:4,
       france:5, norway:6, russia:7),
INSERT SYSLIB.insert_category,
UPDATE SYSLIB.update_category,
DELETE SYSLIB.delete_category,
SELECT SYSLIB.read_category;
```

Create *user_name*.

```
CREATE USER user_name AS
PERMANENT = 1e6,
PASSWORD=my_pwd,
CONSTRAINT = classification_level (top_secret),
CONSTRAINT = classification_category (united_states);
```

Creates user *p/s*.

```
CREATE USER pls AS
PERMANENT = 1e6,
PASSWORD=secret,
CONSTRAINT = classification_level (secret, unclassified DEFAULT),
CONSTRAINT = classification_category (united_states);
```

The following SQL text creates user *arn_anderson*.

```
CREATE USER arn_anderson AS
PERMANENT = 1e6,
PASSWORD=Arn2222ANDERSON,
CONSTRAINT = classification_category (norway);
```

Example: Changing the Security Level and Category for a Session

The *user_name* logs on. The resulting session has a label consisting of an *unclassified* level and a *nato* category. The first request executed for the session changes the label to a *top_secret* level and the category to a combination of *united_states* and *nato*.

```
SET SESSION CONSTRAINT = classification_level (top_secret),
CONSTRAINT = classification_category (nato, united_states);
```

Assume that later on, the session initiated by *user_name* wanted to read one of the three rows from *inventory*, so the user submits the following SELECT request.

```
SELECT *
FROM inventory
WHERE col_1 = 1212;
```

The returned result set would be as follows.

Col_1	Col_2	Col_3	Classification_Level
1212	90505	Widgets	3
			'0100000000000000'XB

Example: Changing the Row-Level Security Level for a Session

User *pls* logs on. The resulting session has a label consisting of an *unclassified* level and a *nato* category. As soon as the session is established, *pls* changes the level to *secret*.

```
SET SESSION CONSTRAINT = classification_level (secret);
```

After the SET SESSION CONSTRAINT request executes, the session has a label of *secret* and *nato*. Suppose that the session initiated by *p/s* is used to insert 3 rows into the table named *inventory*. The INSERT requests used to insert the rows into *inventory* are as follows.

```
INSERT INTO inventory VALUES (1212, 90505, 'Widgets',,);
INSERT INTO inventory VALUES (12122, 90504, 'Buggy Whips',,);
INSERT INTO inventory VALUES (12126, 90501, 'Whip Sockets',,);
```

The last two positional values are generated by the INSERT constraint UDFs, not by the session. The column values for the rows after these INSERT requests complete are as follows.

inventory				
col_1	col_2	col_3	classification_level	classification_category
1212	90505	Widgets	3	'0100000000000000'XB
12122	90504	Buggy Whips	3	'0100000000000000'XB
12126	90501	Whip Sockets	3	'0100000000000000'XB

Example: Changing the Row-Level Security Category for a Session

User *arn_anderson* logs on. The resulting session has a row-level security label consisting of an *unclassified* level and *nato* category. As soon as the session is established *arn_anderson* changes the category to *norway*.

```
SET SESSION CONSTRAINT = classification_category (norway) ;
```

After the SET SESSION CONSTRAINT request executes the session has a label of *unclassified* and *norway*.

Assume that later on, the session initiated by *arn_anderson* wanted to read one of the 3 rows from *inventory*, so the user submits the following SELECT request.

```
SELECT *
FROM inventory
WHERE col_1=12122;
```

The result of this request would be 0 rows and a value of 'F' returned signifying the that the user credentials failed security policy validation, so the constraint predicate added to the query evaluates to FALSE and the row is eliminated from the read.

Vantage does not return any rows for this request because the level of *unclassified* for *arn_anderson* does not allow him to read secret rows or because his category of *norway* does not allow him to read rows with a category of *nato*.

Example: Loading Tables without User OVERRIDE Privileges

When a user without the OVERRIDE privilege performs an INSERT or UPDATE on an RLS table the system converts the session constraint value(s), defined as byte(n) in the assigned user constraint, to hex code and loads them into the table.

For example, assume that:

1. A BYTE(1) non-hierarchical constraint named Countries is defined with these values:
 - USA: 1
 - UK: 2
 - Canada: 3
2. User U1 is assigned the constraint.

```
CONSTRAINT = Countries (USA, UK, Canada)
```

3. User U1 defines a table to include the Countries constraint column:

```
CT rls_table (x INT, Countries CONSTRAINT);
```

4. The security policy defined in the related INSERT UDF does not alter the session constraint for the user.

At logon, the session constraint value for user U1 is calculated by the system as follows:

Constraint Value	Bit Position	Binary Value
USA:1	1	1
UK:2	2	1
Canada:3	3	1
Not applicable	4	0
	5	0
	6	0
	7	0
	8	0

The system evaluates the assigned user constraints and calculates a binary string to represent each set of non-hierarchical values, in the example above, 11100000, which translates to the hex string 'E0'xb.

If user U1 inserts a row into the table `rls_tbl`, the system automatically enters the calculated hex value 'E0'xb in the Countries CONSTRAINT column for the table.

Resetting the Session Constraint

Users have the option to change the default constraint values available for a session using the SET SESSION CONSTRAINT statement.

Based on the previous example, U1 might reset the session default Countries constraint:

```
SET SESSION CONSTRAINT = Countries (UK, Canada);
```

The session constraint value is changed from the 'E0'xb shown above to '60'xb (hex representation of 01100000). Subsequent inserts during the session default to '60'xb for the Countries constraint column.

Note:

You can display the hex string for the default constraint values using the HELP SESSION CONSTRAINT statement.

Example: Loading Tables with User OVERRIDE Privileges

When a user with OVERRIDE privileges uses an INSERT or UPDATE to load a table that has one or more non-hierarchical constraint columns, the system does not calculate and insert the needed hex values. The user must supply the hex values for each non-hierarchical constraint column as part of the load script.

For example:

1. Calculate the binary string for the bit position that represents the constraint values, as shown in [Example: Loading Tables without User OVERRIDE Privileges](#).
2. Convert the binary string to a hex value.
3. Load the hex values into the source data column(s).
4. Specify the column(s) containing the hex data in the load script.
5. When you load the destination table, the hex values are entered into the constraint columns.

Note:

The database checks all constraint column input data for validity, by internally converting the hex string to binary and checking the indicated bit position against dictionary rows in DBC.ConstraintValues, before inserting them into the target table.

Related Information

- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100
- [ALTER CONSTRAINT](#)
- [CREATE CONSTRAINT](#)

SET SESSION DATABASE

Changes the default database for the session.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Although SET SESSION DATABASE is technically a DCL statement, Vantage treats it as a DDL statement for transaction semantics.

Required Privileges

None.

SET SESSION DATABASE Syntax

```
{ SET SESSION | SS } DATABASE database_name [;]
```

SET SESSION DATABASE Syntax Elements

database_name

Name of the new default database for the remainder of the current session.

Usage Notes

Session Database and Called Procedures

Stored procedures retain the default database that was in effect when they were created. They do not reflect the default database in effect for the session in which they are called unless it is the same as the database in effect when the procedure was created.

Example

Assume your default database is *personnel*. The following example shows how to change the default database to *accounting* for the current session.

```
SET SESSION DATABASE accounting;
```


SET SESSION DATEFORM

Changes the default DATE format in field mode and the default format for importing and exporting DATE values for the session.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Although SET SESSION DATEFORM is technically a DCL statement, Vantage treats it as a DDL statement for transaction semantics.

Required Privileges

None.

SET SESSION DATEFORM Syntax

```
{ SET SESSION | SS } DATEFORM= [ ANSIDATE | INTEGERDATE ] [;]
```

SET SESSION DATEFORM Syntax Elements

ANSIDATE

A DATE format of CHARACTER(10) for the session, formatted as 'YYYY-MM-DD' for importing and exporting DATE values.

INTEGERDATE

Encoded integers for importing and exporting DATE values.

When the session DATEFORM is INTEGERDATE, the default DATE format in field mode is 'YY/MM/DD' for date columns created and date constants in character form.

You can use the `tdlocaledef` utility to change the 'YY/MM/DD' default DATE format. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Usage Notes

Session Dateform and Called Procedures

Stored procedures retain the default database that was in effect when they were created. They do not reflect the default database in effect for the session in which they are called unless it is the same as the database in effect when the procedure was created.

SET SESSION DATEFORM Examples

Example: Changing the Session Dateform From ANSIDATE to INTEGERDATE

You need to change the DATEFORM setting for the current session from ANSI session mode to Teradata session mode so you can run a legacy application that does not support ANSI dates. Use the following statement:

```
SET SESSION DATEFORM = INTEGERDATE;
```

Example: Changing the Session Dateform from INTEGERDATE to ANSIDATE

To change the DATEFORM setting back to ANSIDATE, use the following statement:

```
SET SESSION DATEFORM = ANSIDATE;
```

SET SESSION DEBUG FUNCTION

Identifies the UDF, UDM, or stored procedure to be run in debug mode the next time the function, method, or stored procedure is invoked.

For more information, see [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#), [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#), [CREATE METHOD](#), and the information about C/C++ command-line debugging for UDFs in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the EXECUTE FUNCTION and the DROP FUNCTION privilege on the function or stored procedure you want to debug.

You must have the UDTMETHOD and the UDTUSAGE privileges for the method you want to debug.

SET SESSION DEBUG FUNCTION Syntax

```
{ SET SESSION | SS } DEBUG {  
  FUNCTION [ database_name. | user_name. ] function_name |  
  PROCEDURE [ database_name. | user_name. ] procedure_name |  
  METHOD [ database_name. | user_name. ] method_name  
}  
{ ON | OFF } [;]
```

SET SESSION DEBUG FUNCTION Syntax Elements

DEBUG FUNCTION

Keywords to precede function name.

DEBUG PROCEDURE

Keywords to precede stored procedure name.

DEBUG METHOD

Keywords to precede method name.

database_name

Name of database, if other than the current database.

user_name

Name of user, if other than the current user.

function_name

Name of function to debug.

procedure_name

Name of stored procedure to debug.

method_name

Name of method to debug.

ON

Set debugging on.

OFF

Set debugging off.

Usage Notes

You can only debug one function per session at a time. After you set debugging on for a function, debugging remains enabled until one of the following occurs:

- Debugging is turned off for the function.
- Debugging is turned on for another function with a subsequent SET SESSION DEBUG FUNCTION statement.

- The session terminates.

After you set debugging on for a function and a query invokes the function, the function is passed to the debugger. If a debugger is not active, the query cannot complete until the debugger is started, attaches to session running the query, and allows all instances of the function being debugged to complete.

SET SESSION DEBUG FUNCTION Examples

Example: Enabling Debugging for the Session

```
SET SESSION DEBUG FUNCTION function_name ON;
```

Example: Disabling Debugging for the Session

```
SET SESSION DEBUG FUNCTION function_name OFF;
```

SET SESSION DOT NOTATION

Sets the session response for dot notation query results that return a list of values.

To specify the system default, use the DBS Control setting, DotNotationOnErrorCondition. See *Teradata Vantage™ - Database Utilities*, B035-1102.

Required Privileges

None.

SET SESSION DOT NOTATION Syntax

```
SET SESSION DOT NOTATION {
  DEFAULT |
  LIST |
  NULL |
  ERROR
} ON ERROR [;]
```

SET SESSION DOT NOTATION Syntax Elements

DEFAULT

For pre-16.0 dot notation query results that include a list, a warning and an error message appears. Otherwise, the list of values is returned.

LIST

Returns the list of values for dot notation query results that include a list of values.

NULL

Returns null for pre-16.0 dot notation query results.

ERROR

Returns an error for pre-16.0 dot notation query results.

Example: Default Response for 15.0 and 15.10 Dot Notation

The default response for 15.0 and 15.10 is to return an error for dot notation query results that include a list of values.

```
SET SESSION DOT NOTATION DEFAULT ON ERROR;
SELECT CAST('{ "a":1, "a":2 }' as JSON).a;
*** Warning: 7548 More than one result per JSON instance found.
> *** ERROR MULTI RESULT **
```

SET SESSION FOR ISOLATED LOADING

Enables or disables isolated loading for DML operations for the session.

For information on the CREATE TABLE WITH ISOLATED LOADING option, see [isolated loading](#)

For information on performing DML operations with isolated loading, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

None.

SET SESSION FOR ISOLATED LOADING Syntax

```
{ SET SESSION | SS } FOR [NO] [CONCURRENT] ISOLATED LOADING [;]
```

SET SESSION FOR ISOLATED LOADING Syntax Elements**FOR ISOLATED LOADING**

DML operations can be performed as a concurrent load isolated operation on a table that is defined with load isolation. This is the default.

NO

DML operations are not performed as concurrent load isolated operations unless explicitly overridden by a DML statement that includes the WITH ISOLATED LOADING option.

CONCURRENT

Optional keyword that can be included for readability.

SET SESSION FOR ISOLATED LOADING Examples**Example: Enabling Concurrent Isolated Loading for the Session**

```
SET SESSION FOR ISOLATED LOADING;
```

Example: Disabling Concurrent Isolated Loading for the Session

```
SET SESSION FOR NO ISOLATED LOADING;
```

SET SESSION FUNCTION TRACE

Enables function trace output for debugging external user-defined functions and external SQL procedures for the current session.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Although SET SESSION FUNCTION TRACE is technically a DCL statement, Vantage treats it as a DDL statement for transaction semantics.

Required Privileges

None.

SET SESSION FUNCTION TRACE Syntax

```
SET SESSION FUNCTION TRACE
  { trace_enabling_specification | OFF } [;]
```

trace_enabling_specification

```
USING mask_string FOR [TRACE] TABLE [ database_name. ] table_name
```

SET SESSION FUNCTION TRACE Syntax Elements

trace_enabling_specification

Output tracing is enabled for all user-defined functions or external SQL procedures the current user performs in subsequent DML requests in the current session.

The UDF trace output is written to the materialized global temporary trace table defined by *table_name*.

See [CREATE USER](#) for more information about defining default character data types for users.

mask_string

An arbitrary character string to be interpreted by any UDF performed in the current session.

An arbitrary character string to be interpreted by any UDF performed by the current session.

The string is limited to 256 logical characters in the character set defined for the user. This means, for example, that the string has a maximum length of 256 bytes if the UDF was created under a Latin character data type, but has a maximum length of 512 bytes if created under a Unicode character data type.

UDFs convert this string to the default character data type in effect when the function was created, not the default character data type defined for the user for the session.

All necessary conversions are done by the system. If, for example, a UDF was created with Unicode, but is called by a user with a session character set of Latin, the trace string is translated to Latin before it is passed to the UDF. If a pre-conversion trace string contains characters that have no equivalent in the target character set, the system returns an error. See SQL External Routine Programming for more information.

database_name

Name of the containing database or user for *table_name*.

The database name is only required when the table is contained in a database other than the current default database for the session.

table_name

Name of a global temporary trace table into which function trace data is to be written.

See the SET SESSION FUNCTION TRACE “Function Trace Output Table” topic in *Teradata Vantage™ - SQL Data Definition Language Detailed*

Topics, B035-1184 and [CREATE GLOBAL TEMPORARY TRACE TABLE](#) for further information.

OFF

Currently active UDF tracing is disabled for the current user in the current session, so the option to output anything for a function is turned off.

SET SESSION FUNCTION TRACE Example

The following example enables function trace processing for the current session, writing the trace data into the global temporary trace table named *udf_diagnostics*.

```
SET SESSION FUNCTION TRACE USING 'diag,3'
FOR TABLE udf_diagnostics;
```

Related Information

- [ALTER FUNCTION](#)
- [CREATE FUNCTION and REPLACE FUNCTION \(External Form\)](#)
- [CREATE GLOBAL TEMPORARY TRACE TABLE](#)
- [CREATE PROCEDURE and REPLACE PROCEDURE \(External Form\)](#)
- [DROP FUNCTION](#)
- [HELP FUNCTION](#)
- [SHOW object](#)

For more information about best practices for coding UDFs and external SQL procedures, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

SET SESSION JSON IGNORE ERRORS

Enables or disables the validation of JSON data on INSERT operations.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

None.

SET SESSION JSON IGNORE ERRORS Syntax

```
{ SET SESSION | SS } JSON IGNORE ERRORS { ON | OFF } [;]
```

SET SESSION JSON IGNORE ERRORS Syntax Elements

JSON IGNORE ERRORS

Specifies whether to validate JSON data on INSERT operations.

ON

Disables the validation of JSON data for the current session.

OFF

Enables the validation of JSON data for the current session.

Example: Disabling JSON Data Validation

The following example disables the validation of JSON data for the current session.

```
SET SESSION JSON IGNORE ERRORS ON;
```

Related Information

- [HELP TYPE](#)
- [SHOW object](#)

For more information about using the JSON data type, see *Teradata Vantage™ - JSON Data Type*, B035-1150.

SET SESSION SEARCHUIFDBPATH

Sets the database search path for the SCRIPT execution in the SessionTbl.SearchUIFDBPath column.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

None.

SET SESSION SEARCHUIFDBPATH Syntax

```
SET SESSION SEARCHUIFDBPATH = [ database_name | user_name ][,...] [;]
```

SET SESSION SEARCHUIFDBPATH Syntax Elements

database_name

Name of database or list of database names containing the user-installed files (UIFs).

user_name

Name of user or list of user names containing the user-installed files (UIFs).

Example: Setting the Database UIF Search Path

This statement checks if the databases specified in the search path exist.

```
SET SESSION SEARCHUIFDBPATH = DB1, DB2;
*** Search Databases for User Installed Files set successfully
```

SET SESSION TRANSACTION ISOLATION LEVEL

Changes the default transaction isolation level read-only semantics for the current session.

ANSI Compliance

SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL is ANSI SQL:2011-compliant with extensions.

ANSI SQL does not support the abbreviations RU and SR. They are Teradata extensions.

Other SQL dialects support similar non-ANSI standard statements with names such as the following:

- SET CURRENT ISOLATION
- SET ISOLATION
- SET TRANSACTION
- SET SESSION CHARACTERISTICS

Although SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL is technically a DCL statement, Vantage treats it as a DDL statement for transaction semantics.

Required Privileges

None.

SET SESSION TRANSACTION ISOLATION LEVEL Syntax

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL isolation_level [;]
```

isolation_level

```
{ READ UNCOMMITTED | RU | SERIALIZABLE | SR }
```

SET SESSION TRANSACTION ISOLATION LEVEL Syntax Elements

isolation_level

Default transaction isolation level in force for the current session.

The valid transaction isolation level options are the following.

READ UNCOMMITTED

RU

Sets the read-only locking severity for the current session to ACCESS for SELECT operations embedded within DELETE, INSERT, MERGE, and UPDATE requests if the DBS Control parameter AccessLockForUncomRead is set to TRUE; otherwise, the read-only locking severity for such requests remains READ.

For outer SELECT requests and SELECT subqueries that are not embedded within DELETE, INSERT, or UPDATE requests, the default read-only locking severity is always READ.

The read-only locking severity for SELECT operations embedded within DELETE, INSERT, MERGE, or UPDATE requests remains READ when the transaction isolation level is set to READ UNCOMMITTED only if the setting of the DBS Control parameter AccessLockForUncomRead is FALSE.

If AccessLockForUncomRead is set to TRUE, the default read-locking severity lock for these requests is ACCESS.

RU and READ UNCOMMITTED are synonyms.

SERIALIZABLE

SR

Sets the read-only locking severity for all SELECT requests made against nontemporal tables, whether they are outer SELECT requests or subqueries, in the current session to READ regardless of the setting for the DBS Control parameter AccessLockForUncomRead.

Serializability violations can occur with DML operations that use Current semantics or that specify the CURRENT VALIDTIME qualifier whether the transaction isolation level is set to SERIALIZABLE or not. See *Teradata Vantage™ - Temporal Table Support*, B035-1182 for details of when and why this can occur and how you can work around it.

SR and SERIALIZABLE are synonyms.

Note that transaction isolation levels affect locking severities, not locking levels (see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for descriptions of locking severities and locking levels).

SET SESSION TRANSACTION ISOLATION LEVEL Examples

Example: Setting the Default Session Isolation Level To READ UNCOMMITTED

The following equivalent requests set the isolation level for the current session to READ UNCOMMITTED.

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL RU;
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ
UNCOMMITTED;
```

Example: Setting the Default Session Isolation Level Back To SERIALIZABLE

The following equivalent requests set the isolation level for the current session back to SERIALIZABLE.

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SR;
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL
SERIALIZABLE;
```

Example: Failure Conditions

Note that both syntax and resolver errors return Failure responses for this statement in Teradata session mode.

The following request returns a Failure response because SU is not a valid isolation level.

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SU;
*** Failure 3706 Syntax error: expecting isolation level.
```

The following Teradata session mode transaction returns a Failure response because SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL is a DDL statement, but it is not the last request in the transaction, and the only statements that are valid following a DDL statement in an explicit transaction are END TRANSACTION, ABORT/ROLLBACK, or a NULL statement.

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SR;
SELECT *
```

```
FROM table_1;
*** Failure 3932 Only an ET or null statement is legal after a DDL
Statement.
```

Note that in ANSI session mode, these requests would return an Error rather than a Failure. The system would then roll the erring request back.

You can correct this transaction in three different ways.

- Run the SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL request as a single-statement request in Teradata session mode as an implicit transaction.
- Commit or roll back the containing explicit Teradata session mode transaction immediately after issuing the SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL request with one of the following.
 - END TRANSACTION statement.
 - ABORT or ROLLBACK statement.
- Commit or roll back the containing ANSI session mode transaction immediately after issuing the SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL request with one of the following.
 - COMMIT statement.
 - ABORT or ROLLBACK statement.

Related Information

See the description of transaction isolation levels in *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 for further information about transaction isolation.

See *Teradata Vantage™ - Temporal Table Support*, B035-1182 for information about the possible ways that transactions involving temporal tables can violate serializability and information about how to work around those issues.

For further information about the AccessLockForUncomRead parameter, see the description of the DBS Control utility in *Teradata Vantage™ - Database Utilities*, B035-1102.

SET QUERY_BAND

Sets or removes a query band for the current session or transaction.

Middle-tier applications, through which end users access Vantage, can be programmed to use the SET QUERY_BAND statement to attach metadata to end user SQL queries.

You can also use query band names to track activity using Viewpoint and to filter requests for execution in addition to their use for Teradata Active System Management.

Vantage logs query bands in DBQL, providing opportunities for subsequent analysis of queries.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

IF you ...	THEN you must be granted this privilege on your trusted user name ...
do not have a need to submit SET QUERY_BAND requests to set a Proxy User	none.
need to submit SET QUERY_BAND requests to set a proxy user	GRANT CONNECT THROUGH to a proxy user. See <i>Teradata Vantage™ - SQL Data Control Language</i> , B035-1149.
must SET QUERY_BAND TVSTEMPERATURE = VERYHOT	EXECUTE on DBC.VHCTRL.

SET QUERY_BAND Syntax

```
SET QUERY_BAND = { 'band_specification [...] ' | NONE } [ UPDATE ]
FOR { SESSION [VOLATILE] | TRANSACTION } [;]
```

band_specification

```
pair_name = pair_value;
```

SET QUERY_BAND Syntax Elements

pair_name

Name component of a query band specification.

The maximum size for each *pair_name* is 128 UNICODE characters. For more information about database object names, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

The number of *pair_name=pair_value* pairs is limited to the maximum length of the string, which is 4,096 UNICODE characters, including pad characters.

You must enclose the set of *name=value* pairs for the query band with APOSTROPHE characters. If an APOSTROPHE character is embedded within a pair name, you must type it twice, to escape it. Otherwise, the system interprets it as a *pair_name=pair_value* string terminator.

pair_name cannot contain any of the following characters.

- EQUALS SIGN (=)

- SEMICOLON (;)
- a null

Note:

Do not specify reserved pair names. For a list of reserved query band names, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

pair_value

Value component of a query band specification.

pair_value can be up to 256 VARCHAR UNICODE characters in length.

The number of *name=value* pairs is limited to the maximum length of the string, which is 2,048 UNICODE characters, including pad characters.

pair_value cannot contain any of the following characters.

- SEMICOLON (;)
- a null

If an APOSTROPHE character is embedded within a pair value, you must type it twice, to escape it. Otherwise, the system interprets it as a *pair_name =pair_value* string terminator.

Note:

Do not specify reserved pair values. For a list of reserved query band values, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

NONE

The previously specified query band is to be removed for the current session or transaction.

You cannot specify both the NONE option and the UPDATE option within the same request.

An APOSTROPHE-enclosed null string is equivalent to NONE.

UPDATE

Update the current query band *name:value* pairs as follows.

- If the query band name matches a name in the current query band, then Vantage replaces the value.
- If the query band name does not match a name in the current query band, then Vantage adds the *name:value* pair.

The only way to remove a name:value pair from a query band is to replace the entire query band.

If an update causes the new query band to exceed the maximum allowable length, Vantage returns an error to the requestor.

Note that a name can be set to a zero-length string by specifying ‘ *pair_name* =; ’.

If you do not specify UPDATE, then the entire current query band is replaced by the new query band.

You cannot specify both the NONE option and the UPDATE option within the same request.

FOR SESSION

The set of *name:value* pairs applies to the current session.

You can set a session query band concurrently with an existing transaction query band.

Session query bands are stored in DBC.SessionTbl and are recovered after a system reset. The system populates additional columns in DBC.SessionTbl for a trusted session and uses the information in these session proxy columns to recover a trusted session after a system reset.

The query band for the session remains in effect until the current session ends or until you issue a:

- SET QUERY_BAND FOR SESSION request to reset the query band.
- SET QUERY_BAND NONE FOR SESSION request.

You can use SET QUERY_BAND FOR SESSION:

- In a macro definition, as the single request in the macro.
- With Teradata Parallel Transporter.

You cannot use SET QUERY_BAND FOR SESSION in:

- Multistatement requests.
- SQL procedure definitions.

All reserved query bands associated with a load or unload utility should be set FOR SESSION to enable their operation across a restart. For more information about reserved query bands, see SET QUERY_BAND in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 and *Teradata Vantage™ - Teradata® Virtual Storage*, B035-1179.

VOLATILE

The system does not update the *queryband* column of DBC.SessionTbl and the FOR SESSION request is equivalent to a FOR TRANSACTION request.

NOTICE

Use caution if you mix SET QUERY_BAND requests. For example, an application can set some *name:value* pairs FOR SESSION that are saved in DBC.SessionTbl. If you then add query band *name:value* pairs using the SET QUERY_BAND VOLATILE option, the query band in the DBC.SessionTbl will no longer match the current session query band. In this case, the system restores the query band saved in DBC.SessionTbl on a restart.

FOR TRANSACTION

The query band set specified by the set of *name:value* pairs applies to the current transaction.

You can set a transaction query band concurrently with an existing session query band.

Only one transaction query band can be active at a time.

As each sequential SET QUERY_BAND FOR TRANSACTION request within a transaction is executed, its query band replaces the current transaction query band.

SET QUERY_BAND FOR TRANSACTION must be the first request specified if it is stipulated within a multistatement request. In a multistatement request, Vantage applies only one transaction query band to all statements in the request. Teradata Active Systems Management (TASM) classification occurs only once for all statements in a multistatement request.

If a proxy user is set in a SET QUERY_BAND statement in a multistatement request, Vantage applies the proxy user privileges to all statements in the request except when the transaction query band is specified by a:

- ? parameter in a Java or ODBC program.
- Macro.
- SQL procedure.

For these cases only, Vantage does not apply the trusted session privileges to the statements in the multistatement request. Therefore, you can specify only one SET QUERY_BAND FOR TRANSACTION statement per multistatement request.

The query band for a transaction is not stored in DBC.SessionTbl.

Vantage discards the current transaction query band when the transaction ends, as determined by any of the following SQL transaction-terminating statements:

- ABORT
- COMMIT
- ROLLBACK

For more information about these statements, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Vantage does not restore the current transaction query band after a system restart.

SET QUERY_BAND FOR TRANSACTION is supported for both macros and SQL procedures.

SET QUERY_BAND FOR TRANSACTION is not supported for Teradata Parallel Transporter.

The following process illustrates a use case for query bands in the context of a SET QUERY_BAND FOR TRANSACTION request.

1. A web service, implemented in Java, responds to an HTTP/SOAP (Hypertext Transfer Protocol/ Simple Object Access Protocol) request.
2. The web service uses JTA (Java Transaction API) to begin a user-managed transaction.
3. The web service obtains a JDBC connection from a connection pool managed by the application server.
4. The web service uses JDBC (Java Database Connectivity) to submit the SQL request to establish the query band for the transaction.
5. The web service invokes multiple EJBs (Enterprise Java Beans) to do some work. For example, the first EJB inserts a row into table A, and the second EJB updates some rows in table B. All calls to the EJBs participate in the same transaction, and therefore are identified with the same query band.
6. The web service commits the transaction.

The query band value for the transaction is automatically cleared, so any subsequent SQL requests submitted on the pooled connection have a different query band value.

Usage Notes

About Query Banding and Middle Tier Applications

End users can connect to Vantage through middle-tier (non-Teradata) applications using pooled sessions. You can use the SET QUERY_BAND statement to attach metadata to end user SQL queries that cannot otherwise be associated with the session.

Application users with SQL privileges can submit SET QUERY_BAND statements. For users who cannot submit SQL, you can program the application to inject a SET QUERY_BAND statement containing the needed metadata.

Query Banding for Pooled Sessions

In a pooled session, the middle-tier application logs on to Vantage as a database user and establishes a connection pool. The application has a single set of database privileges. All application end users assume the identity and database privileges of the application.

You can use query banding to:

- Identify end users, who otherwise all appear to Vantage as the application user.
- Record resource accounting information, such as job number, application, or report type.

- Determine session or request level access priority, by using the query bands as workload management rule criteria.

Query Banding for Trusted Sessions

For trusted sessions, the application is set up as a trusted user with an associated set of proxy users. For details about setting up trusted users and proxy users, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

In a trusted session, query banding is required to identify the end user, and can also be used to define user privileges beyond those available to the application user.

The following reserved query bands can be used only in trusted sessions.

Name	Description
ProxyRole	<p>Defines the role to be used within the trusted session. The valid value is the name of a role that has been granted to the proxy user.</p> <p>Note: Proxy roles only apply to application end users that are not also permanent database users. Proxy users who are also permanent database users inherit the permanent user privileges.</p>
ProxyUser	<p>Sets a trusted session to the identity of the proxy user. The valid value is the name of a proxy user that has been granted the CONNECT THROUGH privilege on the currently logged on user. See <i>Teradata Vantage™ - SQL Data Control Language</i>, B035-1149 for the syntax and rules for using GRANT CONNECT THROUGH requests.</p>

For a sample SET QUERY_BAND statement, see the examples beginning with [Example: Making a Proxy User Connection](#).

Also see *Using Query Banding in Teradata Vantage Orange Book*, 541-0007069.

Preventing Unauthorized Use of Query Banding by Proxy Users

If no restrictions are imposed, a proxy user could use a SET QUERY_BAND statement to change the proxy user for the session and possibly make unauthorized access to the database. However, use the GRANT CONNECT THROUGH statement and the WITH TRUST ONLY clause to instruct the database to honor SET QUERY_BAND statements that set or update a proxy user only if they are part of a trusted request. You also must program the application trusted user to flag each request as trusted or not trusted to prevent unauthorized use of SET QUERY_BAND to change of the proxy user for a trusted session.

Creator Status for Objects Created in a Trusted Session

In a trusted session:

- If the proxy user is also a permanent database user, Vantage assigns the name of the permanent user as the creator of any database objects the proxy user creates.
- If the proxy user is not also a permanent database user, Vantage assigns the name of the application user as the creator of any database objects the proxy user creates.

Query Bands and Row-Level Security Constraints

When an application logs on and initiates a pooled session, the row-level security constraints assigned to the application are in effect.

When SET QUERY_BAND is used to assert a proxy user, the row-level security constraints assigned to the proxy user are in effect, regardless of whether these constraints have been assigned directly or through a profile.

SET QUERY_BAND FOR SESSION, Proxy Users, and the Deletion of Volatile and Global Temporary Tables

When a SET QUERY_BAND FOR SESSION request sets, changes, or removes a Proxy User, Vantage also removes all volatile and materialized temporary tables from the session. For materialized global temporary tables, this causes additional locks to be placed on the *DBC.TempTables* and *DBC.TempStatistics* system tables.

Vantage does not remove the volatile and materialized temporary tables when you set a Proxy User in a SET QUERY_BAND FOR TRANSACTION request.

Examples

Example: Setting a Query Band For a Session

The following example sets a query band with two name:value pairs for the session.

```
SET QUERY_BAND = 'org=Finance;report=Fin123;' FOR SESSION;
```

The pairs in the query band string are *org=Finance* and *report=Fin123*.

Example: Making a Proxy User Connection

Following is an example of a proxy user connection for a session.

```
SET QUERY_BAND='PROXYUSER=fred;'
FOR SESSION;
```

Below is an example of a proxy user connection for a transaction.

```
SET QUERY_BAND='PROXYUSER=fred;'
FOR TRANSACTION;
```

Example: Setting the Role for a Trusted Session Using a PROXYROLE name:value Pair

The following example set shows how to change the proxy role in a proxy session using SET QUERY_BAND requests. All of the examples in this set are run in Teradata session mode.

This example uses the PROXYROLE name:value pair in a query band to set the proxy role in a trusted session to a specific role.

```
SET QUERY_BAND='PROXYUSER=fred;PROXYROLE=administration;'
FOR SESSION;
```

This example uses the PROXYROLE name:value pair in a query band to change the proxy role in a trusted session that is set by a session query band.

```
SET QUERY_BAND='PROXYUSER=cuser1;PROXYROLE=role1;' FOR SESSION;
BEGIN TRANSACTION;
  SET QUERY_BAND='PROXYROLE=role2;' FOR TRANSACTION;
  SELECT *
  FROM table;
END TRANSACTION;
```

This example shows that the PROXYROLE value in a query band can be changed multiple times in a transaction.

```
BEGIN TRANSACTION;
  SET QUERY_BAND='PROXYUSER=cuser1;PROXYROLE=role1;' FOR TRANSACTION;
  SELECT *
  FROM table;
  SET QUERY_BAND='PROXYUSER=cuser1;PROXYROLE=role2;' FOR TRANSACTION;
  SELECT *
  FROM table2;
END TRANSACTION;
```

This example also uses the PROXYROLE name:value pair in a query band to change the proxy role in a trusted session that is set by a session query band.

```
SET QUERY_BAND='PROXYUSER=cuser1;PROXYROLE=role1;' FOR SESSION;
SELECT *
FROM table;
SET QUERY_BAND='PROXYUSER=cuser1;PROXYROLE=role2;' FOR SESSION;
SELECT *
FROM table2;
```

This example fails because it attempts to change the role for a transaction trusted session using a session-based query band.

```
BEGIN TRANSACTION;
SET QUERY_BAND='PROXYUSER=cuser1;PROXYROLE=role1;' FOR TRANSACTION;
SELECT *
FROM table;
SET QUERY_BAND='PROXYROLE=role2;' FOR SESSION;
END TRANSACTION;
```

Example: Using the BLOCKCOMPRESSION Reserved Query Band

These examples show how to use the BLOCKCOMPRESSION reserved query band to set the block-level compression characteristics for various tables, subtables, or both for a session or transaction.

Set the BlockCompression query band value to ALL or YES to compress all new subtables loaded for the session or transaction.

These examples set the query band for the current session.

The following statements are equivalent:

```
SET QUERY_BAND = 'BLOCKCOMPRESSION=YES;' FOR SESSION;
```

```
SET QUERY_BAND = 'BLOCKCOMPRESSION=ALL;' FOR SESSION;
```

Setting the BlockCompression query band value to FALLBACK compresses all new fallback subtables, including the fallback copy of eligible LOB subtables, loaded for the session or transaction.

This example sets the query band for the current session:

```
SET QUERY_BAND = 'BLOCKCOMPRESSION=FALLBACK;' FOR SESSION;
```

Set the BlockCompression query band value to NO or NONE to not compress any new subtables loaded for the session or transaction.

These examples set the query band for the current session.

The following statements are equivalent:

```
SET QUERY_BAND = 'BLOCKCOMPRESSION=NO;' FOR SESSION;
```

```
SET QUERY_BAND = 'BLOCKCOMPRESSION=NONE;' FOR SESSION;
```

Set the BlockCompression query band value to ONLYCLOBS to compress all new primary and fallback LOB subtables that are eligible for compression in the session or transaction. Data loaded into all other new subtables during the session or transaction is not compressed.

This example sets the query band for the current transaction.

```
BEGIN TRANSACTION
  SET QUERY_BAND = 'BLOCKCOMPRESSION=ONLYCLOBS;' FOR TRANSACTION;
  ...
END TRANSACTION
```

The ellipsis represents the SQL request set that causes the data load to occur.

Setting the BLOCKCOMPRESSION query band value to WITHOUTCLOBS compresses all new primary and fallback subtables loaded for the session or transaction except for LOB subtables, whose data is not compressed.

This example sets the query band for the current session.

```
SET QUERY_BAND = 'BLOCKCOMPRESSION=WITHOUTCLOBS;' FOR SESSION;
```

Example: Setting TVSTEMPERATURE Query Bands to VERYHOT

This example sets TVSTEMPERATURE query bands. The temperature for the table data blocks is set to VERYHOT for the session.

```
SET QUERY_BAND = 'TVSTEMPERATURE=VERYHOT' FOR SESSION;
```

Example: Setting TVSTEMPERATURE Query Bands

This example sets TVSTEMPERATURE query bands to specify an initial temperature setting for subsets of table data.

The temperature for the primary table data blocks is set to HOT and the temperature for primary table compressible LOB data blocks is set to COLD.

```
SET QUERY_BAND = 'TVSTEMPERATURE_PRIMARY=HOT;
                  TVSTEMPERATURE_PRIMARYCLOBS=COLD; '
FOR SESSION;
```

Example: Setting BLOCKCOMPRESSION and TVSTEMPERATURE Query Bands

Suppose that both block-level compression and temperature-based block-level compression are enabled for your site.

If an unpopulated table using manual compression is loaded, the data blocks are stored with block-level compression. However, if an unpopulated table set for automatic compression is loaded, and the default threshold for temperature-based block-level compression is COLD, its data blocks are not compressed because for an automatic table, the compression implied by the specified temperature and the related DBS Control parameters overrides the specific BLOCKCOMPRESSION clause.

```
SET QUERY_BAND = 'BLOCKCOMPRESSION=YES;
                  TVSTEMPERATURE=VERYHOT; '
FOR SESSION;
```

Example: Setting a Query Band That Specifies the EQUALS SIGN Character as Part of the Value

This example sets a query band whose first value specification contains an = (equals sign) for the current session.

```
SET QUERY_BAND='MSTRReportDate=20100110>=20091231;Projektf=4<56; '
FOR SESSION;
```

Example: Removing a Query Band From a Session

The following example removes the query band for the session.

```
SET QUERY_BAND = NONE FOR SESSION;
```

The following request is equivalent.

```
SET QUERY_BAND = '' FOR SESSION;
```


Example: Query Band UPDATE Examples

This example set demonstrates how to specify and use the query band UPDATE option.

This example uses the UPDATE option to add the *name:value* pairs *city:san diego* and *state:california*.

```
SET QUERY_BAND = 'city=san diego;' UPDATE FOR SESSION;
GETQUERYBAND()
-----
=S> city=san diego;
SET QUERY_BAND = 'state=california;' UPDATE FOR SESSION;
GETQUERYBAND()
-----
=S> state=california;city=san diego;
```

This example uses UPDATE to change the value of the *name:value* pair *city:san diego* to *city:fresno*.

```
SET QUERY_BAND = 'city=Fresno;' UPDATE FOR SESSION;
GETQUERYBAND()
-----
=S> 'city=Fresno;state=california;
```

This example replaces the entire query band by not specifying the UPDATE option.

```
SET QUERY_BAND = 'ID=k31293; Job=payroll;' FOR SESSION;
GETQUERYBAND()
-----
=S> ID=k31293;Job=payroll;
```

Example: Setting a Query Band Using FOR SESSION VOLATILE

This example sets a FOR SESSION VOLATILE query band, verifies that the query band set is what was intended, and then shows that when you specify FOR SESSION VOLATILE, Vantage does not update *DBC.SessionTbl*, which makes the performance of the FOR SESSION request as good as if it were a FOR TRANSACTION request.

```
SET QUERY_BAND = 'city=laquinta;cat=asta;tree=maple;'
FOR SESSION VOLATILE;
*** Set QUERY_BAND accepted.
*** Total elapsed time was 1 second.
```

The query band is now 'city=laquinta;cat=asta;tree=maple;', as the following procedure call demonstrates.

```
CALL syslib.GetQueryBandSP(qb);
*** Procedure has been executed.
*** Total elapsed time was 1 second.
QueryBand
-----
=S> city=laquinta;cat=asta;tree=maple;
```

The following SELECT request shows that there are no rows in *DBC.SessionTbl*, which is the desired result when you specify FOR SESSION VOLATILE.

```
SELECT queryband
FROM DBC.SessionTbl
WHERE queryband IS NOT NULL
AND CHAR_LENGTH(queryband) > 0;
*** Query completed. No rows found.
*** Total elapsed time was 1 second.
```

Example: Using UPDATE With a FOR SESSION VOLATILE Query Band

This example sets a FOR SESSION VOLATILE query band with the UPDATE option, verifies that the query band set is what was intended, and then shows that the existing query band has been updated to add the *name:value* pairs *area=west*, *city=sandiego*, *tree=maple*, and *flower=rose*.

Below is the initial SET QUERY_BAND to set the query band and write to DBC.SessionTbl.

```
SET QUERY_BAND = 'area=west;city=sandiego;tree=maple;flower=rose;' FOR SESSION;
```

Following is the SET QUERY_BAND statement with the UPDATE option.

```
SET QUERY_BAND = 'cat=siamese;dog=akita;'
UPDATE FOR SESSION VOLATILE;
*** Set QUERY_BAND accepted.
*** Total elapsed time was 1 second.
```

The query band is now 'cat=siamese;dog=akita;area=west;city=sandiego;tree=maple;flower=rose', as the following procedure call demonstrates.

```
CALL syslib.GetQueryBandSP(qb);
*** Procedure has been executed.
```

```

    *** Total elapsed time was 1 second.
QueryBand
-----
=S>    cat=siamese;dog=akita;area=west;city=sandiego;tree=maple;flower=rose;

```

There should only be one row in DBC.SessionTbl and it should be for the following query band.

```
'city=sandiego;flower=rose;area=west;cat=asta;tree=maple;'
```

The following SELECT request shows that there is only one row in DBC.SessionTbl, and it is the row that was expected to be there.

```

SELECT queryband
FROM DBC.SessionTbl
WHERE queryband IS NOT NULL
AND CHAR_LENGTH(queryband) > 0;
    *** Query completed. One row found. One column returned.
    *** Total elapsed time was 1 second.
QueryBand
-----
city=sandiego;flower=rose;area=west;cat=asta;tree=maple;

```

Example: Setting a Query Band for the Current Transaction

The following example sets a query band with two name:value pairs for the current transaction.

```
SET QUERY_BAND = 'Document=XY1234;Universe=East;' FOR TRANSACTION;
```

The pairs in the query band string are *Document=XY1234* and *Universe=East*.

Example: Removing a Query Band From the Current Transaction

The following example removes the query band for the current transaction.

```
SET QUERY_BAND = NONE FOR TRANSACTION;
```

The following request is equivalent.

```
SET QUERY_BAND = '' FOR TRANSACTION;
```

Example: SET QUERY_BAND FOR TRANSACTION As Parameterized JDBC Request

The following example shows how you can specify the SET QUERY_BAND FOR TRANSACTION request as a parameterized JDBC request.

```
public static void main(String args[])
{
    // Creation of URL to be passed to the JDBC driver
    String url = "jdbc:teradata://borg";
    // Strings representing a prepared statement
    // and its parameter values
    String ssetqbp = "SET QUERY_BAND = ? FOR TRANSACTION;";
    String txnqb = "cat=asta;tree=palm;flower=rose;";
    ...
    PreparedStatement pstmt = con.prepareStatement(ssetqbp);
    // Set parameter values indicated by ? (dynamic update)
    pstmt.setString(1, txnqb);
}
```

Example: Preventing an Injected SET QUERY_BAND from Changing the Proxy User

You can use the standard application programming interfaces such as CLlV2, JDBC, or .NET to enable an application to designate an SQL request as being either trusted or not trusted.

If the application submits SQL requests created or modified by a client user, you might want to require that all SET QUERY_BAND requests that set a proxy user be submitted in trusted requests.

To do this, you grant the CONNECT THROUGH WITH TRUST_ONLY privilege to the trusted user. See *Teradata Vantage™ - SQL Data Control Language*, B035-1149. With this privilege, Vantage requires that any SET QUERY_BAND requests submitted by a trusted user that either set or remove a proxy user be designated as a trusted request; otherwise, it rejects the request and returns an error to the requestor.

By means of the API it is using, the application can mark requests from user input as being not trusted to prevent the client from injecting a SET QUERY_BAND request that changes the proxy user or proxy role.

To use this feature in JDBC, you must create a JDBC DataSource with the TRUSTED_SQL=ON connection parameter. All requests made on the DataSource are then trusted unless they are downgraded using the {fn teradata_untrusted} escape function.

For requests containing user-input SQL requests, the application can prepend the {fn teradata_untrusted} escape function to the SQL call before the request is passed to the Teradata JDBC Driver. This escape function sets the request to the not trusted state. For example,

```
untrustedSQL = "{fn teradata_untrusted}" untrustedSQL ;
```

Example: Initiating a Trusted Session Using a Transaction Query Band

Using the transaction query band, you can change the proxy user within an explicit Teradata session mode transaction. In this example, proxy user bob initially has the INSERT privilege on *bobs_table*, but then that query band is replaced within the transaction with a new query band that enables proxy user joe to have the INSERT privilege on *joes_table*.

At first glance this example seems trivial, but the capability it demonstrates is very useful for use with Java applets.

```
BEGIN TRANSACTION;
  SET QUERY_BAND = 'PROXYUSER=bob;' FOR TRANSACTION;
  INSERT INTO bobs_table VALUES(1, 2, 3);
  SET QUERY_BAND = 'PROXYUSER=joe;' FOR TRANSACTION;
  INSERT INTO joes_table VALUES(a, b, c);
END TRANSACTION;
```

Example: Setting a Query Band for the Current Transaction for a Multistatement Request in a Trusted Session

If PROXYUSER and PROXYROLE are set in a query band, the system enforces the privileges for the trusted session to validate the privileges of the individual statements in the multistatement request.

In this example, proxy user *pxyuser3* must have the SELECT privilege on *taba*, *tabb*, and *tabc*.

```
SET QUERY_BAND = 'PROXYUSER=pxyuser3;' FOR TRANSACTION
;SELECT * FROM qbuser.taba
;SELECT * FROM qbuser.tabb
;SELECT * FROM qbuser.tabc;
```

Related Information

- [DROP USER](#)
- [SET ROLE](#)
- [SET SESSION](#)
- SET QUERY_BAND in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- GRANT (SQL Form) in *Teradata Vantage™ - SQL Data Control Language*, B035-1149
- GRANT CONNECT THROUGH in *Teradata Vantage™ - SQL Data Control Language*, B035-1149

- REVOKE (SQL Form) in *Teradata Vantage™ - SQL Data Control Language*, B035-1149
- REVOKE CONNECT THROUGH in *Teradata Vantage™ - SQL Data Control Language*, B035-1149
- CURRENT_ROLE in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145
- CURRENT_USER in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145
- *Teradata Vantage™ - Teradata® Virtual Storage*, B035-1179
- *Teradata Vantage™ - Database Utilities*, B035-1102
- *Teradata Vantage™ - SQL Data Control Language*, B035-1149
- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100
- *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - Application Programming Reference*, B035-1090
- *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148
- *Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems*, B035-2417
- *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418
- *Teradata JDBC Driver Reference*, available at <https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html>
- *Teradata® MultiLoad Reference*, B035-2409
- *Teradata® Parallel Transporter User Guide*, B035-2445
- *Teradata® Parallel Transporter Operator Programmer Guide*, B035-2435
- *Teradata® Parallel Transporter Reference*, B035-2436

SET TIME ZONE

Changes the default time zone displacement for a session.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

None.

SET TIME ZONE Syntax

```
SET TIME ZONE {
  LOCAL |
  USER |
  expression |
  [ sign ] 'quotestring' |
```

```
'time_zone_string'
} [;]
```

SET TIME ZONE Syntax Elements

LOCAL

Time zone offset for the session as the system default.

USER

Time zone offset for the session as the defined user default. See [CREATE USER](#).

If a default is not defined for the user, the value for the time zone offset is changed to the system default.

expression

Value of the default time zone for the session to the displacement specified by the numeric value of expression in units of hours.

Vantage implicitly converts the expression, as needed and if allowed, to a time zone displacement, where *expression* represents a simple constant numeric expression.

For a list of the implicit data type conversions Vantage performs on numeric expression data types, see SET TIME ZONE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

sign

Plus (+) sign to indicate positive offset or minus (-) sign to indicate negative offset.

quotestring

Value of the default time zone for the session to the displacement specified by the signed value of *quotestring* as an 'hh' hours or 'hh:mm' hours-to-minute interval value.

For a complete list of the implicit data type conversions Vantage performs on '*quotestring*' values, see SET TIME ZONE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

time_zone_string

Sets the value of the default time zone for the session to the displacement specified by '*time_zone_string*'.

If you specify an explicit non-GMT time zone string, it is passed to a system-defined UDF named GetTimeZoneDisplacement that interprets the string and determines the appropriate time zone displacement for the session (see *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211 for information about this system-defined UDF).

Time zone strings that are expressed in terms of GMT do not enable automatic adjustments for Daylight Saving Time.

To set system-wide time zone values, you must use the DBS Control parameters `SystemTimeZoneHour` and `SystemTimeZoneMinute` or see the `tdlocaledef` utility in *Teradata Vantage™ - Database Utilities*, B035-1102.

See `SET TIME ZONE` in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for a complete list of the supported time zone strings.

SET TIME ZONE Examples

Example: Setting the Time Zone to LOCAL

This example sets the session default time zone displacement to `LOCAL`, which is the system default time zone.

```
SET TIME ZONE LOCAL;
```

Example: Setting the Time Zone to USER

This example sets the session default time zone displacement to `USER`, which is the default time zone for the logged on user who submits the request.

```
SET TIME ZONE USER;
```

Example: Setting the Time Zone Using a Simple INTERVAL HOUR TO MONTH Time Zone Constant Expression Displacement String

This example sets the session default time zone displacement by specifying an `INTERVAL HOUR TO MONTH` time displacement string.

```
SET TIME ZONE INTERVAL '08:00' HOUR TO MINUTE;
```

Example: Setting the Time Zone Using a Simple Constant Expression

The first example sets the time zone displacement ahead by `INTERVAL HOUR '01:00' MINUTE` by specifying a simple constant expression of `+1`.

```
SET TIME ZONE +1;
```

If you then submit the following `SELECT` request with an `AT` clause (see `GetTimeZoneDisplacement` in *Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211), it returns the date for time zone displacement `INTERVAL -'08:00' HOUR TO MINUTE`, or `'08/05/31'`.


```
SELECT CAST((TIMESTAMP '2008-06-01 08:30:00' AT TIME ZONE -8)
           AS DATE AT SOURCE TIME ZONE);
```

The following SELECT request demonstrates the results without an AT clause for the target data type.

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;
SELECT TIMESTAMP '2008-06-01 08:30:00'
       AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;
2008-06-01 08:30:00 AT TIME ZONE INTERVAL -8:00 HOUR TO MINU
-----
2008-05-31 23:30:00-08:00
```

If you submit the same SELECT request without an AT clause for the target data type, or with an AT LOCAL clause, the request returns '08/06/01' for the current session time zone displacement INTERVAL HOUR '01:00' MINUTE.

```
SELECT CAST((TIMESTAMP '2008-06-01 08:30:00'
           AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE) AS DATE);
2008-06-01 08:30:00 AT TIME ZONE INTERVAL -8:00 HOUR TO MINU
-----
08/06/01
```

The following example demonstrates the results returned without an AT clause specified with the SELECT request and with a current session time zone displacement of INTERVAL -'08:00' HOUR TO MINUTE.

```
SET TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;
SELECT CAST((CAST(TIMESTAMP '2008-06-01 08:30:00+01:00'
           AS TIMESTAMP(0))
           AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE) AS DATE);
2008-06-01 08:30:00+01:00 AT TIME ZONE INTERVAL -8:00 HOUR T
-----
08/05/31
```

This example sets the time zone displacement for the current session to INTERVAL '08:00' HOUR TO MINUTE.

```
SET TIME ZONE INTERVAL '08:00' HOUR TO MINUTE;
```

Example: Setting the Time Zone Using a Time Zone String

This example sets the session default time zone displacement by specifying a simple constant time zone string expression. Vantage then passes the string to a system-defined UDF that interprets it and sets the default time zone displacement for the session based on the value of the string.

```
SET TIME ZONE 'America Eastern';
```

If the value of the *time_zone_string* you specify is not valid, the request aborts and Vantage returns the following error to the requestor.

```
*** Failure 7455 Invalid Time Zone specified.
```

SET SESSION UDFSEARCHPATH

Allows you to specify a custom UDF search path. When you execute a UDF, Vantage searches this path first, before looking in the default Vantage search path for the UDF.

If you have created your own UDF that has the same name as a Vantage system UDF, you can use this statement to ensure Vantage executes your UDF, rather than the Vantage system UDF.

For more information about creating UDFs, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

ANSI Compliance

SET SESSION UDFSEARCHPATH is ANSI-2011 SQL-compliant with extensions.

Required Privileges

You must have access to any database you specify as part of the UDF search path.

SET SESSION UDFSEARCHPATH Syntax

```
SET SESSION UDFSEARCHPATH = database_name [,database_name...] FOR FUNCTION =  
udf_name [,udf_name...]
```

SET SESSION UDFSEARCHPATH Syntax Elements

database_name

Any valid database name.

udf_name

Any valid UDF name.

Example: SET SESSION UDFSEARCHPATH

```
CREATE FUNCTION functions.bitor(A INTEGER,B INTEGER,C INTEGER)  
RETURNS INTEGER
```

```
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
COLLATION INVOKER INLINE TYPE 1 RETURN
(CASE WHEN A=0 THEN 0
ELSE 2 END);
```

This SET SESSION UDFSEARCHPATH statement will first search functions, then SYSLIB, and finally TD_SYSFNLIB for bitor:

```
SET SESSION UDFSEARCHPATH = functions, SYSLIB, TD_SYSFNLIB FOR FUNCTION = bitor;
```

HELP SESSION

Displays attribute information for the user of the current session or just the row-level constraint attribute information for the user of the current session.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must either own the session or be the user who is logged on for the session.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against a session.

HELP SESSION Syntax

```
HELP SESSION [CONSTRAINT] [;]
```

HELP SESSION Syntax Elements

CONSTRAINT

Reports only the constraint names and values for the session.

Usage Notes

QueryBand Field Length

The HELP SESSION output includes these query band fields:

- QueryBand
- Session QueryBand

- Transaction QueryBand
- Profile QueryBand

The QueryBand field is a concatenation of the session and transaction query bands. When the DBSControl field MaxSetQueryBandSize is greater than 2K and the host character set export width is greater than one, the HELP SESSION QueryBand field may be truncated to prevent the HELP SESSION output from exceeding the 64KB row size limit. The Session QueryBand, Transaction QueryBand, and Profile QueryBand fields are not truncated. For more information on export width specifications, see [Valid Export Width Specifications](#). For more information on the DBSControl field MaxSetQueryBandSize, see *Teradata Vantage™ - Database Utilities*, B035-1102. To find the full concatenated query band string for the current transaction, session, and profile, use the GetQueryBand function. For more information about the GetQueryBand function, see *Teradata Vantage™ - Application Programming Reference*, B035-1090.

Examples

Example: HELP SESSION for the Current User

The following request reports the attributes of the session for the current user.

```
HELP SESSION;

*** Help information returned. One row.
*** Total elapsed time was 1 second.

      User Name DBC
      Account Name DBC
      Logon Date 15/01/29
      Logon Time 20:01:09
      Current DataBase DBC
      Collation ASCII
      Character Set UTF8
      Transaction Semantics Teradata
      Current DateForm IntegerDate
      Session Time Zone -08:00
      Default Character Type LATIN
      Export Latin 2
      Export Unicode 3
      Export Unicode Adjust 0
      Export KanjiSJIS 1
      Export Graphic 3
      Default Date Format YYYY/MM/DD
      Radix Separator .
      Group Separator ,
```

```

      Grouping Rule 3
Currency Radix Separator .
Currency Group Separator ,
  Currency Grouping Rule 3
    Currency Name US Dollars
      Currency $
        ISOCurrency USD
      Dual Currency Name US Dollars
        Dual Currency $
          Dual ISOCurrency USD
        Default ByteInt format -(3)9
        Default Integer format -(10)9
        Default SmallInt format -(5)9
        Default Numeric format --(I).9(F)
        Default Real format -9.999999999999999E-999
        Default Time format HH:MI:SS.S(F)Z
        Default Timestamp format YYYY-MM-DDBHH:MI:SS.S(F)Z
      Current Role
      Logon Account DBC
      Profile
      LDAP N
      Audit Trail Id DBC
Current Isolation Level SR
  Default BigInt format -(19)9
  QueryBand
  Proxy User
  Proxy Role
  Constraint1Name ?
  Constraint1Value ?
  Constraint2Name ?
  Constraint2Value ?
  Constraint3Name ?
  Constraint3Value ?
  Constraint4Name ?
  Constraint4Value ?
  Constraint5Name ?
  Constraint5Value ?
  Constraint6Name ?
  Constraint6Value ?
  Constraint7Name ?
  Constraint7Value ?
  Constraint8Name ?
  Constraint8Value ?
Temporal Qualifier ANSIQUALIFIER

```

```

Calendar TERADATA
Export Width Rule Set 1112211111222232222211121111112222322222
Default Number format FN9
      TTGranularity LogicalRow
Redrive Participation None
      User Dictionary Name DBC
      User SQL Name DBC
      User UEscape ?
Account Dictionary Name DBC
      Account SQL Name DBC
      Account UEscape ?
Current Database Dictionary Name DBC
      Current Database SQL Name DBC
      Current Database UEscape ?
      Current Role Dictionary Name ?
      Current Role SQL Name ?
      Current Role UEscape ?
Logon Account Dictionary Name DBC
      Logon Account SQL Name DBC
      Logon Account UEscape ?
      Profile Dictionary Name ?
      Profile SQL Name ?
      Profile UEscape ?
Audit Trail Id Dictionary Name DBC
      Audit Trail Id SQL Name DBC
      Audit Trail Id UEscape ?
Proxy User Dictionary Name ?
      Proxy User SQL Name ?
      Proxy User UEscape ?
Proxy Role Dictionary Name ?
      Proxy Role SQL Name ?
      Proxy Role UEscape ?
Constraint1Name Dictionary Name ?
      Constraint1Name SQL Name ?
      Constraint1Name UEscape ?
Constraint2Name Dictionary Name ?
      Constraint2Name SQL Name ?
      Constraint2Name UEscape ?
Constraint3Name Dictionary Name ?
      Constraint3Name SQL Name ?
      Constraint3Name UEscape ?
Constraint4Name Dictionary Name ?
      Constraint4Name SQL Name ?
      Constraint4Name UEscape ?

```

```

Constraint5Name Dictionary Name ?
      Constraint5Name SQL Name ?
      Constraint5Name UEscape ?
Constraint6Name Dictionary Name ?
      Constraint6Name SQL Name ?
      Constraint6Name UEscape ?
Constraint7Name Dictionary Name ?
      Constraint7Name SQL Name ?
      Constraint7Name UEscape ?
Constraint8Name Dictionary Name ?
      Constraint8Name SQL Name ?
      Constraint8Name UEscape ?
      Zone Name
      SearchUIFDBPath ?
      Transaction QueryBand ?
      Session QueryBand ?
      Profile QueryBand ?
      Unicode Pass Through F

```

Note:

If row-level security is disabled or if no constraint applies to the session, the constraint name and constraint value fields are NULL.

Example: HELP SESSION for a KanjiEBCDIC user

A KanjiEBCDIC user issues a HELP SESSION request having an export width of the expected defaults. The HELP SESSION request was entered using BTEQ with the SIDETITLES and FOLDLINE formatting options.

The system reports the following information.

```

User Name KANJI_USER
Account Name KANJI_USER
Logon Date 02/05/23
Logon Time 11:43:09
Current Database KANJI_USER
Collation CHARSET_COLL
Character Set KANJI_EBCDIC5035_0I
Transaction Semantics Teradata
Current DateForm ANSI
Session Time Zone -09:00
Default Character Type UNICODE
Export Latin 1
Export Unicode 2
Export Unicode Adjusted 2
Export KanjiSJIS 1
Export Graphic 2
Export Width Rules 111221111122223222221112111112222322222
Default Date Format YYYY\u5E74MM\u6708DD\u65E5

```

```

Radix Separator .
Group Separator ,
Grouping Rule 3
Currency Radix Separator .
Currency Group Separator ,
Currency Grouping Rule 3
Currency Name Yen
Currency \u00A5
ISOCurrency JPY
Dual Currency Name Yen
Dual Currency \u00A5
Dual ISOCurrency JPY
Default ByteInt format -(3)9
Default Integer format G-(I)9
Default SmallInt format G-(I)9
Default Numeric format G--(I)D9(F)
Default Real format G-9D999999999999999E-999
Default Time format HH\u6642MI\u5206SSDS(F)\u79D2Z
Default Timestamp format
YYYY\u5E74MM\u6708DD\u65E5BHH\u6642MI\u5206SSDS(F)\u79D2Z
Current Role Finance
Logon Account KANJI_USER
Profile Human_Resources
LDAP N
Audit Trail ID ?
Proxy User ?
Proxy Role ?
Current Isolation Level SR
Default BigInt format -(19)9
QueryBand =S> JOB=RPT1;ORG=world;
Constraint1Name ?
Constraint1Value ?
Temporal Qualifier CURRENT VALIDTIME AND CURRENT
TRANSACTIONTIME
Calendar Name ISO
Redrive Participation

```

Example: HELP SESSION for Session Attributes

Assume you have submitted the following request to set the session attribute.

```

SET SESSION SEQUENCED VALIDTIME PERIOD(
DATE '2005-01-01',
DATE '2006-12-31') AND NONSEQUENCED TRANSACTIONTIME;

```

Then you submit a HELP SESSION request. Vantage returns the following report.

```

User Name DBC
Account Name DBC
Logon Date 02/05/11
Logon Time 11:43:09
Current Database user_name
Collation CHARSET_COLL
Character Set LATIN
Transaction Semantics Teradata
Current DateForm ANSI
Session Time Zone -09:00
Default Character Type UNICODE
Export Latin 1
Export Unicode 2
Export Unicode Adjusted 2

```



```

Export KanjiSJIS 1
Export Graphic 2
Export Width Rules 1112211111222232222211121111112222322222
Default Date Format YYYY\u5E74MM\u6708DD\u65E5
Radix Separator .
Group Separator ,
Grouping Rule 3
Currency Radix Separator .
Currency Group Separator ,
Currency Grouping Rule 3
Currency Name Yen
Currency \u00A5
ISOCurrency JPY
Dual Currency Name Yen
Dual Currency \u00A5
Dual ISOCurrency JPY
Default ByteInt format -(3)9
Default Integer format G-(I)9
Default SmallInt format G-(I)9
Default Numeric format G--(I)D9(F)
Default Real format G-9D999999999999999E-999
Default Time format HH\u6642MI\u5206SSDS(F)\u79D2Z
Default Timestamp format
YYYY\u5E74MM\u6708DD\u65E5BHH\u6642MI\u5206SSDS(F)\u79D2Z
Current Role Finance
Logon Account user_name
Profile R&D
LDAP N
Audit Trail ID ?
Proxy User ?
Proxy Role ?
Current Isolation Level SR
Default BigInt format -(19)9
QueryBand ?
Constraint1Name LEVELS
Constraint1Value 2
Constraint2Name GROUPS
Constraint2Value 1100
Constraint3Name CATEGORIES
Constraint3Value '90000000'xb
Temporal Qualifier SEQUENCED VALIDTIME PERIOD(
DATE '2010-01-01',
DATE '2011-12-31' AND NONSEQUENCED TRANSACTIONTIME
Calendar Name Teradata
Redrive Participation

```

Example: Row-Level Security Constraints Only

This example is for the same session reported in [Example: HELP SESSION for Session Attributes](#), but it only requests row-level constraint information.

```

HELP SESSION CONSTRAINT;
*** Help information returned. One row.
*** Total elapsed time was 1 second.
Constraint1Name LEVELS
Constraint1Value 2
Constraint2Name GROUPS
Constraint2Value 1100

```

```
Constraint3Name CATEGORIES
Constraint3Value '90000000'xb
```

Example: Additional Object Name and Title Fields Returned with HELP SESSION

When you execute a HELP SESSION statement, the system returns 2 types of names, and where applicable, titles for database objects:

- Primary names and titles (for example User Name), according to the object definitions that exist in the data dictionary. The system substitutes error characters for any characters that are untranslatable into the session character set.
- A set of 3 additional fields associated with each name or title. For example:

The 3 fields for User Name are:

- User Dictionary Name
- User SQL Name
- User UEscape

The 3 fields for a title (only for applicable objects, for example, tables and columns) are:

- Dictionary Title
- SQL Title
- Title UEscape

These fields help users to manage the differences between the data dictionary representation of name or title and the representation in the session (client) character set.

For the rules governing the content and use of name and title fields, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

For example:

```

                                User Name DBC
Account Name DBC
  Logon Date 13/01/31
  Logon Time 02:12:56
Current DataBase DBC
  Collation ASCII
  Character Set ASCII
Transaction Semantics Teradata
  Current DateForm IntegerDate
  Session Time Zone 00:00
Default Character Type LATIN
  Export Latin 1
  Export Unicode 1
```

```

Export Unicode Adjust 0
  Export KanjiSJIS 1
    Export Graphic 0
      Default Date Format YY/MM/DD
        Radix Separator .
          Group Separator ,
            Grouping Rule 3
Currency Radix Separator .
Currency Group Separator ,
  Currency Grouping Rule 3
    Currency Name US Dollars
      Currency $
        ISOCurrency USD
          Dual Currency Name US Dollars
            Dual Currency $
              Dual ISOCurrency USD
Default ByteInt format -(3)9
Default Integer format -(10)9
Default SmallInt format -(5)9
Default Numeric format --(I).9(F)
  Default Real format -9.999999999999999E-999
  Default Time format HH:MI:SS.S(F)Z
Default Timestamp format YYYY-MM-DDBHH:MI:SS.S(F)Z
  Current Role
    Logon Account DBC
      Profile
        LDAP N
          Audit Trail Id DBC
Current Isolation Level SR
  Default BigInt format -(19)9
    QueryBand
      Proxy User
        Proxy Role
          Constraint1Name ?
          Constraint1Value ?
          Constraint2Name ?
          Constraint2Value ?
          Constraint3Name ?
          Constraint3Value ?
          Constraint4Name ?
          Constraint4Value ?
          Constraint5Name ?
          Constraint5Value ?
          Constraint6Name ?

```

```

        Constraint6Value ?
        Constraint7Name ?
        Constraint7Value ?
        Constraint8Name ?
        Constraint8Value ?
    Temporal Qualifier CURRENT VALIDTIME AND CURRENT
                        TRANSACTIONTIME
        Calendar TERADATA
    Export Width Rule Set 1112211111222232222211121111112222322222
    Default Number format FN9
        TTGranularity LogicalRow
    Redrive Participation None
    User Dictionary Name DBC
        User SQL Name DBC
        User UEscape ?
    Account Dictionary Name DBC
        Account SQL Name DBC
        Account UEscape ?
    Current Database Dictionary Name DBC
        Current Database SQL Name DBC
        Current Database UEscape ?
    Current Role Dictionary Name ?
        Current Role SQL Name ?
        Current Role UEscape ?
    Logon Account Dictionary Name DBC
        Logon Account SQL Name DBC
        Logon Account UEscape ?
    Profile Dictionary Name ?
        Profile SQL Name ?
        Profile UEscape ?
    Audit Trail Id Dictionary Name DBC
        Audit Trail Id SQL Name DBC
        Audit Trail Id UEscape ?
    Proxy User Dictionary Name ?
        Proxy User SQL Name ?
        Proxy User UEscape ?
    Proxy Role Dictionary Name ?
        Proxy Role SQL Name ?
        Proxy Role UEscape ?
    Constraint1Name Dictionary Name ?
        Constraint1Name SQL Name ?
        Constraint1Name UEscape ?

```

```
Constraint2Name Dictionary Name ?
Constraint2Name SQL Name ?
```

Example: Show SEARCHUIFDBPATH

For user-installed files (UIFs), HELP SESSION displays the SEARCHUIFDBPATH.

```
HELP SESSION;
*** Help information returned. One row.

      User Name  DBC
Account Name  DBC

      (additional output)

SearchUIFDBPath  DB1, DB2
```

Example: HELP SESSION with Unicode Pass Through Set

HELP SESSION output includes the Unicode Pass Through attribute setting:

Setting	Unicode Pass Through
S	On for the session
F	Off for the session.

The following output shows that Unicode Pass Through is on for the session.

The output is abbreviated.

```
HELP SESSION;

*** Help information returned. One row.
*** Total elapsed time was 1 second.

      User Name  DBC
Account Name  DBC
Logon Date   16/07/07
Logon Time   16:01:51
Current DataBase  DBC
Collation    ASCII
Character Set ASCII
Transaction Semantics  Teradata
```

```

Current DateForm IntegerDate
Session Time Zone -07:00
Default Character Type LATIN
Export Latin 1
Export Unicode 1
Export Unicode Adjust 0
Export KanjiSJIS 1
Export Graphic 0
Default Date Format YY/MM/DD

...

Zone Name
SearchUIFDBPath ?
Transaction QueryBand ?
Session QueryBand ?
Profile QueryBand ?
Unicode Pass Through S

```

Related Information

- *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142
- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Temporal Table Support*, B035-1182

Logging Statements

BEGIN LOGGING

Starts the auditing of SQL requests that attempt to access data.

The system creates a log entry when it checks privileges on the SQL request. For a detailed description of access logging options and use cases, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Other SQL dialects support similar non-ANSI standard statements with names such as START LOGGING.

Required Privileges

You must have the EXECUTE privilege on the *DBC.AccLogRule* macro to use the BEGIN LOGGING statement to initiate access logging on database objects and operations.

Privileges Granted Automatically

None.

Prerequisites

When you initially set up logging, before you can use the BEGIN LOGGING statement, you must run the DIPACC script to create the DBC.ACCLogRule macro, which allows you to create logging rules in the DBC.AccLogRuleTbl using BEGIN LOGGING statements. For more information, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

BEGIN LOGGING Syntax

```
BEGIN LOGGING [DENIALS] [WITH TEXT] ON
  [ logging_frequency ]
  [ FOR CONSTRAINT constraint_name ]
  { ALL | operation [,...] | GRANT }
  [ BY user_name [,...] ]
  [ ON item_list ] [;]
```

logging_frequency

```
{ FIRST | LAST | FIRST AND LAST | EACH }
```

item_list

```
{ AUTHORIZATION authorization_name |  
  DATABASE database_name |  
  USER user_name |  
  { TABLE | VIEW | MACRO | PROCEDURE | FUNCTION | FUNCTION MAPPING  
  | TYPE }  
  [ database_name. | user_name. ] object_name  
} [, ...]
```

BEGIN LOGGING Syntax Elements

DENIALS

Entries should be made only when statement execution fails because the user did not have the privilege set necessary to perform the request.

DENIALS is applied to only those actions listed in the BEGIN LOGGING request that contains it.

For example, two BEGIN LOGGING requests can specify the same object, user, and action, but different frequency and DENIALS options. This allows the user to log all denials, but only the first successful use of a privilege.

If this option is not specified, a log entry is made if the privilege check either fails or succeeds.

Note:

You cannot log DENIALS for DELETE, INSERT, SELECT, or UPDATE operations on row-level security-protected tables because the inability of users to access a row is due to row-level security enforcement by the constraint UDF rather than being the result of a normal database privilege check.

Also see the row-level security section for the *operation* variable later in this table.

WITH TEXT

The text of the request that caused the log entry is to be saved in the log.

If two BEGIN LOGGING requests specify the same user, object, and action, and one of the requests specifies the DENIALS option, then WITH TEXT can be specified on either one of the requests, on both requests, or on neither request.

- If you specify WITH TEXT only on the request that also specifies DENIALS, Vantage saves the text only when the entry is created as a result of a denial.
- If you specify WITH TEXT either without DENIALS or on both requests, Vantage always saves the text.
- If you specify WITH TEXT on END LOGGING, Vantage does not save the text for the specified actions or the DENIALS specification, but the logging frequency is not affected.

logging_frequency

The frequency with which to log events.

A log entry is made for either the first, the last, the first and last, or each time during any session that the specified action is attempted against the specified object.

If logging is already initiated for an action on an object, a subsequent request to begin logging for the same object, action, user, and DENIALS specification causes only the current frequency of logging to be changed to whatever the new request specifies.

FIRST

A log entry is made the first time during any session that the specified action is attempted against the specified object.

The default is FIRST.

LAST

A log entry is made the last time during any session that the specified action is attempted against the specified object.

FIRST AND LAST

A log entry is made the first and last time during any session that the specified action is attempted against the specified object.

The only valid combination of logging frequencies is FIRST AND LAST.

EACH

A log entry is made each time during any session that the specified action is attempted against the specified object.

The only valid options for auditing row-level security enforcement are FIRST and EACH.

FOR CONSTRAINT *constraint_name*

The logging rule is for enforcement of the row-level security *constraint_name*.

You cannot log GRANT requests when you specify the FOR CONSTRAINT option.

For information on logging operations that are enforced by security constraints, see the *operation* entry later in this table.

ALL

A log entry is to be made when any of the *operations* listed in the following row of this table is attempted against the specified object.

Note that both the name and ID for each column are logged when you log privileges to INSERT, REFERENCES, SELECT, and UPDATE.

If logging has begun for ALL actions on an object, a request to begin or end logging for an action will change logging activity only for the specified action.

The ALL option does not include logging of the following actions because they do not apply to a specific object.

- CREATE PROFILE
- CREATE ROLE
- DROP PROFILE

The ALL option does not apply to the NONTEMPORAL privilege because Vantage defines a BEGIN LOGGING rule on it by default (see *Teradata Vantage™ - Temporal Table Support*, B035-1182 for details).

The following restrictions apply to the logging of UDTs:

- UDTTYPE and UDTMETHOD are only valid when specified with ON DATABASE SYSUDTLIB or ON USER
- UDTUSAGE is only valid when specified with ON DATABASE SYSUDTLIB, ON USER, or ON TYPE

operation

The types of SQL requests to be logged.

If you do not specify the ALL option, you must specify one or more operations from the following list to define which privilege checks are logged.

- ALTER EXTERNAL PROCEDURE
- ALTER FUNCTION
- ALTER PROCEDURE
- CHECKPOINT

- CREATE AUTHORIZATION
- CREATE DATABASE
- CREATE EXTERNAL PROCEDURE
- CREATE FUNCTION
- CREATE GLOP SET
- CREATE MACRO
- CREATE PROCEDURE
- CREATE PROFILE
- CREATE ROLE
- CREATE TABLE
- CREATE TRIGGER
- CREATE USER
- CREATE VIEW
- DATABASE
- DELETE
- DROP
- DROP AUTHORIZATION
- DROP DATABASE
- DROP FUNCTION
- DROP GLOP SET
- DROP MACRO
- DROP PROCEDURE
- DROP PROFILE
- DROP ROLE
- DROP TABLE
- DROP TRIGGER
- DROP USER
- DROP VIEW
- DUMP
- EXECUTE
- EXECUTE FUNCTION
- EXECUTE PROCEDURE
- GRANT (including GRANTS of row-level security administration privileges)
- INDEX
- INSERT
- MACRO
- PROCEDURE
- REFERENCES

- RESTORE
- ROLLBACK DATABASE
- ROLLFORWARD DATABASE
- SELECT
- TABLE
- TRIGGER
- UDTMETHOD
- UDTTYPE
- UDTUSAGE
- UPDATE
- USER
- VIEW

The following list of operations applies to row-level security logging.

If the log rule is for row-level security enforcement (a FOR CONSTRAINT *constraint_name* specification), but you do not specify an operation type, Vantage includes all of the row-level security operation types in the logging specification.

- DELETE
- INSERT
- OVERRIDE DELETE
- OVERRIDE DUMP
- OVERRIDE INSERT
- OVERRIDE RESTORE
- OVERRIDE SELECT
- OVERRIDE UPDATE
- SELECT
- UPDATE

If you do not want to log all of these operation types (excluding OVERRIDE DUMP and OVERRIDE RESTORE, which are not logged by default), you must specify the each operation type you want to log in the BEGIN LOGGING request.

The system enters one row in the log for each affected operation. Individual refusals of row access are not logged.

If you specify the logging of row-level security operations, then the only valid object types you can specify are row-level security tables, databases or users.

- If you specify a database or user, then Vantage logs all of the tables contained within the specified database or user that contain the constraint specified by *constraint_name*.
- If you specify a table, then that table must contain a row-level security column that matches *constraint_name*.

- If you do not specify an object, the system logs privilege checks on the specified operations for all tables in which the *constraint_name* appears.

When you specify DENIALS for one of the OVERRIDE privileges, Vantage generates a log entry only if a user attempts to select from a row-level security-protected table, under the following conditions:

- The user does not have a session constraint value for the audited constraint.
- and
- The user does not have the OVERRIDE SELECT CONSTRAINT privilege.

The effects of OVERRIDE DELETE, INSERT, and UPDATE are similar.

If you submit the following request,

```
BEGIN LOGGING DENIALS
ON EACH OVERRIDE DUMP CONSTRAINT
FOR CONSTRAINT levels;
```

Vantage generates a log entry if the user does not have the OVERRIDE DUMP CONSTRAINT privilege and tries to archive a row-level security table that has the audited constraint.

The same applies to RESTORE and not having the OVERRIDE RESTORE CONSTRAINT privilege.

GRANT

All requests which GRANT privileges on the objects specified in the ON clause are logged.

Because many users get GRANT privileges by default on objects that they own or create, it is important to monitor the privileges granted to users on sensitive objects and data.

BY *user_name*

The optional list of users to which the logging rules in this BEGIN LOGGING statement apply.

For logging of users authorized privileges by Vantage, the user name must be an existing user.

For logging of users authorized privileges in an LDAP directory (that is, AuthorizationSupported=yes in the authentication mechanism):

- If the user is mapped to a permanent database user object in the directory, specify the mapped database user name.
- If the user is not mapped to a permanent database user object in the directory, logging is not supported.

Absence of the BY *user_name* option specifies all users (those already defined to the system as well as any defined in the future while this logging directive is in effect).

If neither BY *user_name* nor ON keyword *object_name* are specified, then the specified action is logged for all users and objects throughout the system.

ON *item_list*

One or more database objects for which access is to be logged.

If you do not specify the ON *keyword object_name* option in a BEGIN LOGGING statement, the system applies the other logging rules in the statement to all objects.

If you use the ON *keyword* option in a BEGIN LOGGING statement, you can specify logging on up to 20 objects.

The ON *keyword object_name* option does not apply to the CREATE PROFILE, DROP PROFILE, CREATE ROLE, or DROP ROLE *operation* because profiles and roles are system-level objects with no containing user or database.

Note:

When a hash or join index is used in a query plan, the system checks privileges for the base table, not the index, so you cannot specify a hash index or join index in a BEGIN LOGGING statement. Instead, specify logging for the base table.

Each object in the ON clause must be preceded by the keyword that indicates the object type: DATABASE, FUNCTION, FUNCTION MAPPING, MACRO, PROCEDURE, TABLE, TYPE, USER, or VIEW, followed by the object name.

ON DATABASE *qualified_object_name*

You must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

ON FUNCTION *qualified_object_name*

You must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

ON FUNCTION MAPPING *qualified_object_name*

You must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

ON MACRO *qualified_object_name*

You must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

ON PROCEDURE *qualified_object_name*

You must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

ON TABLE *qualified_object_name*

You must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

ON TYPE SYSUDTLIB.UDT_name

For TYPE objects, the specification must always be *SYSUDTLIB.UDT_name*.

ON USER *qualified_object_name*

You must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

ON VIEW *qualified_object_name*

You must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

Usage Notes

About Logging

Each time a session containing criteria specified in a BEGIN LOGGING request attempts to perform a specified action against a specified object, an entry is logged in the DBC.AccLogTbl. A log entry does not indicate that a request was performed; rather, it indicates that the system checked the privileges necessary to perform the request.

Logging of External Users

During access logging, the system identifies:

- Directory users by their authcid, the logon username by which the directory authenticates the user, which the system stores in DBC.SessionTbl.AuditTrailId when it establishes the session.
- Proxy users by the name by which they are initially authenticated when accessing the middle tier application. The access log includes a copy of the query band string for the proxy user.

Determining Object Level for BEGIN LOGGING Request

You should consider the object level used to grant access when specifying the object level for a logging request. For example, when a BEGIN LOGGING request specifies logging at the database level, all the tables in the database are candidates for log entries.

Use Case 1

Assume that:

- database_a contains several tables
- PUBLIC has the INSERT privilege on table_1 only
- Logging is specified as follows:

```
BEGIN LOGGING DENIALS ON FIRST INSERT ON DATABASE "database_a";
```

When an INSERT request is submitted against a table in database_a, the log entries are as follows:

- A granted entry for the first INSERT into table_1.
- A denied entry for the first attempted INSERT into any other table in the database.

Because logging is specified at the database level, additional INSERT statements against other tables in the database as part of a multistatement request log the first INSERT into each table.

Use Case 2

Assume that:

- database_a contains table_1 and table_2
- User access privileges are at the database level
- Logging is specified as follows:

```
BEGIN LOGGING ON FIRST INSERT ON DATABASE "database_a"
```

If rows are inserted into both table_1 and table_2 in the same request, the system makes a single log entry identifying first insert into each table.

If the access privilege is at the database level and the logging specification is at the table level, only actions against the table are considered for logging entries (the absence of an access privilege at the table level does not necessarily result in a log entry).

A single log entry is generated for the table according to the results of privilege checks at the table level and the database level.

Privilege Checks at Table Level and Database Level	Result
Either check succeeds.	System generates a "granted" log entry.
Neither check succeeds.	System does not generate a "denied" log entry because the keyword DENIALS was not specified in the logging rule.

Use Case 3

A logging request specifying FIRST SELECT ON database_a.table_1 causes a log entry to be generated only if an access request is directly on the table, for example:


```
SELECT ... FROM table_1
```

The following actions do not log access on database_a.table_1 because the user does not require direct access to table_1 to perform the request.

- SELECT ... FROM view1_of_table_1
- EXECUTE macro_1

where macro_1 contains the request SELECT ... FROM table_1.

Requirements for Logging Certain Operations and Objects in Combination

The following table is an alphabetical listing of certain operations that have limitations when specified in a BEGIN LOGGING statement.

When specifying logging of this operation	These conditions apply
ALL	Does not include logging of the following operations, because they refer to system level objects that do not support an ON clause: <ul style="list-style-type: none"> • CREATE PROFILE • CREATE ROLE • DROP PROFILE • DROP ROLE
<ul style="list-style-type: none"> • CREATE DATABASE • CREATE MACRO • CREATE PROCEDURE • CREATE TABLE • CREATE VIEW • CREATE USER • DROP DATABASE • DROP USER 	The ON clause can only specify a database or user.
DROP TABLE	The logging also includes ALTER TABLE, which requires the DROP table privilege.
DROP MACRO, DROP PROCEDURE, or DROP VIEW	Also includes logging of REPLACE for the specified object.
DROP, EXECUTE or GRANT	The only operations allowed if the ON clause specifies a macro or procedure.
DUMP, RESTORE or CHECKPOINT	Are not allowed if the ON clause specifies a VIEW.
The single keyword representation of multiple SQL privileges, for example: <ul style="list-style-type: none"> • TABLE 	Logs all the privileges covered by the keyword. For example, specifying TABLE as the logging <i>operation</i> logs CREATE TABLE, ALTER TABLE, and DROP TABLE.

When specifying logging of this operation	These conditions apply
<ul style="list-style-type: none"> • VIEW • MACRO • TRIGGER • PROFILE • USER • PROCEDURE • ROLE • FUNCTION • UDTTYPE (covers UDTUsage/UDTType) • UDTMETHOD (covers UDTUsage/UDTType/UDTMethod) • AUTHORIZATION • GLOP 	For a list of keywords that represent multiple privileges, see <i>Teradata Vantage™ - SQL Data Control Language</i> , B035-1149.
CREATE FUNCTION, DROP FUNCTION, EXECUTE FUNCTION, ALTER FUNCTION, or GRANT	The only operations that can be specified if the ON clause specifies a FUNCTION (internal or external functions).
UDTMETHOD or UDTTYPE	The ON clause must specify DATABASE or USER, and the database or user must be the SYSUDTLIB database.
UDTUSAGE	The ON clause must specify one of the following: <ul style="list-style-type: none"> • DATABASE or USER, and the database or user must be the SYSUDTLIB database. • TYPE
ALTER EXTERNAL PROCEDURE, DROP PROCEDURE, EXECUTE PROCEDURE, and GRANT	The only operations allowed on external procedures.

Logging MODIFY Statements

Privilege checks generate log entries. MODIFY is not a grantable privilege, so the system does not check for the MODIFY privilege. To MODIFY a DATABASE, PROFILE, or USER (except for certain self-referent changes) you must have the DROP privilege on the object to be modified. Specify the DROP operation if you want to log MODIFY requests.

Logging Self-Referent MODIFY USER Statements

By default, database users have the privilege to modify certain parameters in their user definition. When users submit a self-referent MODIFY USER request to change any of the following parameters, the system does not check any privileges:

- AFTER JOURNAL
- BEFORE JOURNAL

- COLLATION
- DEFAULT DATABASE
- DEFAULT JOURNAL TABLE
- PASSWORD
- STARTUP

Because the system does not do a privilege check on such requests, it performs no logging.

Examples

Example: Using BEGIN LOGGING

The following example illustrates the use of BEGIN LOGGING.

```
BEGIN LOGGING WITH TEXT
ON EACH USER, DATABASE, GRANT;
```

Example: Logging UDTs

This example begins logging all of the following attempts that failed because of insufficient privileges.

- ALTER TYPE
- CREATE CAST
- CREATE ORDERING
- CREATE TRANSFORM
- CREATE TYPE
- DROP CAST
- DROP ORDERING
- DROP TRANSFORM
- DROP TYPE
- REPLACE CAST
- REPLACE ORDERING
- REPLACE TRANSFORM
- method invocation
- NEW specification
- UDT reference

```
BEGIN LOGGING DENIALS
ON EACH UDTTYPE;
```

Example: Logging UDTs with SQL Text

This example is similar to [Example: Logging UDTs](#) except that the log entries contain the SQL text.

```
BEGIN LOGGING DENIALS WITH TEXT
ON EACH UDTTYPE;
```

Example: Logging on First UDT Method with SQL Text

This example logs the first of the following operations and logs its SQL text.

- ALTER METHOD
- ALTER TYPE
- CREATE CAST
- CREATE METHOD
- CREATE ORDERING
- CREATE TRANSFORM
- CREATE TYPE
- DROP CAST
- DROP METHOD
- DROP ORDERING
- DROP TRANSFORM
- DROP TYPE
- REPLACE CAST
- REPLACE ORDERING
- REPLACE TRANSFORM
- method invocation
- NEW specification
- UDT reference

```
BEGIN LOGGING WITH TEXT
ON FIRST UDTMETHOD
ON DATABASE SYSUDTLIB;
```

Example: Logging UDTs with Insufficient Privileges in SYSUDTLIB

This example logs all UDT reference, method invocation, and NEW specification attempts that fail because of insufficient privileges on UDTs within the *SYSUDTLIB* database.

```
BEGIN LOGGING DENIALS
ON EACH UDTUSAGE
ON DATABASE SYSUDTLIB;
```

Example: Logging UDTs with Insufficient Privileges on a UDT

This example begins logging all UDT reference, method invocation, and NEW specification attempts that failed because of insufficient privileges on the UDT named *circle*.

```
BEGIN LOGGING DENIALS
ON EACH UDTUSAGE
ON TYPE SYSUDTLIB.circle;
```

Example: Column-Level Discretionary Access Control and Row-Level Security Audit Logging

Suppose you want to audit any attempt by user *TomSmith* to insert a row into the *emp_record* table because such an attempt is a violation of the discretionary access control security policy at your site.

The request to do this auditing looks like this.

```
BEGIN LOGGING DENIALS WITH TEXT
ON EACH INSERT BY TomSmith
ON emp_record;
```

Suppose you would also like to know whenever a user attempts to insert a row that violates the row-level security policy for the target table, which is maintained by a row-level security constraint named *group_membership*.

You cannot execute DENIALS logging for DELETE, INSERT, SELECT, or UPDATE FOR CONSTRAINT requests. Denials as a result of insufficient security credentials to execute a DML request on an row-level security-protected table are not treated as an access denial.

If you attempt to execute a DML request, but do not have the required row-level security privilege, that request is considered to be a DENIAL.

For example, if you attempt to insert a value into a constraint column, you do not have a default session constraint value, and you do not have the OVERRIDE INSERT CONSTRAINT privilege on the table, that attempt would be considered a denial.

For example, Vantage generates an audit row if logging is enabled by the following request.

```
BEGIN LOGGING DENIALS
ON EACH OVERRIDE INSERT
```

```
FOR CONSTRAINT group_membership
ON TABLE securedb.emp_record;
```

Example: Logging Denials for a Row-Level Security Constraint

This example logs denials when a user attempts to access any table that has the *classification_category* constraint, if the user does not have the `OVERRIDE SELECT CONSTRAINT` privilege.

```
BEGIN LOGGING DENIALS
ON EACH OVERRIDE SELECT
FOR CONSTRAINT classification_category;
```

Example: Logging the Denied First Insert Into a Row-Level Security-Secured Table

This example logs the first insert into table *emp_record* in database *secure_db* when access is denied because the session does not have the `OVERRIDE INSERT CONSTRAINT` privilege for constraint *classification_level* or when the `INSERT` security policy UDF for the constraint permits the insert and you either specify a value for the constraint column, or do not have an assigned session constraint value for the constraint.

```
BEGIN LOGGING DENIALS
ON FIRST OVERRIDE INSERT
FOR CONSTRAINT classification_level
ON TABLE secure_db.emp_record;
```

Example: Logging the Denial of an Attempt to Archive a Row-Level Security-Secured Table by a User Not Having Archive Access to the Table

This example logs denials of any attempt to archive table in the *securedb* on which user *TomSmith* does not have `OVERRIDE DUMP` privileges.

```
BEGIN LOGGING DENIALS
ON EACH OVERRIDE DUMP
FOR CONSTRAINT classification_level
BY TomSmith
ON DATABASE securedb;
```

Related Information

- BEGIN LOGGING in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100
- *Teradata Vantage™ - Temporal Table Support*, B035-1182

BEGIN QUERY CAPTURE

To capture Optimizer query plans in XML format at session level.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the INSERT privilege on *QCD_name*.

BEGIN QUERY CAPTURE Syntax

```
BEGIN QUERY CAPTURE
  [ FOR INDEX ANALYSIS ]
  [ WITH { VERBOSE | [ DETAILED ] STATSUSAGE }[,...] ]
  [ INTO qcd_name ]
  AS WORKLOAD workload_name [;]
```

BEGIN QUERY CAPTURE Syntax Elements

FOR INDEX_ANALYSIS

Insertions are made into the following tables of the specified Query Capture Database:

- Query or QryRelX
- Workload
- WorkloadQueries
- Field
- XMLQCD

VERBOSE

Capture verbose EXPLAIN text.

DETAILED

Capture summary statistics details for all of the database objects referenced in the query plan for the request, in addition to capturing Optimizer statistics usage and recommendation information. When you specify DETAILED, the request captures the following statistics details for all the database objects referenced in the query plan:

- StatTimeStamp
- Version
- OrigVersion
- NumColumns
- NumBValues
- NumEHIntervals
- NumHistoryRecords
- NumNulls
- NumAllNulls
- NumAMPs
- NumPNullDistinctVals
- PNullHighModeFreq
- AvgAmpRPV
- HighModeFreq
- NumDistinctVals
- NumRows
- CPUUsage
- IOUsage

STATSUSAGE

Capture Optimizer statistics usage and recommendation information.

qcd_name

The name of the QCD database to be used to capture and store the query plan data.

If you do not specify a QCD database, Vantage uses the QCD database named *TDQCD* by default.

For more information about *TDQCD*, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

workload_name

The name of the workload to which the requests submitted within a BEGIN QUERY CAPTURE session belong.

Example: Using the Default TDQCD Database to Capture and Store a Query Plan in XML Format

This request specifies all of the supported options and uses the default TDQCD database to store the query plan data.

```
BEGIN QUERY CAPTURE AS workload w1_2;
SELECT *
FROM employee
WHERE emp_code > 2;
```

An END QUERY CAPTURE request ends the BEGIN QUERY CAPTURE mode.

```
END QUERY CAPTURE;
```

Example: Capturing a Mix of DML and DDL Requests in XML Format Using BEGIN QUERY CAPTURE

You have created the following *employee* table in the *user_1* database.

```
CREATE TABLE user_1.employee (
  emp_code  INTEGER,
  emp_name  VARCHAR(30),
  dept_code INTEGER,
  salary    FLOAT);
```

Assume that *employee* is populated with 1 million rows and that you collect statistics on the *dept_code* column using the following request.

```
COLLECT STATS COLUMN (dept_code) ON employee;
```

The following BEGIN QUERY CAPTURE request specifies the VERBOSE and DETAILED STATSUSAGE options and captures query plan data in the *user_1_QCD* query capture database.

```
BEGIN QUERY CAPTURE
WITH VERBOSE, DETAILED STATSUSAGE
INTO user_1_QCD
AS WORKLOAD w1_1;
```

This BEGIN QUERY CAPTURE request captures the Object, SQL, Step Details, Verbose EXPLAIN text, statistics details, and information on the statistics used and statistics recommendations for all the following requests.

```

SELECT *
FROM employee
WHERE emp_code > 2;
DELETE FROM employee
WHERE emp_code = 10;
UPDATE employee
SET dept_code = 11
WHERE emp_code = 11;
CREATE TABLE dept (
    dept_code INTEGER,
    dept_name VARCHAR(30),
    manager   VARCHAR(30));
INSERT DEPT(11, 'HR', 'JOHN');

```

An END QUERY CAPTURE request terminates the BEGIN QUERY CAPTURE mode.

```
END QUERY CAPTURE;      /* ends the BEGIN QUERY CAPTURE mode */
```

The following SELECT request returns the query IDs of all the requests that were captured in the QUERY CAPTURE session.

```

SELECT DISTINCT ID
FROM user_1_QCD.Xmlqcd
WHERE WorkLoadName = 'wl_1';

```

Example: Using the Default TDQCD Database to Capture and Store a Query Plan in XML Format

This request specifies all of the supported options and uses the default TDQCD database to store the query plan data.

```

BEGIN QUERY CAPTURE AS workload wl_2;
SELECT *
FROM employee
WHERE emp_code > 2;

```

An END QUERY CAPTURE request ends the BEGIN QUERY CAPTURE mode.

```
END QUERY CAPTURE;
```

Example: Capturing a Mix of DML and DDL Requests in XML Format Using BEGIN QUERY CAPTURE

You have created the following *employee* table in the *user_1* database.

```
CREATE TABLE user_1.employee (
  emp_code  INTEGER,
  emp_name  VARCHAR(30),
  dept_code INTEGER,
  salary    FLOAT);
```

Assume that *employee* is populated with 1 million rows and that you collect statistics on the *dept_code* column using the following request.

```
COLLECT STATS COLUMN (dept_code) ON employee;
```

The following BEGIN QUERY CAPTURE request specifies the VERBOSE and DETAILED STATSUSAGE options and captures query plan data in the *user_1_QCD* query capture database.

```
BEGIN QUERY CAPTURE
WITH VERBOSE, DETAILED STATSUSAGE
INTO user_1_QCD
AS WORKLOAD wl_1;
```

This BEGIN QUERY CAPTURE request captures the Object, SQL, Step Details, Verbose EXPLAIN text, statistics details, and information on the statistics used and statistics recommendations for all the following requests.

```
SELECT *
FROM employee
WHERE emp_code > 2;
DELETE FROM employee
WHERE emp_code = 10;
UPDATE employee
SET dept_code = 11
WHERE emp_code = 11;
CREATE TABLE dept (
  dept_code INTEGER,
  dept_name VARCHAR(30),
  manager   VARCHAR(30));
INSERT DEPT(11, 'HR', 'JOHN');
```

An END QUERY CAPTURE request terminates the BEGIN QUERY CAPTURE mode.

```
END QUERY CAPTURE;      /* ends the BEGIN QUERY CAPTURE mode */
```

The following SELECT request returns the query IDs of all the requests that were captured in the QUERY CAPTURE session.

```
SELECT DISTINCT ID
FROM user_1_QCD.Xmlqcd
WHERE WorkLoadName = 'wl_1';
```

Related Information

- *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142

BEGIN QUERY LOGGING

Starts the logging of database request information, including some database object creation and drop information.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

- EXECUTE privilege on DBC.DBQLAccessMacro.
- To use the MODE option, EXECUTE privilege on DBC.DBQLModeMacro.

BEGIN QUERY LOGGING Syntax

```
BEGIN QUERY LOGGING
  [ WITH with_item [,...] ]
  [ MODE = m ]
  [ LIMIT limit_item [ AND limit_item ] ]
  ON on_items [;]
```

with_item

```
{ ALL |
  EXPLAIN |
  LOCK = n |
  NONE |
  [ NO COLUMNS ] OBJECTS |
```

```

PARAMINFO |
FEATUREINFO |
SQL |
[ DETAILED ] STATSUSAGE |
STEPINFO |
USECOUNT |
UTILITYINFO |
[ VERBOSE ] XMLPLAN
}

```

limit_item

```

{ SQLTEXT [ = n ] |

  { SUMMARY = n1, n2, n3 | THRESHOLD [ = n ] }
  [ CPUTIME | CPUTIMENORM | ELAPSEDSEC | ELAPSEDTIME | IOCOUNT ]
}

```

on_items

```

{ { ALL | user_name } [ ACCOUNT = { 'account_string' | ('account_string'
[,...]) } ] |

  { user_name | database_name }[,...] |
  APPLNAME = { 'application_name' | ('application_name' [,...])
}
}

```

BEGIN QUERY LOGGING Syntax Elements***with_item***

Specifies logging options for the request objects, the AMP steps produced to perform the request, the EXPLAIN text for the request, the SQL request text, or the optimizer plan for the request as an XML document.

You cannot specify any logging option if you specify LIMIT SUMMARY.

Specifying any of the logging options can have a performance impact on your system.

Only the default set of request information is logged in a defined default row for each request for each specified user, unless you specify a logging option or WITH NONE.

For descriptions of the logging options and option modifiers, see *Teradata Vantage™ - Database Administration*, B035-1093. You can set the logging options listed below.

ALL

Information is logged for the EXPLAIN, OBJECTS, SQL, and STEPINFO options. Information is not logged for the XMLPLAN, LOCK, STATSUSAGE, PARAMINFO, FEATUREINFO, UTILITYINFO, or USECOUNT options.

EXPLAIN

EXPLAIN text for the request is logged.

Note:

You cannot specify the EXPLAIN option with the SUMMARY or THRESHOLD options.

LOCK=*n*

Lock contentions longer than *n* centiseconds are logged in XML format. For more information on DBQLXMLLockTbl, including how to shred the lock plan data, see *Teradata Vantage™ - Database Administration*, B035-1093.

NONE

DBQL data is not logged for the items you specify, including:

- account:user pair or account:user list
- application name or application name list
- user name list
- ALL:account name
- ALL:account string list
- ALL without any other specification, which specifies all accounts

If you specify WITH NONE, you cannot specify additional options.

To remove the WITH NONE setting, you must use the corresponding END QUERY LOGGING statement. See [END QUERY LOGGING](#).

OBJECTS

Database, table, column, and index information is logged.

You can specify this option with SUMMARY or THRESHOLD.

NO COLUMNS OBJECTS

Column data is not logged in DBQLObjTbl. This option can reduce logging overhead for tables with a large number of columns.

PARAMINFO

Parameter values and metadata are logged in DBQLParamTbl.

FEATUREINFO

Log feature usage information into the FeatureUsage column of DBQLLogTbl.

To display the current list of features being logged, you can use this statement:

```
SELECT featurename FROM DBC.QryLogFeatureListV;
```

The FeatureUsage column stores usage information in binary format. You can use the TD_DBQLFUL table function to convert the binary data into a JSON document. For more information, see *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210. To display feature use information in JSON format, you can access the FeatureInfo column of the QryLogFeatureJSON view. To display current use count information, you can access DBC.QryLogFeatureUseCountV and the binary data in the FeatureUsage column of DBC.DBQLLogTbl. The use count information is based on number of rows in DBC.DBQLLogTbl.

SQL

The full text of all SQL statements is logged.

You can specify this option with THRESHOLD.

STATSUSAGE

Optimizer statistics and usage recommendations are logged in an XML document for DML requests.

If you also specify XMLPLAN, the data from STATSUSAGE and XMLPLAN is logged in one XML document.

The XML schema for the log this option produces is located at:

<http://schemas.teradata.com/queryplan/queryplan.xsd>

DETAILED STATSUSAGE

The following statistics details for all of the database objects referenced in the plan for a request are logged:

- StatTimeStamp
- Version
- OrigVersion
- NumColumns
- NumBValues
- NumEHIntervals

- NumHistoryRecords
- NumNulls
- NumAllNulls
- NumAMPs
- NumPNullDistinctVals
- PNullHighModeFreq
- AvgAmpRPV
- HighModeFreq
- NumDistinctVals
- NumRows
- CPUUsage
- IOUsage

The XML schema for the log this option produces is located at:

<http://schemas.teradata.com/queryplan/queryplan.xsd>

The DETAILED keyword is a modifier for the STATSUSAGE option.

Note:

DETAILED does not support geospatial statistics.

STEPINFO

AMP step-level information is logged.

This option can be used with THRESHOLD.

USECOUNT

Use count information is logged for a database or user. Collects use count information on the specified databases or users, regardless of who is accessing the database objects.

If you specify USECOUNT for a:

- user, you can specify any of the other logging options.
- database, you cannot specify any other logging options.

Note:

You cannot enable the WITH USECOUNT option on account strings.

You can specify WITH USECOUNT and ON ALL to log use count information for all objects on the system, for example:

```
BEGIN QUERY LOGGING WITH USECOUNT ON ALL;
```


When USECOUNT is mixed with other options and you specify ON ALL, information for the other options is logged for all objects accessed by all users. Use count information is logged for all objects in all databases, regardless of who is accessing them, for example:

```
BEGIN QUERY LOGGING WITH USECOUNT, SQL, OBJECTS LIMIT SQLTEXT=0
ON ALL;
```

UTILITYINFO

Utility information is logged in DBC.DBQLUtilityTbl.

XMLPLAN

The plan is logged in XML format for DDL, DML, and DCL requests. For DDL statements, XMLPLAN logs basic information such as StatementType and the corresponding StepNames.

In addition, the XMLPLAN option logs detailed information for the following DDL statements:

- COLLECT STATISTICS
- CREATE INDEX
- CREATE TABLE
- DROP INDEX
- DROP TABLE

XMLPLAN also logs detailed information for FastLoad and MultiLoad jobs.

The XML schema for the log this option produces is located at:

<http://schemas.teradata.com/queryplan/queryplan.xsd>

VERBOSE XMLPLAN

VERBOSE EXPLAIN text for a request is logged in XML format. Also includes details on SpoolAsgnList and HashFields that are not available with only EXPLAIN.

The VERBOSE keyword is a modifier for the XMLPLAN option.

The XML schema for the log this option produces is located at:

<http://schemas.teradata.com/queryplan/queryplan.xsd>

m

Specifies the CPU and I/O data collection mode algorithm to use:

<i>m</i>	Algorithm
0	1, the data collection algorithm.

<i>m</i>	Algorithm
1	Classic V2R6 algorithm 1 for step adjustments.
2	AMP algorithm 2, diagnostic only.
3 (default)	AMP algorithm 3, which retrieves statistics on parallel steps using AMP coordination and captures aborted step usage data.

For information about these options, see *Teradata Vantage™ - Database Administration*, B035-1093.

limit_item

Specifies boundary conditions for query logging:

<i>limit_item</i>	Description
SQLTEXT= <i>n</i>	Number of SQL characters to capture, where <i>n</i> is the number of characters from 0 through 10,000. The default is 200.
SUMMARY <i>n1,n2,n3</i>	Requests are grouped by duration intervals or number of I/Os, each group is counted, and the count results are stored. You can specify up to 3 intervals and a fourth interval is created by default as anything longer than <i>n3</i> : <ul style="list-style-type: none"> • 0 through <i>n1</i> • <i>n1</i> through <i>n2</i> • <i>n2</i> through <i>n3</i> • greater than <i>n3</i> Note: SUMMARY cannot be used with LIMIT THRESHOLD or logging options.
THRESHOLD= <i>n</i>	Requests that complete in <i>n</i> or less are counted and queries that exceed <i>n</i> are logged.
CPUTIME	<i>n</i> as AMP CPU time, where <i>n</i> represents hundredths of a second.
CPUTIMENORM	<i>n</i> as normalized AMP CPU time, where <i>n</i> represents hundredths of a second.
ELAPSEDSEC	<i>n</i> as elapsed time in seconds.
ELAPSEDTIME (default)	<i>n</i> as elapsed time, in hundredths of a second.
IOCOUNT	<i>n</i> as a count of total I/Os, where <i>n</i> is specified in the number of I/Os.

Specifying any of the limit options can have a performance impact on your system.

For complete descriptions of these options and their option modifiers. See *Teradata Vantage™ - Database Administration*, B035-1093.

on_items

Specify scope of logging for users, accounts, databases, or applications:

Item	Description
ALL	Logging for all users. The system logs each request performed for every user on the system. If you specify the USECOUNT option, logging is performed for all databases.
<i>user_name</i>	Name of one or more users for whom SQL request information is to be logged. The maximum number of individual user names you can specify in a single BEGIN QUERY LOGGING request is 100. The more users you specify, the greater the impact on system performance. If you specify a <i>user_name</i> with USECOUNT, the use count information for objects in the user database is logged, regardless of the user accessing the objects. If users access objects in another database, use count information is not logged unless the USECOUNT option is also specified for the containing database.
<i>database_name</i>	Name of one or more databases for which object use count data is to be logged.
ACCOUNT = ' <i>account_string</i> '	Set of user accounts for which SQL request information is to be logged for the specified <i>user_name</i> .
APPLNAME = ' <i>application_name</i> '	Logging is enabled or disabled for any user who logs on under the specified application name set. Application names are the names the system passes in the <i>UtilityName</i> query band. See <i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i> , B035-1184.

Examples

Example: Logging All Users for All Accounts

```
BEGIN QUERY LOGGING ON ALL;
```

Example: Logging Up To 1,000 SQL Text Characters in DBQLogTbl

A request submitted without specifying the SQLTEXT limit logs only 200 characters. This request creates a rule to log request information for AMP steps, database objects, and SQL text up to 1,000 characters on user_1:

```
BEGIN QUERY LOGGING WITH STEPINFO, OBJECTS LIMIT SQLTEXT=1000
ON user_1;
```

Example: Logging Specific Users

This request creates a rule to log request information for AMP steps, database objects, and SQL text up to 100 characters for *user_1* and *user_2*:

```
BEGIN QUERY LOGGING WITH STEPINFO, OBJECTS LIMIT SQLTEXT=100
ON user_1, user_2;
```

Example: Handling Short Queries

Log request information with an SQL text limit for *user_1* and *user_2*:

```
BEGIN QUERY LOGGING LIMIT THRESHOLD=4 AND SQLTEXT=100
ON user_1, user_2;
```

The following table describes the logging behavior for various request completion times for the specified threshold.

Query Completion Time in Seconds	Log Table ...
≤ 4	DBQLSummaryTbl. The requests are recorded only as counts in the QueryCount column.
> 4	DBQLLogTbl. The logged SQL text is limited to 100 characters.

Log requests for *user_1* and *user_2*:

```
BEGIN QUERY LOGGING WITH SQL, STEPINFO, OBJECTS LIMIT THRESHOLD=10
CPUTIME ON user_1, user_2;
```

You can specify equivalent query logging using various methods. For example, specify THRESHOLD with the SQL, STEPINFO, and OBJECTS options individually:

```
BEGIN QUERY LOGGING WITH SQL LIMIT THRESHOLD=10 CPUTIME
ON user_1, user_2;
BEGIN QUERY LOGGING WITH STEPINFO LIMIT THRESHOLD=10 CPUTIME
ON user_1, user_2;
BEGIN QUERY LOGGING WITH OBJECTS LIMIT THRESHOLD=10 CPUTIME
ON user_1, user_2;
```

Example: Logging All Users Who Are Logged On Under Specific Accounts

Log request information for AMP steps, database objects, unformatted EXPLAIN text, the request plan as an XML document, and SQL text for all users logged on under account IDs *account_1* and *account_2*:

```
BEGIN QUERY LOGGING WITH STEPINFO, OBJECTS, EXPLAIN, XMLPLAN, SQL
ON ALL ACCOUNT = ('account_1', 'account_2');
```

Example: Logging Requests by Summary Completion Time in Full Second Intervals

Log counts of the number of requests performed by *user_1* into the following intervals set in *DBQLSummaryTbl*:

- 0 - 1 second
- 1 - 5 seconds
- 5 - 10 seconds
- > 10 seconds

```
BEGIN QUERY LOGGING LIMIT SUMMARY=1,5,10 ELAPSEDSEC ON user_1;
```

Example: Logging Requests by Summary Completion Time in Subsecond Intervals

Summarize requests for all users into the following subsecond intervals set in *DBQLSummaryTbl*:

- 0 - 0.05 seconds
- 0.05 - 0.50 seconds
- 0.50 - 1.00 seconds
- > 1 second

Quantify requests in the subsecond range:

```
BEGIN QUERY LOGGING LIMIT SUMMARY=5,50,100 ELAPSEDTIME ON ALL;
```

Example: Short Requests Based on CPU Time

As an alternative to using SUMMARY or THRESHOLD options for short duration requests, specify the CPUTIME option. You can save the count only or fully log request information based on the number of CPU seconds required to complete a request.

This request counts the number of requests for *user_1* during a session in each of the CPU second intervals specified:

```
BEGIN QUERY LOGGING LIMIT SUMMARY=100, 200, 300 CPUTIME ON user_1;
```

CPUTIME intervals are created in units of 0.01 seconds of CPU time. This query specifies these intervals:

- 0 - 1 CPU seconds
- 1 - 2 CPU seconds
- 2 - 3 CPU seconds
- >3 CPU seconds

This request creates a rule to increment the query counter in DBQLSummaryTbl for requests in a session for *user_1* taking less than 100 CPU seconds and to log a complete DBQL entry in DBQLLogTbl for requests over 100 CPU seconds:

```
BEGIN QUERY LOGGING LIMIT THRESHOLD=100 CPUTIME ON user_1;
```

Example: Short Requests Based on I/O Counts

An alternative to specifying only SUMMARY or THRESHOLD options for short duration requests, specify the IOCOUNT option modifier. Specify a count only or fully log request information based on the number of disk I/O operations required to complete a request.

This request counts the number of requests during a session in each of the I/O intervals specified for *user_1*:

```
BEGIN QUERY LOGGING LIMIT SUMMARY=10, 15, 20 IOCOUNT ON user_1;
```

I/O count intervals are created for the number of I/O operations required to complete the request, so this request specifies the following intervals:

- 0 - 10 I/Os
- 10 - 15 I/Os
- 15 - 20 I/Os
- > 20 I/Os

For *user_1*, this request increments the request counter for requests in a session with fewer than 10 I/O operations and logs a complete DBQL entry in DBQLLogTbl for requests with more than 10 I/O operations:

```
BEGIN QUERY LOGGING LIMIT THRESHOLD=10 IOCOUNT ON user_1;
```

Example: Logging Requests by Threshold Normalized CPU Times

This request produces detail logs for all user requests that use more than 0.10 seconds of normalized CPU time:

```
BEGIN QUERY LOGGING LIMIT THRESHOLD=10 CPUTIMENORM ON ALL;
```

Requests that use less than 0.10 seconds of normalized CPU time are counted in DBQLSummaryTbl. The CPU times are normalized by weighting according to processor speed factors.

Example: Logging Requests on All Users for the Default Table

To monitor the usage of *myuser* in more detail, add another rule for myuser that logs STEPINFO and OBJECTS.

These requests create query logging rules on all users in the default table and specific logging of steps and database objects for *myuser*:

```
BEGIN QUERY LOGGING ON ALL;
BEGIN QUERY LOGGING WITH STEPINFO, OBJECTS ON myuser;
```

Example: Disabling Query Logging for MultiLoad Jobs

This request disables logging for MultiLoad jobs, where ALL users are logged:

```
BEGIN QUERY LOGGING WITH NONE ON APPLNAME ='MULTLOAD';
```

To resume logging for MultiLoad jobs, submit the following request and then submit a BEGIN QUERY LOGGING request to enable logging for MultiLoad jobs:

```
END QUERY LOGGING ON APPLNAME ='MULTLOAD';
```

Instead of ending the logging of MultiLoad jobs and then submitting a BEGIN QUERY LOGGING request to begin logging them again, submit a REPLACE QUERY LOGGING request to redefine the existing rule. See "REPLACE QUERY LOGGING."

The WITH NONE specification in the BEGIN QUERY LOGGING request creates a rule in DBC.DBQLRuleTbl. Submit an END QUERY LOGGING request to remove the rule.

Example: Logging Requests Except for a Single User and Account

To log all users for SQL, but not log any requests from the *myuser2* account 'marketing' pair. If *myuser2* submits a request under account 'marketing', DBQL does not log the request. However, if *myuser2* logs on with account 'sales', DBQL logs the request and the system captures the SQL text for *myuser2*: 'sales' in *DBC.DBQLSQLTbl*.

To replace an existing rule for *myuser2*, you can either submit a REPLACE QUERY LOGGING request or submit an END QUERY LOGGING request followed by a new BEGIN QUERY LOGGING request. See [REPLACE QUERY LOGGING](#) and [END QUERY LOGGING](#).

These two requests enable query logging with SQL for all users, but exclude requests for the *myuser2*: 'marketing' pair:

```
BEGIN QUERY LOGGING WITH SQL ON ALL;
BEGIN QUERY LOGGING WITH NONE ON myuser2 ACCOUNT='marketing';
```

These two requests remove the rules for logging all users for SQL with the exclusion of the *myuser2*: 'marketing' pair:

```
END QUERY LOGGING ON ALL;
END QUERY LOGGING ON myuser2 ACCOUNT='marketing';
```

To replace existing rules with a new rule set, submit a REPLACE QUERY LOGGING request. See [REPLACE QUERY LOGGING](#).

Example: Invoking and Disabling Query Logging for a Rule Set

Logging for *myuser2* changes as rules are successively disabled by submitting END QUERY LOGGING requests.

(Rule 1) Summary logging for any user, any account:

```
BEGIN QUERY LOGGING LIMIT SUMMARY = 1,5,10 CPUTIME ON ALL ;
```

These requests create the following two new rules in *DBC.DBQLRuleTbl*:

- (Rule 2) Log default information and SQL for any user with specific account 'sales'
- (Rule 3) Do not log for myuser2 with specific account 'sales'

```
BEGIN QUERY LOGGING WITH SQL ON ALL ACCOUNT='sales';
BEGIN QUERY LOGGING WITH NONE ON myuser2 ACCOUNT = 'sales';
```

This request creates the following two additional new rules in *DBC.DBQLRuleTbl*:

- (Rule 4) Log default, SQL and objects for *myuser2*, any account
- (Rule 5) Log default, SQL and objects for *myuser3*, any account

```
BEGIN QUERY LOGGING WITH SQL, OBJECTS ON myuser2, myuser3;
```

The following query logging activity results from these rules:

1. User *myuser2* logs on with account 'sales', so Rule 3 applies and no logging occurs.
2. This request removes Rule 2, but *myuser2*: 'sales' continues to use Rule 3:

```
END QUERY LOGGING ON ALL ACCOUNT='sales';
```

3. This request removes Rule 3, so *myuser2*: 'sales' begins to log queries using Rule 1 with the summary log instead of using Rule 3 as before:

```
END QUERY LOGGING myuser2 ACCOUNT='sales';
```

Example: Fine Tuning Query Logging Rules

Create a logging rule for *myuser2* and include the WITH NONE option:

```
BEGIN QUERY LOGGING WITH SQL ON ALL;
BEGIN QUERY LOGGING WITH NONE ON myuser2 ACCOUNT='sales';
```

Create a logging rule for *myuser2* to capture step data in *DBQLStepTbl*. Submit an END QUERY LOGGING request to remove the existing rule before adding the new rule:

```
END QUERY LOGGING ON myuser2 ACCOUNT='sales';
```

This request creates the new rule:

```
BEGIN QUERY LOGGING WITH STEPINFO ON myuser2 ACCOUNT='sales';
```

As an alternative, use a REPLACE QUERY LOGGING request instead of the END QUERY LOGGING and BEGIN QUERY LOGGING requests. See [REPLACE QUERY LOGGING](#).

Example: Logging Plans as XML Text

The XMLPLAN option is particularly useful for diagnosing performance issues with requests. Logging query plans in XML format enables the analysis of problematic requests using Teradata support tools.

This request logs the Optimizer request plans of all requests, including SQL DML and selected DDL statements, performed by user *u1* as XML text in *DBC.DBQLXMLTbl*. Because the XML document

contains the full SQL text, specify a LIMIT SQLTEXT value of 0 to avoid duplicate logging of the request text in *DBC.DBQLogTbl*:

```
BEGIN QUERY LOGGING WITH XMLPLAN LIMIT SQLTEXT=0 on u1;
```

Example: Logging Plans as XML Text for Detailed Statistics

```
BEGIN QUERY LOGGING WITH DETAILED STATSUSAGE ON user_1;
```

The detailed statistics logged are listed in the following table.

Logged Detailed Statistics		
AvgAmpRPV	NumBValues	NumPNullDistinctVals
CPUUsage	NumColumns	NumRows
HighModeFreq	NumDistinctVals	OrigVersion
IOUsage	NumEHIntervals	PNullHighModeFreq
NumAllNulls	NumHistoryRecords	StatTimeStamp
NumAMPs	NumNulls	Version

Example: Logging Verbose EXPLAIN as XML Text

```
BEGIN QUERY LOGGING WITH VERBOSE XMLPLAN ON user_1;
```

Example: Logging Statistics Usage for All Requests by a User

This request logs statistics usage data for all DML requests executed by *user_1*:

```
BEGIN QUERY LOGGING WITH STATSUSAGE ON user_1;
```

This request logs statistics usage data and detailed query plan information for all DML and selected DDL requests executed by user *user_1*:

```
BEGIN QUERY LOGGING WITH XMLPLAN, STATSUSAGE ON user_1;
```

Example: Logging an XML Plan with VERBOSE EXPLAIN and DETAILED STATSUSAGE Collections

This example uses multiple query logging requests to log and retrieve request information.

Create the following table in *user_1*:

```
CREATE TABLE employee (
  emp_code  INTEGER,
  emp_name  VARCHAR(30),
  dept_code INTEGER,
  salary    FLOAT);
```

Assume the *employee* table is populated with 30,000 rows and statistics have been collected on column *dept_code* using this request:

```
COLLECT STATISTICS COLUMN (dept_code) ON employee;
```

Log queries in XML format:

```
BEGIN QUERY LOGGING WITH VERBOSE XMLPLAN ON user_1;
```

Log Object, SQL, Step Details, and Verbose EXPLAIN text into the plan data for this request:

```
SELECT *
FROM employee
WHERE dept_name = 'sys_mgt';
```

This request returns verbose and XMLPLAN rules as being TRUE:

```
SHOW QUERY LOGGING ON user_1;
```

Replace the existing BEGIN QUERY LOGGING rule and collect detailed statistics usage data in XML format on *user_1*:

```
REPLACE QUERY LOGGING WITH DETAILED STATSUSAGE ON user_1;
```

Log Object, SQL, Step Details, and Statistics Usage information into the plan data for this request:

```
SELECT *
FROM employee
WHERE dept_name = 'testing';
```

This request returns Detailed Statistics and Statistics Usage rules as being TRUE:

```
SHOW QUERY LOGGING ON user_1;
```

Terminate the logging on *user_1*:

```
END QUERY LOGGING ON user_1;
```

Example: Logging an XML Plan with VERBOSE EXPLAIN and DETAILED STATSUSAGE Collections in the Same Request

This example uses the same table and COLLECT STATISTICS request as [Example: Logging an XML Plan with VERBOSE EXPLAIN and DETAILED STATSUSAGE Collections](#).

Log Object, SQL, Step Details, Verbose Explain, Statistics Details, and Statistics Usage information on *user_1*:

```
BEGIN QUERY LOGGING WITH VERBOSE XMLPLAN,  
                        DETAILED STATSUSAGE ON user_1;
```

Update the employee table, setting the salary for employee John to \$40,000:

```
UPDATE employee  
SET SALARY=40000  
WHERE emp_name='John';
```

This request reports Verbose XMLPLAN, Statistics Details, and Statistics Usage rules as TRUE:

```
SHOW QUERY LOGGING ON user_1;
```

Terminate logging on *user_1*:

```
END QUERY LOGGING ON user_1;
```

Example: Logging Object Use Counts on a Specified Database

```
BEGIN QUERY LOGGING WITH USECOUNT ON database_1;
```

Example: Logging Object Use Counts on a Mix of Databases and Users

Log object use count information on *database_1*, *database_2*, *user_1*, and *user_2*:

```
BEGIN QUERY LOGGING WITH USECOUNT
ON database_1, database_2, user_1, user_2;
```

Related Information

- BEGIN QUERY LOGGING, END QUERY LOGGING, FLUSH QUERY LOGGING, and REPLACE QUERY LOGGING in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- DUMP EXPLAIN, EXPLAIN, and INSERT EXPLAIN in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146
- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Data Dictionary*, B035-1092
- *Teradata Vantage™ - Database Utilities*, B035-1102

The XML schemas that Vantage uses for the STATSUSAGE and XMLPLAN options are located at the following URL:

<http://schemas.teradata.com/queryplan/queryplan.xsd>

FLUSH QUERY LOGGING

Flushes one, several, or all DBQL caches or workload management caches to disk.

This statement is only valid in Teradata session mode.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the EXECUTE privilege on the macro *DBQLAccessMacro* to perform FLUSH QUERY LOGGING.

FLUSH QUERY LOGGING Syntax

```
FLUSH QUERY LOGGING WITH flush_option [;]
```

FLUSH QUERY LOGGING Syntax Elements

flush_option

Specifies which database query logging or Teradata workload management cache to flush:

- ALL
ALL is mutually exclusive to all other options.
- ALLDBQL
ALLDBQL is mutually exclusive to all other DBQL options.
- ALLTDWM
ALLTDWM is mutually exclusive to all other TDWM options.
- DEFAULT
- EXPLAIN
- LOCK
- OBJECTS
- PARAMINFO
- SQL
- STATUSAGE
- STEPINFO
- SUMMARY
- TDWMEVENT
- TDWMEXCEPTION
- TDWMHISTORY
- USECOUNT
- XMLPLAN

For more information, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

FLUSH QUERY LOGGING Examples

This example flushes the caches for all DBQL and TDWM logs.

```
FLUSH QUERY LOGGING WITH ALL;
```

You cannot specify other cache flushing options if you specify ALL.

This example flushes all the caches of DBQL logs but does not flush any of the TDWM caches.

```
FLUSH QUERY LOGGING WITH ALLDBQL;
```

You cannot specify any other DBQL cache flushing options if you specify ALLDBQL.

This example flushes all the caches of TDWM logs but does not flush any of the DBQL caches.

```
FLUSH QUERY LOGGING WITH ALLTDWM;
```

You cannot specify other TDWM cache flushing options if you specify ALLTDWM.

Related Information

- *Teradata Vantage™ - Database Administration*, B035-1093
- BEGIN QUERY LOGGING, END QUERY LOGGING, and REPLACE QUERY LOGGING in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184

REPLACE QUERY LOGGING

Creates a rule or replaces the current rule with the rule you specify.

See [BEGIN QUERY LOGGING](#) for examples.

The REPLACE QUERY LOGGING syntax corresponds to the BEGIN QUERY LOGGING syntax. REPLACE QUERY LOGGING replaces an existing rule. BEGIN QUERY LOGGING creates the new rules that REPLACE QUERY LOGGING requests replace.

If a REPLACE QUERY LOGGING request specifies a rule that does not already exist, the request creates a new rule, similar to a BEGIN QUERY LOGGING request.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

EXECUTE privilege on the macro DBC.DBQLAccessMacro.

If a rule is not already set, creates a rule to satisfy the request.

To use the MODE option, EXECUTE privilege on DBC.DBQLModeMacro.

REPLACE QUERY LOGGING Syntax

```
REPLACE QUERY LOGGING
  [ WITH with_item [, ...] ]
  [ MODE = m ]
```

```
[ LIMIT limit_item [ AND limit_item ] ]
ON on_items [;]
```

Syntax Elements

with_item

```
{ ALL |
  EXPLAIN |
  LOCK = n |
  NONE |
  [ NO COLUMNS ] OBJECTS |
  PARAMINFO |
  FEATUREINFO |
  SQL |
  [ DETAILED ] STATSUSAGE |
  STEPINFO |
  USECOUNT |
  UTILITYINFO |
  [ VERBOSE ] XMLPLAN
}
```

limit_item

```
{ SQLTEXT [=n] |

  { SUMMARY = n1, n2, n3 | THRESHOLD [=n] }
  [ CPUTIME | CPUTIMENORM | ELAPSEDSEC | ELAPSEDTIME | IOCOUNT ]
}
```

on_items

```
{ { ALL | user_name } [ ACCOUNT = { 'account_string' | ('account_string'
[,...]) } ] |

  { user_name | database_name }[,...] |
  APPLNAME= { 'application_name' | ('application_name' [,...])
}
}
```


REPLACE QUERY LOGGING Syntax Elements

with_item

Keyword to use in specifying logging options for request objects, the AMP steps produced to perform the request, the EXPLAIN text for the request, the SQL request text, or the optimizer request plan as an XML document.

You cannot specify logging options if you specify LIMIT SUMMARY.

Specifying any of the logging options can impact system performance.

Only the default set of query information is logged in a defined default row for each request for each specified user, unless you specify a logging option or WITH NONE.

For descriptions of these options and option modifiers, see *Teradata Vantage™ - Database Administration*, B035-1093.

ALL

Information is logged for the EXPLAIN, OBJECTS, SQL, and STEPINFO options.
Information is not logged for the XMLPLAN, LOCK, STATSUSAGE, PARAMINFO, FEATUREINFO, UTILITYINFO, or USECOUNT options.

EXPLAIN

The EXPLAIN text for the request is logged.

Note:

You cannot specify the EXPLAIN option with the SUMMARY or THRESHOLD options.

LOCK=*n*

Lock contentions longer than *n* centiseconds are logged in XML format. For more information on DBQLXMLLockTbl, including how to shred the lock plan data, see the DBQL information *Teradata Vantage™ - Database Administration*, B035-1093.

NONE

DBQL data is not logged for the items you specify, including:

- account:user pair or account:user list
- application name or application name list
- user name list
- ALL:account name
- ALL:account name list
- ALL without any other specification, which specifies all accounts

If you specify WITH NONE, you cannot specify additional options.

To remove the WITH NONE setting, you must use the corresponding END QUERY LOGGING request. See [END QUERY LOGGING](#).

OBJECTS

Database, table, column, and index information is logged.

You can specify this option with SUMMARY or THRESHOLD.

NO COLUMNS OBJECTS

Column data is not logged in DBQLObjTbl. This option can reduce logging overhead for tables with a large number of columns.

PARAMINFO

Parameter values and metadata are logged in DBQLParamTbl.

FEATUREINFO

Log feature usage information into the FeatureUsage column of DBQLogTbl.

To display the current list of features being logged, you can use this statement:

```
SELECT featurename FROM DBC.QryLogFeatureListV;
```

The FeatureUsage column stores usage information in binary format. You can use the TD_DBQLFUL table function to convert the binary data into a JSON document. For more information, see *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210. To display feature use information in JSON format, you can access the FeatureInfo column of the QryLogFeatureJSON view. To display current use count information, you can access DBC.QryLogFeatureUseCountV and the binary data in the FeatureUsage column of DBC.DBQLogTbl. The use count information is based on number of rows in DBC.DBQLogTbl.

SQL

Full text of all SQL statements is logged.

You can specify this option with THRESHOLD.

STATSUSAGE

Query optimizer statistics and usage recommendations are logged in an XML document for optimized DML requests.

If you also specify XMLPLAN, the data from STATSUSAGE and XMLPLAN is logged in one XML document.

The XML schema for the log this option produces is located at:

<http://schemas.teradata.com/queryplan/queryplan.xsd>

DETAILED STATSUSAGE

The following statistics details for all of the database objects referenced in the plan for a request are logged:

- StatTimeStamp
- Version
- OrigVersion
- NumColumns
- NumBValues
- NumEHIntervals
- NumHistoryRecords
- NumNulls
- NumAllNulls
- NumAMPs
- NumPNullDistinctVals
- PNullHighModeFreq
- AvgAmpRPV
- HighModeFreq
- NumDistinctVals
- NumRows
- CPUUsage
- IOUsage

The XML schema for the log this option produces is located at:

<http://schemas.teradata.com/queryplan/queryplan.xsd>

The DETAILED keyword is a modifier for the STATSUSAGE option.

Note:

DETAILED does not support geospatial statistics.

STEPINFO

AMP step-level information is logged.

This option can be used with THRESHOLD.

USECOUNT

Use count information is logged for a database or user. Collects use count information on the specified databases or users, regardless of who is accessing the database objects.

If you specify USECOUNT for a:

- user, you can specify any of the other logging options.
- database, you cannot specify any other logging options.

Note:

You cannot enable the WITH USECOUNT option on account strings.

You can specify WITH USECOUNT and ON ALL to log use count information for all objects on the system, for example:

```
REPLACE QUERY LOGGING WITH USECOUNT ON ALL;
```

When USECOUNT is mixed with other options and you specify ON ALL, information for the other options is logged for all objects accessed by all users. Use count information is logged for all objects in all databases, regardless of who is accessing them, for example:

```
REPLACE QUERY LOGGING WITH USECOUNT, SQL, OBJECTS LIMIT SQLTEXT=0
ON ALL;
```

UTILITYINFO

Utility information is logged in DBC.DBQLUtilityTbl.

XMLPLAN

The plan is logged in XML format for DDL, DML, and DCL requests. For DDL statements, XMLPLAN logs basic information such as StatementType and the corresponding StepNames.

In addition, the XMLPLAN option logs detailed information for the following DDL statements:

- COLLECT STATISTICS
- CREATE INDEX
- CREATE TABLE
- DROP INDEX
- DROP TABLE

XMLPLAN also logs detailed information for FastLoad and MultiLoad jobs.

The XML schema for the log this option produces is located at:

<http://schemas.teradata.com/queryplan/queryplan.xsd>

VERBOSE XMLPLAN

The VERBOSE EXPLAIN text for a request is logged in XML format. Also includes details on SpoolAsgnList and HashFields that are not available with EXPLAIN.

The XML schema for the log this option produces is located at:

<http://schemas.teradata.com/queryplan/queryplan.xsd>

m

Specifies the CPU and I/O data collection mode algorithm to use:

<i>m</i>	Algorithm
0	1, the data collection algorithm.
1	Classic V2R6 algorithm 1 for step adjustments.
2	AMP algorithm 2, diagnostic only.
3 (default)	AMP algorithm 3, which retrieves statistics on parallel steps using AMP coordination and captures aborted step usage data.

For information about these options, see *Teradata Vantage™ - Database Administration*, B035-1093.

limit_item

Specifies boundary conditions for query logging:

<i>limit_item</i>	Description
SQLTEXT= <i>n</i>	Number of SQL characters to capture, where <i>n</i> is the number of characters from 0 through 10,000. The default is 200.
SUMMARY <i>n1,n2,n3</i>	Requests are grouped by duration intervals or number of I/Os, each group is counted, and the count results are stored. You can specify up to 3 intervals and a fourth interval is created by default as anything longer than <i>n3</i> : <ul style="list-style-type: none"> • 0 through <i>n1</i> • <i>n1</i> through <i>n2</i> • <i>n2</i> through <i>n3</i> • greater than <i>n3</i> Note: SUMMARY cannot be used with LIMIT THRESHOLD or logging options.
THRESHOLD= <i>n</i>	Requests that complete in <i>n</i> or less are counted and queries that exceed <i>n</i> are logged.

<i>limit_item</i>	Description
CPUTIME	<i>n</i> as AMP CPU time, where <i>n</i> represents hundredths of a second.
CPUTIMENORM	<i>n</i> as normalized AMP CPU time, where <i>n</i> represents hundredths of a second.
ELAPSEDSEC	<i>n</i> as elapsed time in seconds.
ELAPSEDTIME (default)	<i>n</i> as elapsed time, in hundredths of a second.
IOCOUNT	<i>n</i> as a count of total I/Os, where <i>n</i> is specified in the number of I/Os.

Specifying any of the limit options can have a performance impact on your system.

For complete descriptions of these options and their option modifiers. See *Teradata Vantage™ - Database Administration*, B035-1093.

on_items

Specify scope of logging for users, accounts, databases, or applications:

Item	Description
ALL	Logging for all users. The system logs each request performed for every user on the system. If you specify the USECOUNT option, logging is performed for all databases.
<i>user_name</i>	Name of one or more users for whom SQL request information is to be logged. The maximum number of individual user names you can specify in a single BEGIN QUERY LOGGING request is 100. The more users you specify, the greater the impact on system performance. If you specify a <i>user_name</i> with USECOUNT, the use count information for objects in the user database is logged, regardless of the user accessing the objects. If users access objects in another database, use count information is not logged unless the USECOUNT option is also specified for the containing database.
<i>database_name</i>	Name of one or more databases for which object use count data is to be logged.
ACCOUNT = ' <i>account_string</i> '	Set of user accounts for which SQL request information is to be logged for the specified <i>user_name</i> .
APPLNAME = ' <i>application_name</i> '	Logging is enabled or disabled for any user who logs on under the specified application name set. Application names are the names the system passes in the <i>UtilityName</i> query band. See <i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i> , B035-1184.

Related Information

- *Teradata Vantage™ - Database Administration*, B035-1093

- BEGIN QUERY LOGGING, END QUERY LOGGING, FLUSH QUERY LOGGING in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184

END LOGGING

Ends the auditing of SQL requests that was started with a BEGIN LOGGING request.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Other SQL dialects support similar non-ANSI standard statements with names such as STOP LOGGING.

Required Privileges

None.

END LOGGING Syntax

```
END LOGGING [ DENIALS ] [ WITH TEXT ] ON operation_specification
  [ FOR CONSTRAINT constraint_name ]
  [ BY user_name [,...] ]
  [ ON logged_item [,...] ] [;]
```

operation_specification

```
{ { operation | GRANT }[,...] | ALL }
```

logged_item

```
{ AUTHORIZATION authorization_name |

  DATABASE database_name |

  USER user_name |

  { TABLE |
    VIEW |
    MACRO |
    PROCEDURE |
    FUNCTION |
    FUNCTION WRAPPING |
    TYPE
```

```

    } [ database_name. | user_name. ] object_name
  }

```

END LOGGING Syntax Elements

DENIALS

DENIALS, entries made only when statement execution fails because the user did not have the privilege set necessary to perform the request, are no longer logged.

DENIALS can only be specified in an END LOGGING statement if the option was originally applied in a BEGIN LOGGING statement.

WITH TEXT

Text of the request that previously caused a log entry is no longer saved in the log.

If you specify WITH TEXT in an END LOGGING statement, Vantage no longer saves the text for the specified actions or the DENIALS specification, but the logging frequency is not affected.

operation_specification

Operations for which log entries should no longer be made.

ALL

Log entries are not made if any of the actions listed below are attempted against the specified object. ALL does not include the CREATE ROLE, DROP ROLE, CREATE PROFILE, or DROP PROFILE actions, because they do not apply to a specific object.

The name and ID for each column are logged when you log privileges to INSERT, REFERENCES, SELECT, and UPDATE.

If logging has already begun for ALL actions on an object, a subsequent request to begin or end logging changes logging activity only for the specified action.

operation

One or more SQL statement types from the following list to define the actions on which logging is to end for the specified objects and users.

- ALTER EXTERNAL PROCEDURE
- ALTER FUNCTION
- ALTER GROUP

- ALTER PROCEDURE
- CHECKPOINT
- CREATE AUTHORIZATION
- CREATE DATABASE
- CREATE EXTERNAL PROCEDURE
- CREATE FUNCTION
- CREATE GLOP SET
- CREATE GROUP
- CREATE MACRO
- CREATE PROCEDURE
- CREATE PROFILE
- CREATE ROLE
- CREATE TABLE
- CREATE TRIGGER
- CREATE USER
- CREATE VIEW
- DATABASE
- DELETE
- DROP
- DROP AUTHORIZATION
- DROP DATABASE
- DROP FUNCTION
- DROP GLOP SET
- DROP GROUP
- DROP MACRO
- DROP PROCEDURE
- DROP PROFILE
- DROP ROLE
- DROP TABLE
- DROP TRIGGER
- DROP USER
- DROP VIEW
- DUMP
- EXECUTE
- EXECUTE FUNCTION
- EXECUTE PROCEDURE
- GRANT
- INDEX

- INSERT
- MACRO
- OVERRIDE DELETE CONSTRAINT
- OVERRIDE DUMP CONSTRAINT
- OVERRIDE INSERT CONSTRAINT
- OVERRIDE RESTORE CONSTRAINT
- OVERRIDE SELECT CONSTRAINT
- OVERRIDE UPDATE CONSTRAINT
- PROCEDURE
- REFERENCES
- RESTORE
- ROLLBACK DATABASE
- ROLLFORWARD DATABASE
- SELECT
- TABLE
- TRIGGER
- UDTMETHOD
- UDTTYPE
- UDTUSAGE
- UPDATE
- USER
- VIEW

GRANT

Specifies that logging is to end on all GRANT statements for the specified objects and users.

constraint_name

Name of the row-level security constraint.

Access logging is to be discontinued on the row-level security constraint specified by *constraint_name*.

The following list of operations applies to row-level security logging.

If the END LOGGING statement is for row-level security logging (a FOR CONSTRAINT *constraint_name* specification), and you do not specify an operation type, the statement stops logging of all row-level security related SQL operation types:

- DELETE
- INSERT
- OVERRIDE DELETE

- OVERRIDE DUMP
- OVERRIDE INSERT
- OVERRIDE RESTORE
- OVERRIDE SELECT
- OVERRIDE UPDATE
- SELECT
- UPDATE

If you do not want to end logging of all of these operation types (excluding OVERRIDE DUMP and OVERRIDE RESTORE, which are not logged by default), you must specify the each operation type for which you want to END LOGGING.

If you specify an ON clause for an END LOGGING of row-level security operations, then the only valid object types you can specify are row-level security tables, databases or users.

- If you specify a database or user, then Vantage ends logging of all tables in the specified database or user that contain the constraint specified by *constraint_name*.
- If you specify a table, then that table must contain a row-level security column that matches *constraint_name*.
- If you do not specify an object, the system ends logging of privilege checks on the specified operations for all tables in which the *constraint_name* appears.

user_name

Community for which log entries should be made.

If you specify *user_name*, it must be a user currently defined in the Vantage. That is, a name for which space has been created and under which a session can be established.

Absence of the BY *user_name* option implies all users, that is, those already defined to the Vantage as well as any defined in the future while this logging directive is in effect. Logging begun for specific user names cannot be ended by omitting the BY *user_name* option.

If both BY *user_name* and ON *keyword object_name* are absent, then the specified action is logged for all users and objects throughout the system.

logged_item

One or more database objects for which access logging is to be discontinued. You can specify to end logging on up to 20 objects.

If you do not specify the ON *keyword object_name* option, the system applies the other end logging rules specified elsewhere in the statement to all objects.

Each object in the ON clause must be preceded by the keyword that indicates the object type: DATABASE, FUNCTION, FUNCTION MAPPING, MACRO, PROCEDURE, TABLE, TYPE, USER, or VIEW, followed by the object name.

The ON keyword *object_name* option does not apply to the CREATE PROFILE, DROP PROFILE, CREATE ROLE, or DROP ROLE *operation* because profiles and roles are system-level objects with no containing user or database.

DATABASE

For databases, you must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

FUNCTION

For functions, you must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

FUNCTION MAPPING

For function mappings, you must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

MACRO

For macros, you must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

PROCEDURE

For procedures, you must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

TABLE

For tables, you must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

USER

For users, you must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

VIEW

For views, you must specify a qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

qualified_object_name

A qualified object name that includes the name of the containing database or user. For example, *database_name.object_name* or *user_name.object_name*.

TYPE

For user defined types, the specification must always be `SYSUDTLIB.UDT_name`, for example:

`ON TYPE SYSUDTLIB. UDT_name`

END LOGGING Examples

Example 1: Stopping UDT Logging And SQL Text

This example stops logging the SQL text for the failed attempts.

```
END LOGGING DENIALS WITH TEXT ON UDTTYPE;
```

Example 2: Stopping UDT Logging On a UDT Type

This example stops logging for all of the following attempts that failed because of insufficient privileges.

- ALTER TYPE
- CREATE CAST
- CREATE ORDERING
- CREATE TRANSFORM
- CREATE TYPE
- DROP CAST
- DROP ORDERING
- DROP TRANSFORM
- DROP TYPE
- REPLACE CAST
- REPLACE ORDERING
- REPLACE TRANSFORM
- method invocation
- NEW specification
- UDT reference

```
END LOGGING DENIALS ON UDTTYPE;
```

Example 3: Stopping UDT Logging On a UDT Method

This example stops logging for all of the following request attempts that fail because of insufficient privileges.

- ALTER METHOD
- ALTER TYPE
- CREATE CAST
- CREATE METHOD
- CREATE ORDERING
- CREATE TRANSFORM
- CREATE TYPE
- DROP CAST
- DROP METHOD
- DROP ORDERING
- DROP TRANSFORM
- DROP TYPE
- REPLACE CAST
- REPLACE ORDERING
- REPLACE TRANSFORM
- method invocation
- NEW specification
- UDT reference

```
END LOGGING ON UDTMETHOD;
```

Related Information

- *Teradata Vantage™ - Database Administration*, B035-1093
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100

END QUERY CAPTURE

Stops the capture of SQL requests initiated by a BEGIN QUERY CAPTURE request for the session.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

None required.

END QUERY CAPTURE Syntax

```
END QUERY CAPTURE [;]
```

Usage Notes

Rules and Restrictions for END QUERY CAPTURE

An END QUERY CAPTURE statement must include the WITH option or the ON ALL RULES option. A BEGIN QUERY CAPTURE session must be active before you can submit an END QUERY CAPTURE statement.

Logging Options

An END QUERY CAPTURE statement that includes the WITH logging_option clause ends the scope of the BEGIN QUERY CAPTURE session issued with the specified option.

You can specify logging options for END QUERY CAPTURE statements throughout a BEGIN QUERY CAPTURE session to decrement the number of logging options being captured.

After an option has been removed from a query capture session, it can again be added later in the same session.

ON ALL RULE Clause

An END QUERY CAPTURE statement that includes the ON ALL RULES option ends the scope of the active BEGIN QUERY CAPTURE session.

All query capture options that had been enabled during the session are terminated and the session ends.

Related Information

- *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142

END QUERY LOGGING

Stops the logging of SQL requests initiated by a BEGIN QUERY LOGGING request and commits the query log cache.

See [BEGIN QUERY LOGGING](#).

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

EXECUTE privilege on DBC.DBQLAccessMacro.

END QUERY LOGGING Syntax

```
END QUERY LOGGING ON {
  account_specification |
  { user_name_2 | database_name } [,...] |
  application_specification
} [;]
```

account_specification

```
{ ALL | user_name_1 } [ ACCOUNT = { 'account_string' |
('account_string' [,...]) } ]
```

application_specification

```
APPLNAME = { 'application_name' | ('application_name' [,...]) }
```

END QUERY LOGGING Syntax Elements

account_specification

ALL

All logging rules are removed for the specified accounts from *DBC.DBQLRulesTbl* and the rules cache and logging based on those rules is discontinued.

user_name_1

Logging rules are removed for the specified user:account pair from *DBC.DBQLRulesTbl* and the rules cache. Logging based on those rules is discontinued. You can optionally specify multiple accounts on which to discontinue logging for the user.

If query logging is active for a specific user:account pair, you must specify that user:account pair to end query logging.

You cannot specify multiple user names if you specify ACCOUNT = 'account_string'.

user_name_2

Logging rules are removed for the specified user name set from *DBC.DBQLRulesTbl* and the rules cache. Logging based on those rules is discontinued.

database_name

Logging rules are removed for the specified database name set from *DBC.DBQLRulesTbl* and the rules cache. Logging based on those rules is discontinued.

application_specification**application_name**

Logging rules are removed for the specified application name set from *DBC.DBQLRulesTbl* and the rules cache. Logging based on those rules is discontinued.

Vantage reevaluates all active sessions to determine if any other rules apply.

END QUERY LOGGING Examples

Example: Removing the ALL rule from the rule cache and *DBC.DBQLRulesTbl*

This request removes the ALL rule from the rule cache and *DBC.DBQLRulesTbl* and ends all query logging based on the rule.

```
END QUERY LOGGING ON ALL;
```

Example: Remove all DBQL rules in effect from *DBC.DBQLRuleTbl*

To remove all DBQL rules in effect from *DBC.DBQLRuleTbl* and update all active sessions to remove any DBQL logging at the next request, submit this request.

```
END QUERY LOGGING ON ALL RULES;
```

Example: Specify an account explicitly to discontinue request logging

If request logging is active on a specific account, you must specify that account explicitly to discontinue request logging.

This request removes the rule for *user_1* logged on through the *order entry* account from the rule cache and *DBC.DBQLRulesTbl* and ends all query logging based on the rule.

```
END QUERY LOGGING ON user_1 ACCOUNT = ('order_entry');
```

Example: Removing the rule for *user_1* from the rule cache and *DBC.DBQLRulesTbl*

This request removes the rule for *user_1* from the rule cache and *DBC.DBQLRulesTbl* and ends all query logging based on the rule.

```
END QUERY LOGGING ON user_1;
```

Example: Removing the rule for *user_1* and *user_2* from the rule cache and *DBC.DBQLRulesTbl*

This request removes the rules for *user_1* and *user_2* from the rule cache and *DBC.DBQLRulesTbl* and ends all query logging based on the rule.

```
END QUERY LOGGING ON user_1, user_2;
```

Related Information

- *Teradata Vantage™ - Database Administration*, B035-1093
- BEGIN QUERY LOGGING, FLUSH QUERY LOGGING, and REPLACE QUERY LOGGING in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184

SHOW QUERY LOGGING

Returns the query logging rule set applied to the specified user, database, user:account set, application set, or all users from the rules cache or from *DBC.DBQLRuleTbl*.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

None.

SHOW QUERY LOGGING Syntax

```
SHOW QUERY LOGGING ON {
  account_specification |
  { user_name_2 | database_name } [, ...] |
```

```
application_specification
} [;]
```

account_specification

```
{ ALL | user_name_1 } [ ACCOUNT = { 'account_name' |
('account_name' [,...]) } ]
```

application_specification

```
APPLNAME = { 'application_name' | ('application_name' [,...]) }
```

SHOW QUERY LOGGING Syntax Elements

account_specification

ALL

All users and all accounts.

ALL ACCOUNT = '*account_name*'

ALL ACCOUNT = ('*account_name*' [,...])

All users with this account or accounts.

If a matching rule is not found, then Vantage returns the rule for all users and all accounts.

user_name_1

The user or users.

If a matching rule is not found, Vantage returns the rule for all users and all accounts.

***user_name_1* ACCOUNT = '*account_name*'**

***user_name_1* ACCOUNT = ('*account_name*' [,...])**

The user with this account or accounts.

If a matching rule is not found, Vantage searches for a matching rule in the order indicated and returns the first match that it finds.

- User name and all accounts.
- All users with this account or accounts.
- All users and all accounts.

user_name_2

The user or users.
If a matching rule is not found, Vantage returns the rule for all users and all accounts.

database_name

the database or databases.

application_specification

APPLNAME = 'application_name'
APPLNAME = ('application_name_1', 'application_name_2')
the application or applications.

Usage Notes

Alternative Method for Listing Query Logging Rules

You can also list the query logging rules in DBC.DBQLRuleTbl by selecting all from DBQLRulesV view:

```
SELECT *
FROM DBC.DBQLRulesV;
```

Rules Hierarchy for SHOW QUERY LOGGING

The SHOW QUERY LOGGING request searches the rules cache and DBC.LogRuleTbl in the following order for a match.

Order in Hierarchy	Type of Rule
1	A rule based on an application name.
2	A rule for this user and account.
3	A rule for this user or database and any account.
4	A rule for all users with this account.
5	A rule for all users, databases, and any account.

If you create multiple specific logging rules that match a rule type, Vantage places them within each rule type in the hierarchy in the order they are created.

Vantage uses the following criteria to determine how to match SHOW QUERY LOGGING requests with the rules in the rules cache and in DBC.LogRuleTbl. Note that these criteria allow for conditions where the request does not match any specific rules in the rules cache or DBC.LogRuleTbl.

IF you specify ...	THEN the request searches for the following single best fit rule ...
an application name	the matching application name rule.
a single user or database name and account name	the matching rule from the following set, searching in this order: specified user name:specified account name specified user or database name:all account names all user names:specified account name all user and database names:all account names
a single user or database name, without an account name	the matching rule from the following set, searching in this order: specified user or database name:all account names all user and database names:all account names
ALL and a specific account name	the matching rule from the following set, searching in this order: all user names:specified account name all user names:all account names
ALL, without an account name	the matching rule for all user names:all account names .

Examples

Example: SHOW QUERY LOGGING for Simple Conditions

These examples show how SHOW QUERY LOGGING requests report query logging rules under simple conditions where it is possible for rule matches to be made at the first search level in the rule hierarchy. See [Example: Report a DBQL Rule That Logs All Users for Any Account String](#) through [Example: Report the DBQL Logging Rule for the MultiLoad Utility](#).

Assume that the following query logging rules, as indicated by the SQL text and the text of the rule in the DBC.DBQLRules view, have been created.

```
BEGIN QUERY LOGGING ON ALL;
```

This statement creates a logging rule for all users and all accounts using default options.

The row for rule 1 in DBC.DBQLRules is as follows.

```

Rule UserName  "ALL" (From an ALL rule)
Rule UserId    00000000
Account        (Rule for any Account)

DBQL RULE:
  Explain      F

```

Object	F
SQL	F
Step	F
XMLPlan	F
StatsUsage	F
UseCount	F
Param	F
Verbose	F
StatsDetails	F
UtilityInfo	F
Summary	F
Threshold	F
Text Size Limit	200

```
BEGIN QUERY LOGGING ON ALL ACCOUNT = 'finance';
```

This request creates a logging rule for all users logged on under account name finance using default logging options.

The row for rule 2 in DBC.DBQLRules is as follows.

Rule UserName	"ALL" (From an ALL rule)
Rule UserId	00000000
Account	FINANCE
DBQL RULE:	
Explain	F
Object	F
SQL	F
Step	F
XMLPlan	F
StatsUsage	F
UseCount	F
Param	F
Verbose	F
StatsDetails	F
UtilityInfo	F
Summary	F
Threshold	F
Text Size Limit	200

```
BEGIN QUERY LOGGING ON user1 ACCOUNT = 'marketing';
```

This request creates a logging rule for user1 when logged on under account name marketing using default options.

The row for rule 3 as seen using the DBC.DBQLRules view is as follows.

```

Rule UserName    user1
  Rule UserId    00001244
    Account      MARKETING

    DBQL RULE:
      Explain     F
      Object      F
      SQL         F
      Step        F
      XMLPlan     F
      StatsUsage  F
      UseCount    F
      Param       F
      Verbose     F
      StatsDetails F
      UtilityInfo F
      Summary     F
      Threshold   F
      Text Size Limit 200

```

```
BEGIN QUERY LOGGING ON user1 ACCOUNT = 'hr';
```

This request creates a logging rule for *user1* when logged on under account name *hr* using default logging options.

The row for rule 4 in DBC.DBQLRules is as follows.

```

Rule UserName    user1
  Rule UserId    00001244
    Account      HR

    DBQL RULE:
      Explain     F
      Object      F
      SQL         F
      Step        F
      XMLPlan     F
      StatsUsage  F
      UseCount    F
      Param       F
      Verbose     F
      StatsDetails F
      UtilityInfo F

```

Summary	F
Threshold	F
Text Size Limit	200

```
BEGIN QUERY LOGGING ON user1;
```

This request creates a logging rule for user1 when logged on under any account name using default options.

The row for rule 5 in DBC.DBQLRules is as follows.

```
Rule UserName    user1
  Rule UserId    00001244
    Account      (Rule for any Account)

  DBQL RULE:
    Explain      F
    Object       F
    SQL          F
    Step         F
    XMLPlan      F
    StatsUsage   F
    UseCount     F
    Param        F
    Verbose      F
    StatsDetails F
    UtilityInfo  F
    Summary      F
    Threshold    F
    Text Size Limit 200
```

```
BEGIN QUERY LOGGING ON APPLNAME = 'multload';
```

This request creates a logging rule for the MultiLoad application using default options.

The row for rule 6 in DBC.DBQLRules is as follows.

```
Rule UserName    "ALL" (From an ALL rule)
  Rule UserId    00000000
    Account      (Rule for any Account)
  ApplicationName MULTLOAD

  DBQL RULE:
    Explain      F
    Object       F
    SQL          F
```



```

Step      F
XMLPlan   F
StatsUsage F
UseCount  F
Param     F
Verbose   F
StatsDetails F
UtilityInfo F
Summary   F
Threshold F
Text Size Limit 200

```

The hierarchy for these rules is as follows.

Hierarchy Level	Rule Number
1	6
2	3
3	4
4	5
5	2
6	1

Example: Report a DBQL Rule That Logs All Users for Any Account String

Search for a rule that logs all users for any account string.

```
SHOW QUERY LOGGING ON ALL;
```

The system searches level 5 of the rules hierarchy, finds an exact match, and returns Rule 1, as follows.

```

Rule UserName  "ALL" (From an ALL rule)
  Rule UserId   00000000
    Account     (Rule for any Account)

  DBQL RULE:
    Explain     F
    Object      F
    SQL         F
    Step        F

```

XMLPlan	F
StatsUsage	F
UseCount	F
Param	F
Verbose	F
StatsDetails	F
UtilityInfo	F
Summary	F
Threshold	F
Text Size Limit	200

Example: Report a DBQL Rule That Logs All Users for a Specific Account

Search for a rule that logs all users logged on under the finance account.

```
SHOW QUERY LOGGING ON ALL ACCOUNT = 'finance';
```

The system searches level 2 of the rules hierarchy, finds an exact match, and returns Rule 2, as follows.

```
Rule UserName  "ALL" (From an ALL rule)
  Rule UserId   00000000
    Account     FINANCE

  DBQL RULE:
    Explain     F
    Object      F
    SQL         F
    Step        F
    XMLPlan     F
    StatsUsage  F
    UseCount    F
    Param       F
    Verbose     F
    StatsDetails F
    UtilityInfo F
    Summary     F
    Threshold   F
    Text Size Limit 200
```

Example: Report a DBQL Rule That Logs a Specific User for a Specific Account

Search for a rule that logs user1 when logged on under the marketing account.

```
SHOW QUERY LOGGING ON user1 ACCOUNT = 'marketing';
```

The system searches level 2 of the rules hierarchy, finds an exact match, and returns Rule 3, as follows.

Rule	UserName	user1
Rule	UserId	00001244
	Account	MARKETING
DBQL RULE:		
	Explain	F
	Object	F
	SQL	F
	Step	F
	XMLPlan	F
	StatsUsage	F
	UseCount	F
	Param	F
	Verbose	F
	StatsDetails	F
	UtilityInfo	F
	Summary	F
	Threshold	F
	Text Size Limit	200

Example: Report a DBQL Rule That Logs a Specific User for a Specific Account

Search for a rule that logs user1 , but only when logged on under account 'hr'.

```
SHOW QUERY LOGGING ON user1 ACCOUNT = 'hr';
```

The system searches level 3 of the rules hierarchy, finds an exact match, and returns Rule 4, as follows.

Rule	UserName	user1
------	----------	-------

```

Rule UserId      00001244
   Account      HR

DBQL RULE:
  Explain       F
  Object        F
  SQL           F
  Step          F
  XMLPlan       F
  StatsUsage    F
  UseCount      F
  Param         F
  Verbose       F
  StatsDetails  F
  UtilityInfo   F
  Summary       F
  Threshold     F
Text Size Limit 200

```

Example: Report a DBQL Rule That Logs a Specific User for All Accounts

Search for a rule that logs user1 for all accounts.

```
SHOW QUERY LOGGING ON user1;
```

The system searches level 4 of the rules hierarchy, finds an exact match, and returns Rule 5, as follows.

```

Rule UserName    user1
Rule UserId      00001244
   Account      (Rule for any Account)

DBQL RULE:
  Explain       F
  Object        F
  SQL           F
  Step          F
  XMLPlan       F
  StatsUsage    F
  UseCount      F
  Param         F
  Verbose       F
  StatsDetails  F

```

UtilityInfo	F
Summary	F
Threshold	F
Text Size Limit	200

Example: Report the DBQL Logging Rule for the MultiLoad Utility

Search for a rule for the MultiLoad utility.

```
SHOW QUERY LOGGING ON APPLNAME = 'multload';
```

The system searches level 1 of the rules hierarchy, finds an exact match, and returns Rule 6, as follows.

Rule UserName	"ALL" (From an ALL rule)
Rule UserId	00000000
Account	MULTLOAD
DBQL RULE:	
Explain	F
Object	F
SQL	F
Step	F
XMLPlan	F
StatsUsage	F
UseCount	F
Param	F
Verbose	F
StatsDetails	F
UtilityInfo	F
Summary	F
Threshold	F
Text Size Limit	200

Example: Searching Multiple Levels of the DBQL Rules Hierarchy

These examples show how SHOW QUERY LOGGING requests report query logging rules under conditions where it is not always possible for a rule to be matched at the first search level in the rules hierarchy. See [Example: Report a DBQL Rule That Logs All Users for All Accounts Using Default Logging](#) through [Example: Report a DBQL Rule That Logs All Queries for the MultiLoad Utility](#).

Assume that the following query logging rules, as indicated by their SQL text, have been created in DBC.DBQLRulesTbl and as seen using the *DBC.DBQLRules* view. Note that except for a few differences in the specified logging options (see Rules 2, 3, and 5), these logging rules are identical to those for [Example: SHOW QUERY LOGGING for Simple Conditions](#).

1. BEGIN QUERY LOGGING ON ALL;

This request creates a logging rule for all users and all accounts using default logging.

The row for Rule 1 in DBC.DBQLRules is as follows.

Rule UserName	"ALL" (From an ALL rule)
Rule UserId	00000000
Account	(Rule for any Account)
DBQL RULE:	
Explain	F
Object	F
SQL	F
Step	F
XMLPlan	F
StatsUsage	F
UseCount	F
Param	F
Verbose	F
StatsDetails	F
UtilityInfo	F
Summary	F
Threshold	F
Text Size Limit	200

2. BEGIN QUERY LOGGING WITH OBJECTS ON ALL ACCOUNT = 'finance';

This request creates a logging rule for all users logged on under account name finance and logs objects.

The row for Rule 2 in DBC.DBQLRules is as follows.

Rule UserName	"ALL" (From an ALL rule)
Rule UserId	00000000
Account	FINANCE
DBQL RULE:	
Explain	F
Object	F
SQL	F
Step	F

XMLPlan	F
StatsUsage	F
UseCount	F
Param	F
Verbose	F
StatsDetails	F
UtilityInfo	F
Summary	F
Threshold	F
Text Size Limit	200

3. BEGIN QUERY LOGGING WITH STEPINFO ON user1 ACCOUNT = 'marketing';

This request creates a logging rule for user1 when that user is logged on under account name marketing and logs step information.

The row for Rule 3 in DBC.DBQLRules is as follows.

Rule UserName	user1
Rule UserId	00001244
Account	MARKETING
DBQL RULE:	
Explain	F
Object	F
SQL	F
Step	F
XMLPlan	F
StatsUsage	F
UseCount	F
Param	F
Verbose	F
StatsDetails	F
UtilityInfo	F
Summary	F
Threshold	F
Text Size Limit	200

4. BEGIN QUERY LOGGING ON user1 ACCOUNT = 'hr';

This request creates a logging rule for user1 when that user is logged on under account name hr using default logging.

The row for Rule 4 in DBC.DBQLRules is as follows.

```

Rule UserName      user1
Rule UserId        00001244
Account            HR

DBQL RULE:
  Explain          F
  Object           F
  SQL              F
  Step             F
  XMLPlan          F
StatsUsage         F
UseCount           F
Param             F
Verbose           F
StatsDetails       F
UtilityInfo        F
Summary           F
Threshold          F
Text Size Limit    200

```

5. BEGIN QUERY LOGGING WITH SQL ON user1;

This request creates a logging rule for user1 when that user is logged on under any account name and logs SQL.

The row for Rule 5 in DBC.DBQLRules is as follows.

```

Rule UserName      user1
Rule UserId        00001244
Account            (Rule for any Account)

DBQL RULE:
  Explain          F
  Object           F
  SQL              F
  Step             F
  XMLPlan          F
StatsUsage         F
UseCount           F
Param             F
Verbose           F
StatsDetails       F
UtilityInfo        F

```



```

Summary      F
Threshold    F
Text Size Limit  200

```

6. BEGIN QUERY LOGGING WITH NONE ON APPLNAME = 'multload';

This request creates a logging rule for the MultiLoad application with no logging.

The row for Rule 6 in DBC.DBQLRules is as follows.

```

Rule UserName  "ALL"
Rule UserId    00000000
Account        (Rule for any Account)
ApplicationName MULTLOAD
WITH NONE      (No DBQL Logging)

```

The hierarchy for these rules is the same as that for the first example set, as follows.

Hierarchy Level	Rule Number
1	6
2	3
3	4
4	5
5	2
6	1

Example: Report a DBQL Rule That Logs All Users for All Accounts Using Default Logging

Search for a DBQL rule that logs all users for any account name using default logging. The system searches level 5 of the rules hierarchy and finds Rule 1.

```
SHOW QUERY LOGGING ON ALL;
```

The system searches level 5 of the rules hierarchy and finds a match (because there is no rule for user2 with any account, so Vantage defaults to the ALL user rule), and returns the DBC.DBQLRules row for Rule 1 as follows.

```

Rule UserName  "ALL" (From an ALL rule)
Rule UserId    00000000

```

```

Account      (Rule for any Account)

DBQL RULE:
  Explain    F
  Object     F
  SQL        F
  Step       F
  XMLPlan    F
  StatsUsage F
  UseCount   F
  Param      F
  Verbose    F
  StatsDetails F
  UtilityInfo F
  Summary    F
  Threshold  F
  Text Size Limit 200

```

Example: Report a DBQL Rule That Logs a Specific User for All Accounts

Search for a DBQL rule that logs user2 for all accounts.

```
SHOW QUERY LOGGING ON user2;
```

The system finds possible matches at levels 3 and 5 of the rules hierarchy and determines that the best fit match is at level 5 (because there is no rule for user2 with any account, so Vantage defaults to the ALL user rule, Rule 1), and returns the DBC.DBQRules row for Rule 1, as follows.

```

Rule UserName  "ALL" (From an ALL rule)
Rule UserId    00000000
Account        (Rule for any Account)

DBQL RULE:
  Explain    F
  Object     F
  SQL        F
  Step       F
  XMLPlan    F
  StatsUsage F
  UseCount   F
  Param      F
  Verbose    F

```

StatsDetails	F
UtilityInfo	F
Summary	F
Threshold	F
Text Size Limit	200

Example: Report a DBQL Rule That Logs a Specific User for a Specific Account

Search for a DBQL rule that logs user3 for the finance account.

```
SHOW QUERY LOGGING ON user3 ACCOUNT = 'finance';
```

The system finds possible matches at levels 2, 3, 4 and 5 of the rules hierarchy and determines that the best fit match is at level 4 (because there is no rule for user3 , but level 4 has a rule for ALL users and the finance account, so Vantage defaults to it), and returns the DBC.DBQRules row for Rule 2 as follows.

Rule UserName	"ALL" (From an ALL rule)
Rule UserId	00000000
Account	(Rule for any Account)
DBQL RULE:	
Explain	F
Object	F
SQL	F
Step	F
XMLPlan	F
StatsUsage	F
UseCount	F
Param	F
Verbose	F
StatsDetails	F
UtilityInfo	F
Summary	F
Threshold	F
Text Size Limit	200

Example: Report a DBQL Rule That Logs All Users for a Specific Account

Search for a DBQL rule that logs all users for the finance account.

```
SHOW QUERY LOGGING ON ALL ACCOUNT = 'finance';
```

The system finds possible matches at levels 2, 3, 4 and 5 of the rules hierarchy and determines that the best fit match is at level 4 (because there is no rule for user3 , but level 4 has a rule for ALL users and the finance account, so Vantage defaults to the that rule), and returns the DBC.DBQRules row for Rule 2 as follows.

```

Rule UserName  "ALL" (From an ALL rule)
Rule UserId    00000000
Account        FINANCE

DBQL RULE:
  Explain      F
  Object       F
  SQL          F
  Step         F
  XMLPlan      F
  StatsUsage   F
  UseCount     F
  Param        F
  Verbose      F
  StatsDetails F
  UtilityInfo  F
  Summary      F
  Threshold    F
Text Size Limit 200

```

Example: Report a DBQL Rule That Logs a Specific User for a Specific Account

Search for a DBQL rule that logs user1 for the marketing account.

```
SHOW QUERY LOGGING ON user1 ACCOUNT = 'marketing';
```

The system finds possible matches at levels 2, 3, 4 and 5 of the rules hierarchy and determines that the best fit match is at level 2 (because it matches the user1 specification and defaults to all accounts when an exact account match cannot be found), and returns the DBC.DBQRules row for Rule 3 as follows.

```

Rule UserName  user1
Rule UserId    00001244

```

```

Account      MARKETING

DBQL RULE:
  Explain    F
  Object     F
  SQL        F
  Step       F
  XMLPlan    F
  StatsUsage F
  UseCount   F
  Param      F
  Verbose    F
  StatsDetails F
  UtilityInfo F
  Summary    F
  Threshold  F
  Text Size Limit 200

```

Example: Report a DBQL Rule That Logs a Specific User for a Specific Account

Search for a DBQL rule that logs user1 for the hr account.

```
SHOW QUERY LOGGING ON user1 ACCOUNT = 'hr';
```

The system finds possible matches at levels 2, 3, 4 and 5 of the rules hierarchy, determines that level 2 is an exact match, and returns the DBC.DBQRules row for Rule 3 as follows.

```

Rule UserName  user1
Rule UserId    00001244
Account        HR

DBQL RULE:
  Explain    F
  Object     F
  SQL        F
  Step       F
  XMLPlan    F
  StatsUsage F
  UseCount   F
  Param      F
  Verbose    F

```

StatsDetails	F
UtilityInfo	F
Summary	F
Threshold	F
Text Size Limit	200

Example: Report a DBQL Rule That Logs a Specific User for Any Account

Search for a DBQL rule that logs user1 for any account.

```
SHOW QUERY LOGGING ON user1;
```

The system finds possible matches at levels 3 and 5 of the rules hierarchy and determines that the best fit match is at level 3, which logs the SQL text for user1, and returns the DBC.DBQRules row for Rule 5, as follows.

Rule UserName	user1
Rule UserId	00001244
Account	(Rule for any Account)
DBQL RULE:	
Explain	F
Object	F
SQL	F
Step	F
XMLPlan	F
Param	F
FeatureUsage	F
StatsUsage	F
UseCount	F
Verbose	F
StatsDetails	F
UtilityInfo	F
Summary	F
Threshold	F
Text Size Limit	200

Example: Report a DBQL Rule That Logs All Queries for the MultiLoad Utility

Search for a DBQL rule that logs all queries for the MultiLoad application.

```
SHOW QUERY LOGGING ON APPLNAME = 'multload';
```

The system finds a possible match at level 1 of the rules hierarchy and determines that the best fit is that rule, which excludes logging for all users, but logs queries for the MultiLoad application, and returns the DBC.DBQRules row for Rule 6, as follows.

```
Rule UserName      "ALL"
Rule UserId        00000000
Account            (Rule for any Account)
ApplicationName    MULTLOAD
WITH NONE (No DBQL Logging)
```

Example: Report a DBQL Rule That Shows SQL, StepInfo, and FeatureInfo Options Enabled

This statement requests the DBQL options for user01:

```
SHOW QUERY LOGGING ON user01;
```

The report shows that the SQL, StepInfo and FeatureInfo DBQL options are enabled for user01:

```
Rule UserName      "User01"
Rule UserId        000097F6
Account            (Rule for any Account)

DBQL RULE:
  Explain          F
  Object            F
  SQL               T
  Step              T
  XMLPlan           F
  Param             F
FeatureUsage        T
StatsUsage          F
UseCount            F
Verbose             F
StatsDetails        F
UtilityInfo         F
Summary             F
Threshold           F
Text Size Limit     200
```

Statistics Statements

COLLECT STATISTICS (Optimizer Form)

Collects demographic data for one or more columns, computes a statistical profile of the collected data, stores the synopsis in DBC.StatsTbl in the data dictionary, and optionally copies the statistics for one or more columns to a duplicate target table.

The columns can reside in a:

- Table
- Hash index
- Join index

The Optimizer uses the data when generating access and join plans.

You cannot collect statistics on columns with the following data types.

- BLOB
- CLOB
- JSON
- XML
- Nondeterministic UDT
- BLOB-derived UDT
- CLOB-derived UDT
- XML-derived UDT
- ARRAY/VARRAY
- Period
- Derived Period

Although you cannot collect statistics on an entire column with the JSON data type, you can collect statistics on extracted portions of the JSON document. See *Teradata Vantage™ - JSON Data Type*, B035-1150.

For information about collecting statistics on ST_GEOMETRY columns, see *Teradata Vantage™ - Geospatial Data Types*, B035-1181.

This statement collects statistical and demographic information only for use by the Optimizer. For information about collecting statistics for a query capture database, see COLLECT STATISTICS (QCD Form) in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Although SET SESSION ACCOUNT is technically a DCL statement, Vantage treats it as a DDL statement for transaction semantics.

Other SQL dialects support similar non-ANSI standard statements with names such as the following:

- CREATE STATISTICS
- UPDATE STATISTICS

Required Privileges

The following table lists the privileges required to collect statistics on various database objects.

Object on which to collect statistics	Required privilege	Object
Permanent or base global temporary table	STATISTICS	Table.
Materialized global temporary table	none.	None.
Volatile table	none.	None.
Join index	STATISTICS	Join index or its containing database or user.
Hash index	STATISTICS	Hash index or its containing database or user.
Table protected by a row-level security policy	OVERRIDE SELECT CONSTRAINT	Table.
Archived database object	<ul style="list-style-type: none"> • STATISTICS or • RESTORE 	Archived object. This enables you to restore archived statistics from an archived data source.

The following table lists the privileges required to copy statistics using the COLLECT STATISTICS FROM *source_table* syntax.

Database Object from which to Copy Statistics	Object on which SELECT Privilege is Required
Base table	Base table or its containing database or user.
join index	Underlying table set of the join index or its containing database or user.
hash index	Underlying table of the hash index or its containing database or user.

The following table lists the privileges required to submit a COLLECT STATISTICS request on various database objects that are protected by row-level security constraints.

To execute **COLLECT STATISTICS** with a **VALUES** clause on this row-level security-protected database object ...

You must have the **STATISTICS** privilege plus the **OVERRIDE SELECT CONSTRAINT** privileges on the ...

base table	table or its containing database or user.
join index	underlying table set of the index or its containing database or user.
hash index	underlying table of the index or its containing database or user.

COLLECT STATISTICS Syntax (Optimizer Form)

```
COLLECT [ SUMMARY ] { STATISTICS | STAT }
  [ USING using_option [ AND using_option ] [...] ]
  [ [ UNIQUE ] INDEX index_specification | COLUMN column_specification ] [...]
  ON collection_source FROM from_option [;]
```

using_option

```
{ SAMPLE | SYSTEM SAMPLE | SAMPLE n PERCENT | NO SAMPLE |

  { SYSTEM THRESHOLD | THRESHOLD n | NO THRESHOLD } [ PERCENT |
  DAYS ] |

  MAXINTERVALS n |

  SYSTEM MAXINTERVALS |

  MAXVALUELENGTH n |

  SYSTEM MAXVALUELENGTH

} [ FOR CURRENT ]
```

index_specification

```
{ index_name |

  [ index_name ] [ ALL ] ( column_name_1 [,...] )
  [ ORDER BY { VALUES | HASH } ( column_name_2 ) ]

}
```

column_specification

```

{ { expression |
  column_name |
  ( { expression | column_name | PARTITION }[,...] )
} [ [AS] statistics_name ] |

PARTITION |

statistics_name
}

```

collection_source

```

{ [ TEMPORARY ] [ database_name. | user_name. ] table_name_1 |

[ database_name. | user_name. ] { join_index_name |
hash_index_name }
}

```

from_option

```

{ [ TEMPORARY ] [ database_name. | user_name. ] table_name_2 |
[ database_name. | user_name. ] { join_index_name |
hash_index_name }
} [ COLUMN {
  { column_name_3 | PARTITION } |
  statistics_name |
  ( { column_name | PARTITION }[,...] )
}
]

```

COLLECT STATISTICS Syntax Elements (Optimizer Form)**SUMMARY**

Update only object-level statistical information such as cardinality, one-AMP and all-AMPs sampling estimates, average row size, average block size, and so on.

Calculates an estimated compression ratio for the primary subtable of tables compressed manually with block-level compression. The compression ratio is also calculated implicitly when statistics are collected on a column or index. The compression information is collected for informational purposes only and is not used by the Optimizer in determining the execution plan.

If you specify SUMMARY, you cannot specify USING options, column references, or explicit COLUMN or INDEX references.

You cannot specify SUMMARY for volatile tables.

using_option

You cannot specify USING options with the SUMMARY option.

SAMPLE

Scan a system-determined percentage of table rows to collect the specified statistics. SAMPLE has the same meaning as SYSTEM SAMPLE and is only provided for backward compatibility to enable existing COLLECT STATISTICS scripts that specify the USING SAMPLE option to continue to run. Use the SYSTEM SAMPLE option instead of SAMPLE. Geospatial NUSIs are always sampled at 100%.

SYSTEM SAMPLE

Scan a system-determined percentage of table rows to collect statistics.

The database may collect a sample of 100% several times before downgrading the sample percentage to a lower value.

This is the default if you do not specify a sample option, unless it is overridden by the setting of the DBSControl field SysSampleOption.

You can only specify this option if you also specify an explicit column or index. You cannot specify SYSTEM SAMPLE for a standard recollection of statistics on an implicitly specified column or index set.

Geospatial NUSIs are always sampled at 100%.

SAMPLE *n* PERCENT

Scans the percentage of table rows that you specify rather than scanning all of the rows in the table to collect statistics, where *n* is a decimal or an integer from 2 through 100.

Specifying SAMPLE 100 PERCENT is equivalent to collecting full statistics.

For the first collection of statistics, the specified sample percentage overrides the default specified in the setting of the DBSControl parameter SysSampleOption.

For recollection of statistics, SAMPLE *n* PERCENT overrides any previous SAMPLE option specifications and instead scans *n* percent of the rows in the table.

Geospatial NUSIs are always sampled at 100%.

NO SAMPLE

Use a full-table scan to collect the specified statistics.

You can only specify this option if you also specify an explicit index or column set.

Note:

Geospatial NUSIs are always sampled at 100%.

SYSTEM THRESHOLD

Do not collect statistics if the percentage of changed data and the age of the current statistics are below the values determined by the system.

This option is valid only for tables.

You can only specify this option if you also specify an explicit index or column set.

You cannot use this option to collect statistics on geospatial NUSIs.

SYSTEM THRESHOLD PERCENT

Change the percentages. This is the default if you do not specify a PERCENT threshold option unless it is overridden by the setting of the DBSControl parameters DefaultUserChangeThreshold and SysChangeThresholdOption.

This option is valid only for tables.

You can only specify this option if you also specify an explicit index or column set.

You cannot use this option to collect statistics on geospatial NUSIs.

SYSTEM THRESHOLD DAYS

The age in days of the current statistics. This is the default if you do not specify a DAYS threshold option unless it is overridden by the setting of the DBSControl parameter DefaultTimeThreshold.

This option is only valid for tables.

The system does not enforce day thresholds, so recollection of statistics is determined by the PERCENT threshold option.

You can only specify this option if you also specify an explicit index or column set.

You cannot use this option to collect statistics on geospatial NUSIs.

THRESHOLD n PERCENT

Recollect statistics if the percentage of change in the statistics exceeds the specified percentage, where n is a decimal or an integer.

Note:

Statistics are not recollected if:

- the age of the statistics and the percentage of change do not exceed the values specified for THRESHOLD n DAYS and THRESHOLD n PERCENT.
- the percentage of change does not exceed the number specified in THRESHOLD n PERCENT and NO THRESHOLD DAYS is specified.

You cannot use this option to collect statistics on geospatial NUSIs.

THRESHOLD n DAYS

Recollect statistics if the age of the statistic is greater than or equal to the number of days specified, where n is an integer from 1 through 9999.

This option is only valid for tables.

You can only specify this option if you also specify an explicit index or column set.

Note:

Statistics are not recollected if:

- the age of the statistics and the percentage of change do not exceed the values specified for THRESHOLD n DAYS and THRESHOLD n PERCENT.
- the age of statistics does not exceed the specified number of days specified in THRESHOLD n DAYS and NO THRESHOLD PERCENT is specified.

You cannot use this option to collect statistics on geospatial NUSIs.

NO THRESHOLD

One of the following:

- Do not apply any thresholds to the collection of statistics.
- Remove the existing threshold before collecting the statistics.

You can only specify this option if you also specify an explicit index or column set.

Note:

You cannot use this option to collect statistics on geospatial NUSIs.

NO THRESHOLD PERCENT

One of the following:

- Do not apply a PERCENT change threshold to the collection of statistics.
- Remove the existing PERCENT change threshold before collecting the statistics.

You can only specify this option if you also specify an explicit index or column set.

The effect of this option varies, depending on whether the request is for a first collection of statistics or for a recollection:

- For the first collection of statistics, NO THRESHOLD PERCENT overrides the default setting of the DBSControl parameters SysChangeThresholdOption and DefaultUserChangeThreshold.
- For recollections of statistics, NO THRESHOLD PERCENT overrides any previous change threshold percent specification.

Note:

Statistics are not recollected if the age of the statistics does not exceed the specified number of days specified in THRESHOLD *n* DAYS and NO THRESHOLD PERCENT is specified.

You cannot use this option to collect statistics on geospatial NUSIs.

NO THRESHOLD DAYS

One of the following:

- Do not apply a DAYS threshold to the collection of statistics.
- Remove the existing DAYS threshold before collecting the statistics.

You can only specify this option if you also specify an explicit index or column set.

For the first collection of statistics, NO THRESHOLD DAYS overrides the default setting of the DBSControl parameter DefaultTimeThreshold.

For recollection of statistics, NO THRESHOLD DAYS overrides any previous THRESHOLD *n* DAYS specification.

Note:

Statistics are not recollected if the percentage of change does not exceed the number specified in THRESHOLD *n* PERCENT and NO THRESHOLD DAYS is specified.

You cannot use this option to collect statistics on geospatial NUSIs.

SYSTEM MAXINTERVALS

The system-determined maximum number of intervals is used for this histogram.

You can only specify this option if you also specify an explicit index or column set.

This option is only valid for tables.

You cannot use this option to collect statistics on a geospatial NUSI.

MAXINTERVALS *n*

Specifies the maximum number of histogram intervals to be used for the collected statistics, where *n* an integer from 0 through 500. The database may adjust the specified maximum number of intervals depending on the maximum histogram size.

This option is valid only for tables.

You can only specify this option if you also specify an explicit index or column set.

If the optimizer uses summary statistics most of the time and rarely uses detailed statistics, consider lowering the MAXINTERVALS value you specify to 0 to make the Optimizer aware of how it is using statistics. However, make sure that the performance of queries that use detailed statistics is not degraded because of this change.

You cannot use this option to collect statistics on a geospatial NUSI.

SYSTEM MAXVALUELENGTH

The system-determined maximum column width for histogram values such as MinValue, ModeValue, MaxValue, and so on.

This option is valid only for tables.

You can only specify this option if you also specify an explicit index or column set.

You cannot use this option to collect statistics on a geospatial NUSI.

MAXVALUELENGTH *n*

Specifies the maximum size for histogram values such as MinValue, ModeValue, MaxValue, and so on, where *n* is an integer.

For single-character statistics on CHARACTER and VARCHAR columns, *n* specifies the number of characters. For all other options, *n* specifies number of bytes.

You can only specify this option if you also specify an explicit index or column set.

You cannot use this option to collect statistics on a geospatial NUSI.

This option is valid only for tables.

FOR CURRENT

The USING options specified for the current statistics recollection request are to be used only for the current COLLECT STATISTICS request and that the previously specified

options for the request are to be used for any future recollections of statistics, unless otherwise overridden.

index_specification

Index for which statistics are to be collected.

UNIQUE

The index for which statistics are to be collected is a unique index. This option is not supported for geospatial indexes, because a geospatial index must be a NUSI.

index_name

The name of the named index for which statistics are to be collected. Note that you specify column names with a named index only when it is defined with an ORDER BY clause.

ALL

The ALL option was used to create the index. You cannot use this option to collect statistics on a geospatial NUSI.

column_name_1

The names of one or more columns on which the index exists and for which statistics are to be collected.

You cannot collect statistics on columns with a data type of Period, JSON, XML, BLOB, CLOB, or any UDT other than Geospatial.

However, you can collect statistics on the BEGIN and END expressions for a Period type column.

Note:

A geospatial NUSI column cannot be specified with other columns.

Statistics can be collected on extracted portions of the JSON type. See *Teradata Vantage™ - JSON Data Type*, B035-1150.

ORDER BY

An ORDER BY clause was used to create the index.

ORDER BY orders the index by the contents of a single column.

You cannot use this option to collect statistics on a geospatial NUSI.

VALUES

The ORDER BY VALUES option was used to create the index.

VALUES can order a single numeric column of four bytes or less. VALUES is the default.

You cannot use this option to collect statistics on a geospatial NUSI.

HASH

The ORDER BY HASH option was used to create the index.

HASH hash-orders a single column instead of hash-ordering all columns (the default).

You cannot use this option to collect statistics on a geospatial NUSI.

column_name_2

The name of the column on which the index is ordered.

You cannot collect statistics on columns with a data type of Period, JSON, XML, BLOB, CLOB, or any UDT other than Geospatial.

However, you can collect statistics on the BEGIN and END expressions for a Period type column.

Statistics can be collected on extracted portions of the JSON type. See *Teradata Vantage™ - JSON Data Type*, B035-1150.

column_specification

Statistics are to be collected for a column set.

expression

An expression or a list of expressions for which statistics are to be collected. The result of the expression must be a data type on which field statistics can be collected directly, such as an integer, character, or date. Expressions can reference scalar UDFs, UDTs, and the BEGIN or END expressions of a Period type column. For more information on the SQL expressions you can use, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

Note:

You cannot specify nondeterministic expressions such as RANDOM, nondeterministic UDFs, and nondeterministic CAST or EXTRACT operations.

column_name

The names of the columns for which statistics are to be collected.

The maximum number of columns on which joint statistics can be collected for a single `COLLECT STATISTICS` request is 64.

You cannot collect statistics on columns with a data type of Period, JSON, XML, BLOB, CLOB, or any UDT other than Geospatial.

You can collect statistics on the `BEGIN` and `END` expressions for a Period type column.

Note:

A geospatial NUSI column cannot be specified with other columns.

Statistics can be collected on extracted portions of the JSON type. See *Teradata Vantage™ - JSON Data Type*, B035-1150.

PARTITION

Statistics are to be collected on the system-derived `PARTITION` column set for a table.

You should collect `PARTITION` statistics on tables that are partitioned by row or column because the Optimizer uses them to do the costing and cardinality estimation based on the partition elimination. For all partitioned tables, any refreshment operation should include the system-derived `PARTITION` column set.

You cannot use this option to collect statistics on a geospatial NUSI.

You cannot collect statistics on the system-derived `PARTITION#Ln` columns.

statistics_name

The statistics collected are to be saved under *statistics_name*.

A *statistics_name* is required if statistics are for other than column references. When recollecting statistics, if the column ordering is different or if the expressions do not find an exact match with existing statistics, the Optimizer collects the statistics as new. Naming the statistics and using the names during recollections ensures that the existing statistics are recollected instead of creating a new set of statistics.

You cannot specify this option for `COLUMN PARTITION` or `COLUMN (PARTITION)`.

You can use the statistics name for recollections, copies, transfers, `HELP STATISTICS`, `SHOW STATISTICS`, and `DROP STATISTICS`.

collection_source

Following are the `ON` clause options.

TEMPORARY

Statistics are to be collected for a materialized instance of a global temporary table in the current session. A request to collect statistics on a global temporary table materializes the global temporary table, if the table is not already materialized in the session.

You cannot use this option to collect statistics on a geospatial NUSI. This keyword is not valid for permanent tables.

You cannot collect system-derived PARTITION or PARTITION#L *n* statistics for global temporary tables.

database_name

The containing database for the *table_name*, *join_index_name*, or *hash_index_name*, if other than the default database.

user_name

The containing user for the *table_name*, *join_index_name*, or *hash_index_name*, if other than the default database.

table_name_1

The name of the table or global temporary table for which column or index statistics are to be collected. You cannot collect statistics on journal tables.

join_index_name

The name of the join index for which statistics are to be collected.

hash_index_name

The name of the hash index for which statistics are to be collected.

from_option

You can specify the following options.

TEMPORARY

The name of the global temporary source table from which statistics are to be copied to the target table.

database_name

The containing database for the *table_name_2*, *join_index_name*, or *hash_index_name*, if other than the default database.

user_name

The containing user for the *table_name_2*, *join_index_name*, or *hash_index_name*, if other than the default database.

table_name_2

The name of the base source table from which statistics are to be copied to the target table.

join_index_name

The name of the join index for which statistics are to be collected.

hash_index_name

The name of the hash index for which statistics are to be collected.

column_name_3

The column name set, partition, or index for which statistics are to be copied from the source table to the target table.

PARTITION

The column name set, partition, or index for which statistics are to be copied from the source table to the target table.

Examples

Example: Collecting Full Statistics

Assuming that an index is defined on the empno and name columns of the employee table, the following request collects statistics for that index.

```
COLLECT STATISTICS
  INDEX (empno, name)
  ON employee;
```

Example: Collecting Sampled Statistics

The following request is similar to that of [Example: Collecting Full Statistics](#). However, sampled statistics on the index are collected instead of scanning the entire index.

```
COLLECT STATISTICS
  USING SAMPLE INDEX (empno, name)
ON employee;
```

Example: Collecting Sampled Statistics

This request samples statistics on column *orders.o_orderkey* using a user-specified sample percentage of 10%. The database collects the statistics by reading only 10% of the rows in *orders*.

```
COLLECT STATISTICS
  USING SAMPLE 10 PERCENT
  COLUMN o_orderkey
ON orders;
```

This request recollects the statistics on the *o_orderkey* column with the sample size of 10% and on the *o_orderdatetime* column with a system-selected sample size.

```
COLLECT STATISTICS
  COLUMN o_orderkey,
  COLUMN o_orderdatetime
ON orders;
```

This request recollects the statistics on *o_orderkey* column with the sample size of 10% (set by the first request) and on *o_orderdatetime* with a system-selected sample size (set by the second request).

```
COLLECT STATISTICS
ON orders;
```

Example: Collecting SUMMARY Statistics

The following request collects or refreshes the table-level demographics for a table named *orders*.

```
COLLECT SUMMARY STATISTICS
ON orders;
```

Example: Recollecting Statistics Without Specifying Thresholds

Subsequent entry of the following request on the same table refreshes the statistics for the *empno* *-name* index, along with any other statistics previously collected for the employee table, including PARTITION statistics.

```
COLLECT STATISTICS ON employee;
```

The following request collects statistics on *o_orderstatus* for the first time without specifying a THRESHOLD option. The database collects the statistics and the system uses the SYSTEM THRESHOLD, which is the default threshold option when DBSControl fields DefaultUserChangeThreshold, SysChangeThresholdOption, and DefaultTimeThreshold are not set.

```
COLLECT STATISTICS
COLUMN o_orderstatus
ON orders;
```

Example: A Simple Recollect Statistics Request Using the THRESHOLD Option

The following request collects statistics on the multicolumn set (*o_orderstatus*, *o_orderkey*) for the first time and specifies the FOR CURRENT option. The database collects the requested statistics, but because of the FOR CURRENT specification, it does not store the THRESHOLD setting nor does it store the default threshold options from the DBSControl fields DefaultUserChangeThreshold and SysChangeThresholdOption.

```
COLLECT STATISTICS
USING SYSTEM THRESHOLD FOR CURRENT
COLUMN (o_orderstatus, o_orderkey)
ON orders;
```

Example: Using the THRESHOLD Option to Recollect Statistics

The following two requests collect statistics on the *o_orderkey* and *o_orderdatetime* columns of the *orders* table for the first time. The database collects full statistics for both requests, and the system retains the THRESHOLD options at each level.

The first request specifies thresholds of a 10% change in the data or 7 days having passed since statistics were last collected on *orders.o_orderkey*. While the database does not use these criteria for a first time collection of statistics, it does save them in *DBC.StatsTbl* to consult for future requests to recollect statistics on *orders.o_orderkey*.

For later requests to recollect statistics on *orders.o_orderkey*, the database determines whether the percent change in the data exceeds 10% and if more than 7 days have passed since statistics were last collected. If either or both criteria are exceeded, the system recollects the statistics, but if neither criterion is exceeded, the system does not recollect the statistics on *orders.o_orderkey*.

```
COLLECT STATISTICS USING THRESHOLD 10 PERCENT AND THRESHOLD 7 DAYS
COLUMN o_orderkey
ON orders;
```

The second request specifies a system-determined data change threshold. While the database does not use this criterion for a first time collection of statistics, it does save it in *DBC.StatsTbl* to consult for future requests to recollect statistics on *orders.o_orderdatetime*.

For later requests to recollect statistics on *orders.o_orderdatetime*, the database determines whether the percent change in the data since statistics were last collected exceeds the system-determined threshold criterion. If it does, the system recollects the statistics, but if it does not, the system does not recollect the statistics on *orders.o_orderdatetime*.

```
COLLECT STATISTICS USING SYSTEM THRESHOLD
COLUMN o_orderdatetime
ON orders;
```

The database stores the thresholds you specify and uses them for future requests to recollect statistics on the *o_orderkey* or *o_orderdatetime* columns of *orders*.

The following request recollects statistics on the *orders* table. Based on the THRESHOLD settings that were in place when the statistics were collected, the Optimizer does not recollect statistics on *orders.o_orderkey* if the table growth is less than 10% or the age of the statistics is less than 7 days. The database only recollects these statistics if the growth exceeds either 10% or the age is greater than 7 days. Similarly, if the growth of the column *o_orderdatetime* is less than the system-selected threshold for *o_orderdatetime*, the Optimizer does not recollect those statistics.

```
COLLECT STATISTICS ON orders;
```

Depending on the settings for the THRESHOLD option in those requests, the following criteria determine whether the Optimizer actually recollects the requested statistics.

- If the growth in cardinality of *orders* is less than 10% and if the existing statistics are less than 7 days old, the Optimizer does not recollect those statistics.

Even though you specified both a growth percentage threshold and a number of days threshold, both criteria need not be met for the database to recollect statistics on *o_orderkey*. For example, if the system determines that the growth percentage of the table exceeds its system-determined threshold, it recollects the specified statistics even if the number of days threshold is not exceeded.

- If the growth in cardinality of *orders* is less than the system-selected threshold for *o_orderdatetime*, the Optimizer does not recollect those statistics.

The thresholds that the database selects depend on the history of the column update-delete-insert counts from the last statistics collection, and so forth. For details, see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

The intent of the following request is to recollect statistics on *orders.o_orderdatetime* if the established thresholds are met. If the growth of the table is less than the system-selected threshold, the Optimizer does not recollect the requested statistics.

```
COLLECT STATISTICS
COLUMN o_orderdatetime
ON orders;
```

The intent of the following request is to recollect statistics on *o_orderdatetime* with a user-specified growth threshold of 10%. If the table growth is not greater than 10%, the Optimizer does not recollect the requested statistics. The database records the newly specified THRESHOLD percentage so it can be used for future recollections of statistics.

```
COLLECT STATISTICS USING THRESHOLD 10 PERCENT
COLUMN o_orderdatetime
ON orders;
```

The following request recollects statistics on *o_orderdatetime* with no checking for threshold options. This forces the statistics to be recollected. Because you specify the FOR CURRENT clause, the existing threshold options are not changed, and the specified NO THRESHOLD setting is not remembered for subsequent recollections of statistics on *orders.orderdatetime*.

```
COLLECT STATISTICS USING NO THRESHOLD FOR CURRENT
COLUMN o_orderdatetime
ON orders;
```

The intent of the following request is to recollect statistics on *o_orderdatetime*. The NO THRESHOLD specification forces the database to recollect the statistics. The Optimizer remembers the newly specified THRESHOLD option and always recollects statistics in the future.

```
COLLECT STATISTICS USING NO THRESHOLD
COLUMN o_orderdatetime
ON orders;
```

The following request recollects the statistics collected in the previous example. The Optimizer skips the recollection on *o_orderkey* if the number of changes to the table demographics from the last collection time is less than 10% and the age of the statistics is less than 7 days.

In other words, if the number of changes exceeds 10% or if the age of the previously collected statistics is greater than 7 days, the database recollects the statistics for *o_orderkey*. Similarly, if changes to the table demographics are less than the system-determined change threshold for *o_orderdatetime* and *o_orderstatus*, the Optimizer skips the recollection for those two columns.

```
COLLECT STATISTICS ON orders;
```

The following request recollects statistics on *o_orderdatetime*. If the table demographics changes since the last time statistics were collected are less than the system-determined change threshold, the Optimizer does not recollect the requested statistics.

```
COLLECT STATISTICS
COLUMN o_orderdatetime
ON orders;
```

The following request recollects statistics on *o_orderdatetime* with a user-specified change threshold of 10 percent. If the table demographics changes since the last time statistics were collected are less than 10%, the Optimizer does not recollect the requested statistics.

The database stores the change THRESHOLD percentage to be used for future recollections.

```
COLLECT STATISTICS
USING THRESHOLD 10 PERCENT
COLUMN o_orderdatetime
ON orders;
```

The following request recollects statistics on *o_orderdatetime* without checking for thresholds. In other words, the request forces the statistics to be recollected. Because you specify the FOR CURRENT option, the database does not change the existing THRESHOLD specifications, and does not store the specified NO THRESHOLD option for later recollections.

```
COLLECT STATISTICS
USING NO THRESHOLD FOR CURRENT
COLUMN o_orderdatetime
ON orders;
```

The following request is identical to the previous request except that unlike that request, this COLLECT STATISTICS request does not specify the FOR CURRENT option. Because the request does not specify FOR CURRENT, the Optimizer stores this NO THRESHOLD specification and does not skip recollecting statistics for subsequent requests.

```
COLLECT STATISTICS
USING NO THRESHOLD
COLUMN o_orderdatetime
ON orders;
```

Example: Collecting Statistics on Multiple Columns

These requests collect statistics on two columns of table_1.

```
COLLECT STATISTICS ON table_1 COLUMN (column_1, column_2);
```

Example: Collecting Single-Column PARTITION Statistics

The following examples collect statistics on the system-derived PARTITION column for the table named table_1.

Even though you have specified sampled statistics for the second example, the system ignores it and uses a value of 100 percent because you cannot collect sampled statistics on a system-derived PARTITION column. See *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

```
COLLECT STATISTICS ON table_1 COLUMN PARTITION;
COLLECT STATISTICS USING SAMPLE table_1 COLUMN PARTITION;
```

Example: Collecting Multicolumn PARTITION Statistics

This example collects multicolumn statistics on the following column set for the orders table.

- System-derived PARTITION column
- quant_ord
- quant_shpd

```
COLLECT STATISTICS ON orders
  COLUMN (quant_ord, PARTITION, quant_shpd);
```

The database honors the column ordering that you specify. The recommended best practice is to specify the PARTITION as the first column for multicolumn PARTITION statistics. Collecting single-column PARTITION statistics is recommended for partitioned and nonpartitioned tables. However, multicolumn PARTITION statistics are only potentially useful for row partitioned tables.

Example: Collecting Statistics on a Single-Column with an Expression

This example uses an expression to collect statistics on a single column.

```
COLLECT STATISTICS USING SYSTEM SAMPLE
  COLUMN BEGIN(eff_dt_duration) AS stats_begin_eff_dt_duration
  ON orders;
```

Example: Collecting Statistics on Multiple Columns with an Expression

This example includes an expression and collects statistics on multiple columns.

```
COLLECT STATISTICS USING SYSTEM SAMPLE
COLUMN (EXTRACT(MONTH FROM o_orderdate), o_totalprice)
AS stats_month_price
ON orders;
```

Example: Collecting Statistics on a Constant Expression

This example collects statistics on constant expressions, that is, an expression that does not refer to columns by name.

```
COLLECT STATISTICS USING SYSTEM SAMPLE
COLUMN 100,
COLUMN CURRENT_DATE+10
ON orders;
```

In this example, the Optimizer eliminates the constant expressions 100 and CURRENT_DATE+10 from the COLLECT STATISTICS request.

Because the rewritten expressions do not refer to columns, the Optimizer rewrites the request to use the SUMMARY option. The Optimizer also removes the USING SYSTEM SAMPLE option from the final request.

```
COLLECT SUMMARY STATISTICS
ON orders;
```

Example: Collecting Statistics on Expressions Using Statistics Names

This example collects statistics on columns identified with the expressions, COLUMN 100 and COLUMN CURRENT_DATE+10, using the respective statistics names, AS stats_one_hundred and AS stats_ten_days_from_today.

```
COLLECT STATISTICS USING SYSTEM SAMPLE
COLUMN 100 AS stats_one_hundred,
COLUMN CURRENT_DATE+10 AS stats_ten_days_from_today
ON orders;
```

Example: Collecting Statistics on Built-In Function Expressions

You can also collect statistics on constant expressions that refer to built-in functions.

The Optimizer remembers the materialized values for the `CURRENT_TIMESTAMP`, `CURRENT_DATE` and `USER` functions when statistics are collected and then substitutes the stored materialized values when the statistics are used.

For other built-in functions like `SESSION` and `CURRENT_ROLE`, the Optimizer substitutes the values based on the session parameters at the time the statistics are used. Even though the database allows you to specify any built-in functions, you should restrict your collecting of statistics to built-in functions other than `CURRENT_TIMESTAMP`, `CURRENT_DATE` and `USER` in the expressions.

In the following request, the Optimizer substitutes the value for `CURRENT_DATE` for the `CASE` expression and then remembers it. When the statistics are used for this expression, the Optimizer substitutes the saved value for `CURRENT_DATE`.

```
COLLECT STATISTICS
COLUMN CASE WHEN CAST(o_orderdatetime AS DATE AT 0) = CURRENT_DATE
            THEN 'orders-today'
            ELSE 'orders-old'
END AS stats_orders_today_and_old
ON orders;
```

Example: Collecting Statistics on a Geospatial Column

This example collects statistics on a non-indexed geospatial column.

You define the following table.

```
CREATE TABLE stat_t2 (
  rownum INTEGER,
  a      INTEGER,
  b      INTEGER,
  c      INTEGER,
  d      INTEGER,
  e      INTEGER,
  sp     ST_GEOMETRY,
  dummy  CHARACTER(20000));
```

You can then collect statistics on the geospatial column `sp`, using a request like the following.

```
COLLECT STATISTICS COLUMN(sp) ON stat_t2;
```

Example: Collecting Statistics on a Geospatial Index

This example collects statistics on the geospatial index defined on geospatial column *sp*.

You define the following table.

```
CREATE TABLE stat_t2_idx (
  rownum INTEGER,
  a      INTEGER,
  b      INTEGER,
  c      INTEGER,
  d      INTEGER,
  e      INTEGER,
  sp     ST_GEOMETRY,
  dummy  CHARACTER(20000))
INDEX (sp);
```

You can then collect statistics on the geospatial index *sp*, using a request like the following.

```
COLLECT STATISTICS INDEX(sp) ON stat_t2_idx;
```

Example: Collecting Statistics on Named Indexes

This example collects statistics on an index named *unique_1*.

```
COLLECT STATISTICS INDEX unique_1 ON table_1;
```

Example: Collecting Statistics on an Unnamed Indexes

The following example illustrates collecting statistics on an unnamed index that includes the index definition.

```
COLLECT STATISTICS INDEX (column_1, column_2) ON table_1;
```

Example: Copying Statistics From a Source Table to an Identical Target Table

When you copy base table statistics from a source table to an identical target table, the database also copies the USING options from the source to the target by default. You cannot modify the USING options for the source table as part of a copy operation.

When you copy PARTITION statistics, the statistics copied to the target table might not correctly represent the data in the target table because of differences in internal partition number mapping between the source and target tables. This is true even if the table definitions returned by a SHOW TABLE request are identical and the data is the same in both tables.

As a general rule, you should always recollect the PARTITION statistics for the target table when you copy them from a source table.

The copy operation implicitly copies the object-level demographics from the source table to the target table.

The following COLLECT STATISTICS requests copy all the statistics from *orders* to *orders_new*, which has the same definition as *orders*. The database copies the corresponding USING options to *orders_new* as well as the statistics.

```
COLLECT STATISTICS ON orders_new
FROM orders;

COLLECT STATISTICS USING SYSTEM SAMPLE
COLUMN o_orderkey
ON orders;
```

Example: Collecting Sampled Statistics with USING Options

The following statements sample statistics on column *o_orderkey* with a user-specified sampling percentage.

The following statement collects statistics by reading a sample of 10% of the table rows.

```
COLLECT STATISTICS USING SAMPLE 10 PERCENT
COLUMN o_orderkey
ON orders;
```

The following statement samples statistics on the *o_orderdatetime* column using a system-selected sampling percentage. The system collects full statistics and remembers the system-selected sampling percentage.

The system determines whether to downgrade the sampling percentage after you have recollected those statistics a few times.

```
COLLECT STATISTICS USING SYSTEM SAMPLE
COLUMN o_orderdatetime
ON orders;
```

The following statement recollets the statistics on *orders.o_orderkey* using a sampling size of 10% and recollets statistics on *orders.o_orderdatetime* using a system-selected sampling size because the Optimizer remembers those specifications from previous specifications for collecting statistics on *orders*.

```
COLLECT STATISTICS ON orders;
```

The following statement is equivalent to the previous statement but because you explicitly specify the column references, the system recollets the statistics on *o_orderkey* using the stored sampling size of 10 percent and on *o_orderdatetime* using the stored system-selected sample size.

```
COLLECT STATISTICS
COLUMN o_orderkey,
COLUMN o_orderdatetime
ON orders;
```

The following statement changes the existing sample clause from a user-specified sample of 10 percent to a system-selected sample percentage and then recollets the statistics.

```
COLLECT STATISTICS USING SYSTEM SAMPLE
COLUMN o_orderkey
ON orders;
```

Example: Recollecting Statistics When Statistics Were First Collected Specifying a USING Clause

USING clause option settings are retained for each column, index, or multicolumn set on which statistics are collected and applied when statistics are recollected.

To continue to use the same USING options, do not specify USING options when you recollect statistics. Specifying USING options at recollection time resets the previous options and starts over with new options you specify, if different from the previous options.

If the Optimizer decides to skip the recollection for a request with new USING options, the *DBC.StatsTbl* dictionary row is updated to remember these new options without recollecting the statistics. The column *DBC.StatsTbl.LastAlterTimeStamp* is updated to reflect the change of options, but *DBC.StatsTbl.LastCollectTimeStamp* remains the same and reflects the last statistics collection timestamp.

The following requests collect the statistics on *o_orderkey* and *o_orderdatetime* for the first time. The USING options are retained for each column, index, or multicolumn set on which statistics are collected.

```
COLLECT STATISTICS
  USING SYSTEM SAMPLE AND SYSTEM THRESHOLD
  COLUMN o_orderkey
  ON orders;
COLLECT STATISTICS
  USING SYSTEM THRESHOLD
  COLUMN o_orderdatetime
  ON orders;
```

The next table-level example recollects the statistics with the saved USING options for *o_orderkey* and *o_orderdatetime*.

```
COLLECT STATISTICS ON orders;
```

The following request recollects the statistics with the saved USING options for *o_orderkey* and *o_orderdatetime*. This is equivalent to the table-level COLLECT STATISTICS request shown in the earlier example.

```
COLLECT STATISTICS COLUMN o_orderkey,
                  COLUMN o_orderdatetime
  ON orders;
```

The next example resets the using option from system-determined sample to a user-specified sample of 10 percent for *o_orderkey* and recollects the statistics

Note that the system-determined THRESHOLD option is still in effect.

If the THRESHOLD option decides to skip the statistics recollection, the dictionary row for this recollection is updated with the new sample percentage without recollecting the statistics.

```
COLLECT STATISTICS
  USING SAMPLE 10 PERCENT
  COLUMN o_orderkey
  ON orders;
```

The next example resets the USING option from system-determined threshold to a new user-specified threshold of 7 days on *o_orderdatetime* and recollects the statistics.

If the new THRESHOLD option decides to skip the recollection, the dictionary row for this recollection is updated with the new THRESHOLD option without recollecting statistics.

```
COLLECT STATISTICS
USING THRESHOLD 7 DAYS
COLUMN o_orderdatetime
ON orders;
```

Example: Overriding USING Options With the FOR CURRENT Option

The FOR CURRENT option is useful for overriding the existing USING options for the current request, but to continue to use those options for future requests.

For example, if a table has massive updates or deletes, and you want to force a recollection of statistics for the current request without honoring the THRESHOLD options currently in effect, specify the NO THRESHOLD FOR CURRENT option to force statistics to be recollected. Subsequent recollections are still subject to the previous THRESHOLD option.

The following examples illustrate this functionality.

The first request uses a user-specified sample percentage of SAMPLE 75 PERCENT for the current recollection of statistics on *orders.o_orderkey*. The database remembers the existing USING options and uses them for subsequent recollections.

```
COLLECT STATISTICS
USING SAMPLE 75 PERCENT FOR CURRENT
COLUMN o_orderkey
ON orders;
```

The next request forces statistics to be recollected on *orders.o_orderdatetime* by ignoring the existing threshold options.

The existing USING options continue to be used for subsequent recollections on *orders.o_orderdatetime*.

```
COLLECT STATISTICS
USING NO THRESHOLD FOR CURRENT
COLUMN o_orderdatetime
ON orders;
```

Example: Collecting Statistics on a Foreign Table Column

Although you cannot collect statistics on the payload JSON column, you can collect statistics on specific data in the payload column of the foreign table by using dot notation and specifying an alias for the column data.

This example collects statistics on the City data in the payload column of the foreign table *s3_tbl*. For the *payload.City*, the statement specifies the alias *City*.

```
COLLECT STATS COLUMN(payload.City) AS City ON s3_tbl;
```

Cast any columns referenced in this statement to reasonable data types to improve performance of COLLECT STATS.

You cannot collect statistics on a \$PATH expression.

Related Information

- COLLECT STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- [BEGIN QUERY LOGGING](#)
- *Teradata Vantage™ - Application Programming Reference*, B035-1090
- *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142
- *Teradata® Viewpoint User Guide*, B035-2206

SHOW STATISTICS

Reports the SQL text for the Optimizer and QCD forms of COLLECT STATISTICS requests that collected the statistics and optionally reports the detailed or summary statistics.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

The privileges required to execute a SHOW STATISTICS request depend on the specific type of request you want to submit.

The following table lists the privileges required to submit a SHOW STATISTICS request with a VALUES clause on various database objects.

Database Object	Privileges Required
base table	SELECT on the base table or its containing database or user.
join index	<ul style="list-style-type: none"> • SELECT on the containing database or user for the join index. or • SELECT on the underlying table set for the join index. or • DUMP on the join index or its containing database or user.
hash index	hash index or the containing database or user for its underlying table.

The DUMP privilege is used to enable an archive of statistics to be reported by a SHOW STATISTICS VALUES request.

The following table lists the privileges required to submit a SHOW STATISTICS request without a VALUES clause on various database objects.

Database Object	Privilege Required
base table	base table or its containing database or user.
join index	underlying table set of the index or its containing database or user.
hash index	underlying table of the index or its containing database or user.

The following table lists the privileges required to submit a SHOW STATISTICS request on various database objects that are protected by row-level security constraints.

Row-level Security-protected Database Object	STATISTICS Privilege and the OVERRIDE SELECT CONSTRAINT Privileges Required
base table	table or its containing database or user.
join index	underlying table set of the index or the containing database or user of the underlying table set.
hash index	underlying table of the index or the containing database or user of the underlying table.

SHOW STATISTICS Syntax

Optimizer Form

```
SHOW [IN XML] [SUMMARY] [CURRENT] { STATISTICS | STATS | STAT }
  [ VALUES [SEQUENCED] ]
  { [ UNIQUE ] INDEX index_name |

    [ UNIQUE ] INDEX [ index_name ] [ALL] ( column_name_1 [,...] )
    [ ORDER BY { VALUES | HASH } ( column_name_2 ) ] |

    COLUMN column_specification
  }
ON table_specification [;]
```

column_specification

```
{ { expression | ( expression [,...] ) } AS statistics_name |

  { statistics_name |
    column_name |
```

```

    PARTITION |
    ( { column_name | PARTITION } [,...] )
  } [ [AS] statistics_name ]
}

```

table_specification

```

{ [TEMPORARY] [ database_name. | user_name. ] table_name_1 |

  [ database_name. | database_name. ] { join_index_name |
hash_index_name }
}

```

QCD Form

```

SHOW [IN XML] { STATISTICS | STATS | STAT }
[ VALUES [ SEQUENCED ] ]
{ [ UNIQUE ] INDEX index_name |

  [ UNIQUE ] INDEX [ index_name ] [ALL] ( column_name_1 [,...] )
  [ ORDER BY { VALUES | HASH } ( column_name_2 ) ] |

  COLUMN column_specification
}
ON table_specification
[ FOR QUERY query_id ]
[ SAMPLEID statistics_id ]
[ USING MODIFIED ] [;]

```

column_specification

```

{ statistics_name |
  column_name |
  PARTITION |
  ( { column_name | PARTITION } [,...] )
} [ [AS] statistics_name ]

```

table_specification

```
[ database_name. | user_name. ]
  { table_name_1 | join_index_name | hash_index_name } FROM qcd_name
```

SHOW STATISTICS Syntax Elements**IN XML**

Reports the output in XML format.

This output format can be useful for advanced processing of optimizer statistics such as graphical display, various transformations, and so on.

The XML schema for the output produced by this option is maintained in:

<http://schemas.teradata.com/queryplan/queryplan.xsd>

SUMMARY

Reports table-level SUMMARY statistics such as cardinality, average block size, and average row size for the specified column or index.

CURRENT

Reports extrapolated table-level summary statistics.

You can only use this option with the Optimizer form of SHOW STATISTICS.

STATISTICS**STATS****STAT**

Statistics to be shown were saved as *statistics_name*.

VALUES

Displays the create text for the specified database object and the detailed statistics that have been collected on it using a COLLECT STATISTICS statement.

You must specify the VALUES keyword to report the detailed statistics for the specified database object.

SEQUENCED

Reports a sequence number with detailed statistics output.

This option is not valid unless you also specify VALUES.

Because the spool file size for each group of statistics is limited to a maximum of 64KB, the database uses the sequence numbers within each set of statistics reported to order the spool file chunks within each set. See *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184 for more information about this.

UNIQUE

The index for which the SQL create text or create text and detailed statistics are to be reported is unique.

INDEX

Report the SQL create text or create text and detailed statistics for an index.

This option is not valid for geospatial indexes.

index_name

Name of the named index for which the SQL create text or create text and detailed statistics are to be reported.

This option is not valid for geospatial indexes.

ALL

The ALL option was used to create the index whose detailed statistics are to be reported.

This option is not valid for geospatial indexes.

column_name_1

Names of the index columns for which detailed statistics are to be reported.

ORDER BY VALUES

The reported index columns that were value-ordered when they were collected.

VALUES is the default.

ORDER BY HASH

The reported index columns that were hash-ordered when they were collected.

column_name_2

Name of the column on which detailed index statistics are to be reported.

column_specification***column_name***

Name of the column on which statistics were collected.

statistics_name

Name specified for the statistics when they were collected by a COLLECT STATISTICS request.

table_specification**TEMPORARY**

The SQL create text or the SQL create text and the collected statistics are to be displayed is for a global temporary table.

database_name***user_name***

Name of the containing database or user for *database_object_name*.

table_name_1

Name of one of the following database objects whose summary statistics and, optionally, detailed statistics are to be reported.

join_index_name

Name of join index whose summary statistics and, optionally, detailed statistics are to be reported.

hash_index_name

Name of the hash index whose summary statistics and, optionally, detailed statistics are to be reported.

qcd_name

Statistics to be retrieved are found in the *QCD. TableStatistics* table of the specified QCD database or user.

FOR QUERY *query_ID*

Unique identifier for the set of statistics to be found in *QCD. TableStatistics. QueryId*.

The default value is 0.

SAMPLEID *statistics_id*

Unique identifier for the given column or index in *QCD.TableStatistics.StatisticsId*.

USING MODIFIED

Reports the modified statistics stored in *QCD.TableStatistics.ModifiedStats*.

If you do not specify USING MODIFIED, the database reports the unmodified statistics stored in *QCD.TableStatistics.StatisticsInfo*.

Usage Notes

Response Sequences for Detailed Statistics (Single Record and Indicator Modes)

The result is returned as a single record parcel consisting of multiple columns whose values retain the underlying data type of the column on which statistics were collected. The columns and their data types for the table attributes reported by a SHOW STATISTICS VALUES request when run in Record or Indicator modes are as follows:

Information Reported Once Across All Returned Histogram Intervals

This set of information is reported once across all the returned histogram intervals.

Attribute	Description	Data Type
Version	The version number for the COLLECT STATISTICS syntax used to collect the statistics	BYTE
Collect TimeStamp	The timestamp for when statistics were first collected for the column or index set.	TIMESTAMP(0)
LastAlter TimeStamp	The timestamp for when statistics were last collected for the column or index set.	TIMESTAMP(0)
DBSVersion	The version number for the Vantage software on your server.	VARCHAR(32)
SamplePct	If the statistics were sampled, the sampling percentage used to collect them.	DECIMAL(5,2)
UsageType	A codes used to indicate whether the usage is Summary or Detailed. <ul style="list-style-type: none"> • S is Summary usage. • D is Detailed usage. 	CHARACTER(1) LATIN

Histogram Header

The information that follows this heading is taken from the histogram header for the interval histogram being reported.

Attribute	Description	Data Type
NumBValues	Number of biased values in the histogram.	SMALLINT
NumEHIntervals	Number of equal height intervals in the histogram.	SMALLINT
NumHistoryIntervals	Number of history records in the histogram.	SMALLINT
NumAmps	Number of AMPs in the system.	INTEGER
NumNulls	Row cardinality in a composite column or index set that has 1 or more nulls in columns in the set.	REAL
NumAllNulls	Row cardinality in a composite column or index set that has nulls for all of the columns in the set.	REAL
AvgAmpRPV	The average rows per value across all AMPs in the system for NUSIs. The database reports a value only for NUSIs. If the column set reported is not a NUSI, the average AMP rows per value reported is 0.	REAL
Min Value	The minimum data value in a histogram interval. The database returns a separate column for each reported column value.	Stored as the native type for the column with the following exceptions. <ul style="list-style-type: none"> • If the minimum value is a non-LATIN CHARACTER or VARCHAR type, the database converts it to UNICODE. • If the minimum value is not a case-specific CHARACTER or VARCHAR type, the database converts it to UPPERCASE. • The database may truncate a minimum value if its size is greater than MAXVALUELENGTH.
Mode Value	The data value that occurs the most frequently in a histogram interval.	Stored as the native type for the column with the following exceptions. <ul style="list-style-type: none"> • If the modal value is a non-LATIN CHARACTER or VARCHAR type, the database converts it to UNICODE.

Attribute	Description	Data Type
	The database returns a separate column for each reported column value.	<ul style="list-style-type: none"> If the modal value is not a case-specific CHARACTER or VARCHAR type, the database converts it to UPPERCASE. The database may truncate a modal value if its size is greater than MAXVALUELENGTH.
Max Value	The maximum data value in a histogram interval. The database returns a separate column for each reported column value.	<p>Stored as the native type for the column with the following exceptions.</p> <ul style="list-style-type: none"> If the maximum value is a non-LATIN CHARACTER or VARCHAR type, the database converts it to UNICODE. If the maximum value is not a case-specific CHARACTER or VARCHAR type, the database converts it to UPPERCASE. The database may truncate a maximum value if its size is greater than MAXVALUELENGTH.
Mode Freq.	The cardinality of the modal data value in a histogram interval.	REAL
Mode FreqPNulls	The cardinality of the modal distinct partial nulls in a histogram interval.	REAL
NumPNullsDistVals	The cardinality of the distinct partial null sets in a histogram interval.	REAL
Total Values	The composite cardinality of all data values in a histogram interval.	REAL
Total Rows	The composite cardinality of all rows in a histogram.	REAL
Biased Value1	The data value of the first biased value in the histogram.	<p>Stored as the native type for the data value of the first biased value in the histogram with the following exceptions.</p> <ul style="list-style-type: none"> If the first biased value is a non-LATIN CHARACTER or VARCHAR type, the database converts it to UNICODE. If the first biased value is not a case-specific CHARACTER or VARCHAR type, the database converts it to UPPERCASE. The database may truncate a biased value if its size is greater than MAXVALUELENGTH.
Biased Freq1	The frequency of occurrence of the first biased value.	REAL
...		

Attribute	Description	Data Type
Biased Value <i>n</i>	The data value of the <i>n</i> th biased value in the histogram.	Stored as the native type for the data value of the <i>n</i> th biased value in the histogram with the following exceptions. <ul style="list-style-type: none"> • If the <i>n</i> th biased value is a non-LATIN CHARACTER or VARCHAR type, the database converts it to UNICODE. • If the <i>n</i> th biased value is not a case-specific CHARACTER or VARCHAR type, the database converts it to UPPERCASE. • The database may truncate a biased value if its size is greater than MAXVALUELENGTH.
Biased Freq <i>n</i>	The frequency of occurrence of the <i>n</i> th biased value.	REAL

Equal-Height Interval [1]

The information that follows this heading is taken from the first equal-height interval in the histogram being reported.

Attribute	Description	Data Type
Max Value	The maximum data value in the first equal height interval in the histogram. The database returns a separate column for each reported column value.	Stored as the native type for the column with the following exceptions. <ul style="list-style-type: none"> • If the maximum value is a non-LATIN CHARACTER or VARCHAR type, the database converts it to UNICODE. • If the maximum value is not a case-specific CHARACTER or VARCHAR type, the database converts it to UPPERCASE. • The database may truncate a maximum value if its size is greater than MAXVALUELENGTH.
Mode Value	The modal data value in the first equal height interval in the histogram. The database returns a separate column for each reported column value.	Stored as the native type for the column with the following exceptions. <ul style="list-style-type: none"> • If the modal value is a non-LATIN CHARACTER or VARCHAR type, the database converts it to UNICODE. • If the modal value is not a case-specific CHARACTER or VARCHAR type, the database converts it to UPPERCASE. • The database may truncate a modal value if its size is greater than MAXVALUELENGTH.
Mode Freq	The frequency of the modal data value in the first equal height interval in the histogram.	REAL
LowFrequency	The lowest frequency in the equal height interval in the histogram.	REAL

Attribute	Description	Data Type
	The Optimizer uses this value to determine the deviation of the values in an equal-height interval compared with its modal and lowest frequencies.	
Other Values	The cardinality of values other than modal values in the interval.	REAL
Other Rows	The cardinality of rows other than modal value rows in the interval.	REAL

Equal-Height Interval [2]

The information that follows this heading is taken from the second equal-height interval in the histogram being reported.

Attribute	Description	Data Type
Max Value	The maximum data value in the second equal height interval in the histogram. The database returns a separate column for each reported column value.	Stored as the native type for the column with the following exceptions. <ul style="list-style-type: none"> • If the maximum value is a non-LATIN CHARACTER or VARCHAR type, the database converts it to UNICODE. • If the maximum value is not a case-specific CHARACTER or VARCHAR type, the database converts it to UPPERCASE. • The database may truncate a maximum value if its size is greater than MAXVALUELENGTH.
Mode Value	The modal data value in the second equal height interval in the histogram. The database returns a separate column for each reported column value.	Stored as the native type for the column with the following exceptions. <ul style="list-style-type: none"> • If the modal value is a non-LATIN CHARACTER or VARCHAR type, the database converts it to UNICODE. • If the modal value is not a case-specific CHARACTER or VARCHAR type, the database converts it to UPPERCASE. • The database may truncate a modal value if its size is greater than MAXVALUELENGTH.
Mode Freq	The frequency of the modal data value in the second equal height interval in the histogram.	REAL
Other Values	The cardinality of values other than modal values in the interval.	REAL
Other Rows	The cardinality of rows other than modal value rows in the interval.	REAL

Geospatial Histogram Header

The information that follows this heading is taken from the histogram header being reported.

Even though statistics can be collected on columns of 3D geospatial data, statistics are collected only for the X and Y coordinate values. The Z coordinate values are ignored for statistics collection.

Attribute	Description	Data Type
Version	The Version Number for this GeoStatsHeader. Incremented each time new data members are added or altered within this header structure.	SMALLINT
NumAMPs	Number of AMPs that existed at COLLECT STATS time.	SMALLINT
SizeOfLowResGeoGrid	Cell count for the Low Resolution Grid.	INTEGER
SizeOfHighResGeoGrid	Cell count for the original High Resolution Grid.	INTEGER
NumberHighResCells	Actual number of high resolution grid cells being stored.	INTEGER
MBRxLow	MBR (Minimum Bounding Rectangle) surrounding all RTREE entries, Min X value.	REAL
MBRyLow	MBR surrounding all RTREE entries, Min Y value.	REAL
MBRxHigh	MBR surrounding all RTREE entries Max X value.	REAL
MBRyHigh	MBR surrounding all RTREE entries Max Y value.	REAL
SkewMBRxLow	MBR surrounding the statistics Min X value.	REAL
SkewMBRyLow	MBR surrounding the statistics Min Y value.	REAL
SkewMBRxHigh	MBR surrounding the statistics Max X value.	REAL
SkewMBRyHigh	MBR surrounding the statistics Max Y value.	REAL
NumberNulls	Number of nulls.	BIGINT
NumberRows	Total number of rows including nulls.	BIGINT
NumberOutliers	Number of entries not counted in the SkewMBR.	BIGINT
AvgMBRWidth	Average Width of the MBR of all rows.	REAL
AvgMBRHeight	Average Height of the MBR of all rows.	REAL
TotalIC	Total Intersects Count in the Low Resolution Grid.	BIGINT
TotalDC	Total Direct Count in the Low Resolution Grid.	BIGINT
MudCount	Count of the Mudpiles of the rows.	REAL
NumberLevels	Maximum Number of levels in the RTrees.	BIGINT
NumberLeafBlks	Maximum Number of Leaf Virtual Blocks in the RTrees.	BIGINT

Attribute	Description	Data Type
NumberLeafRows	Maximum Number of Leaf Rows in the RTrees.	BIGINT
NumberLeafEntries	Maximum Number of Leaf Entries in the RTrees.	BIGINT
LeafRowSpaceUtilization	Maximum Space Utilization of Leaf Rows in the RTrees.	BIGINT
NumberDirBlks	Maximum Number of Directory Virtual Blocks in the RTrees.	BIGINT
NumberDirRows	Maximum Number of Directory Rows in the RTrees.	BIGINT
NumberPoints	Number of Points in all rows.	REAL
NumberLineString	Number of LineStrings in all rows.	REAL
AvgSizeLineString	Average number of Points in the LineStrings in all rows.	BIGINT
NumberPolygon	Number of Polygons in all rows.	REAL
AvgSizePolygon	Average number of Points in the Polygons in all rows.	BIGINT
NumberGeomColl	Number of Geometry Collections in all rows.	REAL
AvgSizeGeomColl	Average number of Points in the Geometry Collections in all rows.	BIGINT
NumberMultiPoint	Number of MultiPoints in all rows.	REAL
AvgSizeMultiPoint	Average number of Points in the MultiPoints in all rows.	BIGINT
NumberMultiLineString	Number of MultiLineStrings in all rows.	REAL
AvgSizeMultiLineString	Average number of Points in the MultiLineStrings in all rows.	BIGINT
NumberMultiPolygon	Number of MultiPolygons in all rows.	REAL
AvgSizeMultiPolygon	Average number of Points in the MultiPolygons in all rows.	BIGINT
LowResCells	Below is a list of low resolution cell attributes.	
Distinct Count	Distinct Count in the cell.	REAL
Intersects Count	Intersects Count in the cell.	REAL
HighResCells	Below is a list of high resolution cell attributes.	
CellID	Cell ID.	INTEGER
Distinct Count	Distinct Count in the cell.	REAL
Intersects Count	Intersects Count in the cell.	REAL

For information about Record and Indicator modes, see *Basic Teradata® Query Reference*, B035-2414, *Teradata® Call-Level Interface Version 2 Reference for Mainframe-Attached Systems*, B035-2417 or *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418.

Response Sequences for Detailed Statistics (Multiple Records and Indicator Modes)

The result is returned as multiple record parcels (one per interval, including the master record) consisting of multiple fields whose values retain the underlying data type of the column on which statistics were collected. The summary set of information across all intervals is repeated for each interval. This information is reported by the following attributes:

- Date of statistics collection
- Time of statistics collection
- Number of rows on which statistics were collected for the specified column set
- Number of rows having at least one null in the specified column set
- Number of rows having all nulls in the specified column set
- Number of AMPs from which statistics were collected
- Overall average AMP rows per value
- Estimated cardinality of the table based on a single-AMP sample
- Estimated cardinality of the table based on an all-AMP sample
- Number of statistical intervals in the histogram

Rules and Guidelines for SHOW STATISTICS Requests

The following rules and guidelines apply to SHOW STATISTICS requests.

The database groups columns or indexes having the same USING options together in a single COLLECT STATISTICS request when it reports the SQL text for a database object.

If there are multiple columns with different USING options, the database reports multiple COLLECT STATISTICS requests with the corresponding using options.

The VALUES option reports detailed statistics for the specified database object. If you do not specify any columns or indexes, the database reports detailed statistics for all of the columns in the database object.

When you specify the SUMMARY option, the database reports only the table-level summary statistics for the specified table or index.

If you submit your SHOW STATISTICS request without also specifying the IN XML option, you can do any of the following things with the reported statistics.

- Use the detailed statistics reported by the VALUES option as a backup of those statistics.
- Submit them to the database as they are reported.
- Transfer them to other systems in dual-active environments.

If you submit your SHOW STATISTICS request and specify the IN XML option, you can use the output for advanced processing of statistical data using methods such as graphical displays, transformations, and so on.

The database displays date, time and timestamp data in UTC format.

No conversion to local time zone is done when statistics are exported.

Similarly, when imported using COLLECT STATISTICS with VALUES clause, the Optimizer does not do any conversions to UTC. It assumes the incoming data is formatted in UTC. This ensures that the data is consistent during exports made using SHOW STATISTICS requests and imports resubmitting the SHOW STATISTICS output to the database operations irrespective of the session time zone.

Examples

Example: SHOW STATISTICS for COLLECT STATISTICS SQL Text Only

Suppose you have collected the following statistics on table *orders*.

```
COLLECT STATISTICS
  COLUMN o_orderdatetime ON orders;
COLLECT STATISTICS
  USING SAMPLE
  COLUMN CAST(o_orderdatetime AS DATE) AS orderdate ON orders;
COLLECT STATISTICS
  USING SAMPLE
  COLUMN o_orderpriority ON orders;
COLLECT STATISTICS
  USING SAMPLE 10 PERCENT
  COLUMN o_custkey ON orders;
COLLECT STATISTICS
  USING SAMPLE 10 PERCENT
  COLUMN o_orderkey ON orders;
COLLECT STATISTICS
  USING MAXINTERVALS 250 AND MAXVALUELENGTH 15
  COLUMN (o_orderdatetime, o_orderkey) ON orders;
COLLECT STATISTICS
  COLUMN o_totalprice ON orders;
```

The following SHOW STATISTICS request does not specify the VALUES option, so it reports only the SQL text for the COLLECT STATISTICS requests listed above, grouping them by the USING option used to collect the statistics.

```
SHOW STATISTICS ON orders;
COLLECT STATISTICS
USING NO SYSTEM SAMPLE
AND SYSTEM MAXINTERVALS
AND SYSTEM MAXVALUELENGTH
```

```

    COLUMN o_orderdatetime,
    COLUMN o_totalprice
ON orders;
    COLLECT STATISTICS
USING SYSTEM SAMPLE
AND SYSTEM MAXINTERVALS
AND SYSTEM MAXVALUELENGTH
    COLUMN CAST(o_orderdatetime AS DATE) AS OrderDate,
    COLUMN o_orderpriority
ON orders;
    COLLECT STATISTICS
USING SAMPLE 10 PERCENT
AND SYSTEM MAXINTERVALS
AND SYSTEM MAXVALUELENGTH
    COLUMN o_custkey,
    COLUMN o_orderkey
ON orders;
    COLLECT STATISTICS
USING NO SYSTEM SAMPLE
AND MAXINTERVALS 250
AND MAXVALUELENGTH 15
    COLUMN (o_orderdatetime, o_orderkey)
ON orders;

```

Example: SHOW STATISTICS SEQUENCED Option

This example demonstrates how to use the SEQUENCED option with a SHOW STATISTICS (Optimizer Form) request.

The following SHOW STATISTICS request displays the statistics on columns *x1* and *y1* from *table_1* with sequence numbers.

```
SHOW STATISTICS VALUES SEQUENCED COLUMN x1, COLUMN y1 ON table_1;
```

The request produces a wide 2-column report with the headings *SequenceNumber* and *RequestText*, where the *RequestText* column also reports the current detailed statistics for columns *x1* and *y1* from *table_1*.

Example: SHOW STATISTICS with a VALUES Clause for a Table With Statistics Collected on a Geospatial Column

Consider the following COLLECT STATISTICS request.

```
COLLECT STATISTICS
  COLUMN o_warehouseloc
ON orders;
```

The following SHOW STATISTICS request reports COLLECT STATISTICS SQL text with a VALUES clause for column `o_warehouseloc`, a geospatial column that contains points that specify the longitude and latitude for the warehouse from which the order is to be shipped.

```
SHOW STATISTICS VALUES COLUMN o_warehouseloc ON orders;
COLLECT STATISTICS
  COLUMN "o_warehouseloc"
ON "orders"
VALUES
(
'EIXSTATKIND', 'GEOSPATIAL', TIMESTAMP '2013-05-01 20:35:04',
'Version', 1,
'NumAMPs', 4,
'SizeOfLowResGeoGrid', 1024,
'SizeOfHighResGeoGrid', 16384,
'NumberHighResCells', 0,
'MBRxLow', 2.00000000000000E 000,
'MBRyLow', 2.00000000000000E 000,
'MBRxHigh', 2.70000000000000E 001,
'MBRyHigh', 2.70000000000000E 001,
'SkewMBRxLow', 2.00000000000000E 000,
'SkewMBRyLow', 2.00000000000000E 000,
'SkewMBRxHigh', 1.68437500000000E 001,
'SkewMBRyHigh', 1.60625000000000E 001,
'NumberNulls', 0,
'NumberRows', 174,
'NumberOutliers', 7,
'AvgMBRWidth', 7.00000000000000E 000,
'AvgMBRHeight', 7.00000000000000E 000,
'TotalIC', 42522,
'TotalDC', 0,
'MudCount', 1.80000000000000E 001,
'NumberLevels', 2,
'NumberLeafBlks', 1,
'NumberLeafRows', 3,
'NumberLeafEntries', 44,
'LeafRowSpaceUtilization', 568,
'NumberDirBlks', 1,
'NumberDirRows', 1,
```

```
'NumberPoints', 0.000000000000000E 000,  
'NumberLineString', 0.000000000000000E 000,  
'AvgSizeLineString', 0,  
'NumberPolygon', 1.740000000000000E 002,  
'AvgSizePolygon', 125,  
'NumberGeomColl', 0.000000000000000E 000,  
'AvgSizeGeomColl', 0,  
'NumberMultiPoint', 0.000000000000000E 000,  
'AvgSizeMultiPoint', 0,  
'NumberMultiLineString', 0.000000000000000E 000,  
'AvgSizeMultiLineString', 0,  
'NumberMultiPolygon', 0.000000000000000E 000,  
'AvgSizeMultiPolygon', 0,  
'LowResCells', 0.000000000000000E 000,  
1.620000000000000E 002, 0.000000000000000E 000,  
1.620000000000000E 002, 0.000000000000000E 000,  
1.620000000000000E 002, 0.000000000000000E 000,  
1.620000000000000E 002, 0.000000000000000E 000,  
1.620000000000000E 002, 0.000000000000000E 000,  
1.620000000000000E 002, 0.000000000000000E 000,  
1.620000000000000E 002, 0.000000000000000E 000,  
1.620000000000000E 002, 0.000000000000000E 000,  
1.620000000000000E 002, 0.000000000000000E 000,  
1.620000000000000E 002, 0.000000000000000E 000,  
  
      o                                o  
      o                                o  
      o                                o  
  
);
```

Example: SHOW SUMMARY STATISTICS with VALUES Clause

The following SHOW SUMMARY STATISTICS request displays the COLLECT STATISTICS request with VALUES clause for all the statistics on the *orders* table. When statistics are collected or refreshed on any column, the database automatically updates the table-level information. The details of the table-level statistical information are displayed when you submit a SHOW STATISTICS request with a SUMMARY clause as indicated in this example.

```
SHOW SUMMARY STATISTICS VALUES
ON orders;
```

Table-level demographics information.

```

      COLLECT SUMMARY STATISTICS
            ON "Sales"."Orders"
      VALUES
(
/* Version          */ 1,
/* NumIntervals     */ 1,
/* Interval[1]*/
/* IntervalType     */ 0,
/* Collect TimeStamp */  TIMESTAMP '2009-07-20 19:47:00',
/* OneAMPSampleEst  */ 950,
/* AllAMPSampleEst  */ 990,
/* RowCount         */ 1000,
/* AvgRowsPerBlock  */ 20,
/* AvgBlockSize     */ 61440,
/* UncompressCPUCost */ 0,
/* BLCURatio        */ 1
);

```

Detailed statistics on column *o_orderkey*.

```

      COLLECT STATISTICS
            USING SYSTEM SAMPLE
            AND MAXINTERVALS 50
      COLUMN "O_OrderKey"
            ON "Sales"."Orders"
      VALUES
(
/* Version          */ 5,
/* Collect TimeStamp */  TIMESTAMP '2009-07-20 19:47:00',
/* LastAlter TimeStamp*/  TIMESTAMP '2009-07-20 19:47:00',
/* DBSVersion       */  '14.00.00.00'
/* SamplePct        */ 100,
/* UsageType        */  'S'
/* Histogram Header */
/* NumBValues        */ 12
/* NumEHIntervals    */ 2,
/* NumHistoryIntervals */ 0,
/* NumAmps           */ 130,
/* NumNulls          */ 0,
/* NumAllNulls       */ 0,
/* AvgAmpRPV         */ 0.000000,
/* Min. Value        */ 101,
/* Mode Value        */ 103,

```

```

/* Max Value          */ 506
/* Mode Freq.         */ 2562150610,
/* Mode FreqPNulls    */ 0,
/* NumPNullsDistVals */ 0,
/* Total Values       */ 164,
/* Total Rows         */ 5179858957,
/* Biased Value1      */ 101,
/* Biased Freq1       */ 1862561589,
/* Biased Value2      */ 102,
/* Biased Freq2       */ 167221704,
/* Biased Value3      */ 103,
/* Biased Freq3       */ 2562150610,
/* Biased Value4      */ 108,
/* Biased Freq4       */ 415231432,
/* Biased Value5      */ 115,
/* Biased Freq5       */ 8712443,
/* Biased Value6      */ 116,
/* Biased Freq6       */ 68543086,
/* Biased Value7      */ 500,
/* Biased Freq7       */ 16633628,
/* Biased Value8      */ 501,
/* Biased Freq8       */ 5461522,
/* Biased Value9      */ 502,
/* Biased Freq9       */ 20504730,
/* Biased Value10     */ 503,
/* Biased Freq10      */ 41812047,
/* Biased Value11     */ 504,
/* Biased Freq11      */ 10215418,
/* Biased Value12     */ 506,
/* Biased Freq12      */ 778243
/* Equal-Height Interval[1]*/
/* Max Value*/          250
/* Mode Value*/          200
/* Mode Freq */          5005
/* Other Values*/        50
/* Other Rows */         10000
/* Equal-Height Interval[2]*/
/* Max Value*/          500
/* Mode Value*/          400
/* Mode Freq */          2500
/* Other Values*/        100
/* Other Rows */         15000
);

```

Example: Reporting Detailed QCD Statistics

This example requests all of the detailed QCD statistics on the *employee* table in database *personnel* from the QCD database *my_qcd*.

```
SHOW STATISTICS VALUES ON personnel.employee FROM my_qcd;
```

Example: SHOW STATISTICS with a VALUES Clause for a Table With Statistics Collected on a Geospatial Column

Consider the following COLLECT STATISTICS request.

```
COLLECT STATISTICS
  COLUMN o_warehouseloc
ON orders;
```

The following SHOW STATISTICS request reports COLLECT STATISTICS SQL text with a VALUES clause for column *o_warehouseloc*, a geospatial column that contains points that specify the longitude and latitude for the warehouse from which the order is to be shipped.

```
SHOW STATISTICS VALUES COLUMN o_warehouseloc ON orders;
COLLECT STATISTICS
  COLUMN "o_warehouseloc"
ON "orders"
VALUES
(
'EIXSTATKIND', 'GEOSPATIAL', TIMESTAMP '2013-05-01 20:35:04',
'Version', 1,
'NumAMPs', 4,
'SizeOfLowResGeoGrid', 1024,
'SizeOfHighResGeoGrid', 16384,
'NumberHighResCells', 0,
'MBRxLow', 2.00000000000000E 000,
'MBRyLow', 2.00000000000000E 000,
'MBRxHigh', 2.70000000000000E 001,
'MBRyHigh', 2.70000000000000E 001,
'SkewMBRxLow', 2.00000000000000E 000,
'SkewMBRyLow', 2.00000000000000E 000,
'SkewMBRxHigh', 1.68437500000000E 001,
'SkewMBRyHigh', 1.60625000000000E 001,
```

[illegible]

Related Information

- SHOW STATISTICS in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- HELP STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- HELP STATISTICS (QCD Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- COLLECT STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- COLLECT STATISTICS (QCD Form) in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146
- *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142

DROP STATISTICS (Optimizer Form)

Drops the statistical data that was collected for specified columns of a table, hash index, or join index by a COLLECT STATISTICS (Optimizer Form) request. To drop the SUMMARY statistics for a database object, drop all statistics on that object.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Other SQL dialects support similar non-ANSI standard statements with names such as DISASSOCIATE STATISTICS.

Required Privileges

To DROP STATISTICS on a ...	You must have this privilege ...	On ...
permanent or base global temporary table	STATISTICS	the table.
materialized global temporary table	none.	none.
volatile table	none.	none.
join index	STATISTICS	the join index.
hash index	STATISTICS	the hash index.

DROP STATISTICS Syntax (Optimizer Form)

```
DROP { STATISTICS | STATS | STAT } [ statistic ][,...] ON indexed_item [;]
```

statistic

```
{ [UNIQUE] INDEX [ index_name ] [ALL] ( column_name_1 [,...] )
  [ ORDER BY [ VALUES | HASH ] ( column_name_2 ) ] |

  COLUMN { column_index | ( column_index [,...] ) }
}
```

indexed_item

```
{ [TEMPORARY] [ database_name. | user_name. ] table_name |
  [ database_name. | user_name. ] { join_index_name | hash_index_name }
}
```

column_index

```
{ column_name [ [AS] statistics_name ] | PARTITION }
```

DROP STATISTICS Syntax Elements (Optimizer Form)

statistic

Named index for which statistics are to be dropped.

This does not drop SUMMARY statistics for the specified index. To drop SUMMARY statistics on a database object, you must drop all statistics for the object.

Note that you specify column names with a named index only when it is defined with an ORDER BY clause.

UNIQUE

The index for which statistics are to be dropped is unique.

index_name

Name of the index.

ALL

ALL was used to create the index.

column_name_1

Names of one or more columns on which the index exists and for which statistics are to be dropped.

ORDER BY

ORDER BY was used to create the index.

ORDER BY orders the index by the contents of a single column. If you specify ORDER BY but not VALUES or HASH, then VALUES is assumed.

VALUES

ORDER BY VALUES was used to create the index.

VALUES can order a single numeric column of four bytes or less.

HASH

ORDER BY HASH was used to create the index.

HASH hash-orders a single column instead of hash-ordering all columns (the default).

column_name_2

Name of a column on which the index was ordered.

indexed_item**TEMPORARY**

Statistics are to be dropped for a materialized global temporary table.

This keyword is not valid for permanent tables and if you use it with a table defined as permanent, an error is returned.

If you do not specify TEMPORARY, but the referenced table is a global temporary table, the statistics are dropped for an empty table, but on the base global temporary table rather than its materialized instance.

If you specify TEMPORARY correctly, but no materialized instance of the global temporary table exists in the session, the system materializes an instance with all of the statistics defined on the base temporary table before dropping statistics for it.

database_name

Containing database for *table_name*, *join_index_name*, or *hash_index_name* if something other than the current database.

user_name

Containing user for *table_name*, *join_index_name*, or *hash_index_name* if something other than the current database.

table_name

Name of one of the following database objects whose statistics are to be dropped.

- Table or global temporary table
- Single-table view

You can only request HELP STATISTICS reports for single-table views.

- Single-table query

You can only request HELP STATISTICS reports for single-table queries.

You cannot drop statistics on journal or global temporary trace tables because you cannot collect statistics on those database objects.

join_index_name

Name of the join index for which column or index statistics are to be dropped.

hash_index_name

Name of the hash index for which column or index statistics are to be dropped.

column_index***column_name***

Names of the columns for which statistics are to be dropped.

expression

Column expression that was used to collect the statistics to be dropped.

statistics_name

Name of the statistics collected on *column_name*.

Name of the named column statistics to be dropped. Statistics to be dropped were saved as *statistics_name*.

PARTITION

Column partition statistics are to be dropped.

column_index

Statistics are to be dropped for the specified column set.

This does not drop SUMMARY statistics for the specified column. To drop SUMMARY statistics on a database object, you must drop all statistics for the object.

column_name

Names of the columns for which statistics are to be dropped.

expression

Column expression that was used to collect the statistics to be dropped.

statistics_name

Name of the statistics collected on *column_name*.

Name of the named column statistics to be dropped. Statistics to be dropped were saved as *statistics_name*.

PARTITION

Column partition statistics are to be dropped.

DROP STATISTICS Examples (Optimizer Form)

Example: Dropping Statistics on a Composite Index

This request drops statistics on a composite index defined on the *emp_no* and *name* columns of the *employee* table.

```
DROP STATISTICS INDEX(emp_no, name) ON employee;
```

Example: Dropping Statistics on a Named Index

This example drops any statistics on an index named *unique_1*.

```
DROP STATISTICS INDEX unique_1 ON table_1;
```

Example: Dropping Statistics on an Unnamed Index

This example drops any statistics on an unnamed composite index defined on *column_1* and *column_2*.

```
DROP STATISTICS INDEX (column_1,column_2) ON table_1;
```

Example: Dropping All Statistics for a Table

This request drops any statistics that exist for the *employee* table.

```
DROP STATISTICS ON employee;
```

Example: Dropping PARTITION Statistics

This DROP STATISTICS request drops statistics on the PARTITION column only from the PPI table named *table_1*.

```
DROP STATISTICS COLUMN PARTITION ON table_1;
```

This DROP STATISTICS request drops statistics on the column named *column_1* and the PARTITION column from the PPI table named *table_2*:

```
DROP STATISTICS COLUMN (column_1, PARTITION) ON table_2;
```

This table-level DROP STATISTICS request drops all the statistics, including PARTITION statistics, from the PPI table named *table_3*:

```
DROP STATISTICS ON table_3;
```

Example: Dropping a Mix of Single-Column and Multicolumn Statistics

Assume that you collect the following statistics on *orders*.

```
COLLECT STATISTICS COLUMN o_orderdatetime,  
                        COLUMN o_orderkey,  
                        COLUMN (o_orderdatetime, o_orderkey),  
                        COLUMN (o_orderkey, o_orderdatetime)  
ON orders;
```

The following request drops the single-column statistics on (*o_orderdatetime*), (*o_orderkey*) and multicolumn statistics on (*o_orderdatetime*, *o_orderkey*) from *orders*.

```
DROP STATISTICS COLUMN o_orderdatetime,  
                        COLUMN o_orderkey,  
                        COLUMN (o_orderdatetime, o_orderkey)  
ON orders;
```

Example: Dropping Statistics on a Geospatial Column

This DROP STATISTICS request drops statistics on the geospatial column *sp* in table *stat_t2*.

```
DROP STATISTICS COLUMN(sp) ON stat_t2;
```

Related Information

- COLLECT STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- HELP STATISTICS (Optimizer Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- SHOW STATISTICS in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142

HELP STATISTICS (Optimizer Form)

Reports the summary statistics that have been collected in *DBC.StatsTbl* for a table, join index, or hash index.

Submit a SHOW STATISTICS request and specify the VALUES option to report detailed attributes for the statistics that have been collected for the following:

- table
- join index
- hash index

You can also use SHOW STATISTICS to report SUMMARY statistics for a table, join index, or hash index.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have one of the following privileges to perform the standard form of HELP STATISTICS.

To submit a HELP STATISTICS request on a ...	You must have this privilege ...	On ...
permanent or global temporary table	<ul style="list-style-type: none"> • any or • be the owner 	the specified table.
volatile table	none.	none.
join index	<ul style="list-style-type: none"> • any or • be the owner 	the specified join index.
hash index	any	the specified hash index.

To submit a HELP STATISTICS request on a ...	You must have this privilege ...	On ...
table that is protected by row-level security	STATISTICS and OVERRIDE SELECT CONSTRAINT	the specified table.

Use either the STATISTICS privilege or the SHOW privilege to enable a user to perform HELP or SHOW requests only against the optimizer statistics collected on a specified database object.

The following table lists the privileges required to submit a HELP STATISTICS request on various database objects that are protected by row-level security constraints.

To execute HELP STATISTICS with a VALUES clause on this row-level security-protected database object ...	You must have the STATISTICS and OVERRIDE SELECT CONSTRAINT privileges on the ...
base table	base table or its containing database or user.
join index	underlying table set of the index or the containing database or user of the underlying table set.
hash index	underlying table of the index or the containing database or user of the underlying table.

HELP STATISTICS Syntax (Optimizer Form)

```
HELP [CURRENT] STATISTICS [ON] {
  [TEMPORARY] [ database_name_1. | user_name_1. ] table_name |
  { database_name_2. | user_name_2. } { join_index_name | hash_index_name }
} [;]
```

HELP STATISTICS Syntax Elements (Optimizer Form)

CURRENT

Report extrapolated table-level summary statistics.

STATISTICS

Report summary statistics.

You cannot request information about detailed statistics using the INDEX or COLUMN options. Instead, you must use SHOW STATISTICS to retrieve this information. See [SHOW STATISTICS](#).

TEMPORARY

Summary statistics information for a temporary table.

database_name_1

Name of the containing database for *database_object_name* if different from the default database.

user_name_1

Name of the containing user for *database_object_name* if different from the default user.

table_name

Name of table for which to display information.

database_name_2

Name of the containing database for the join index or hash index if different from the default database.

user_name_2

Name of the containing user for the join index or hash index if different from the default user.

join_index_name

Name of the join index whose summary statistics are to be reported.

hash_index_name

Name of the hash index whose summary statistics are to be reported.

Usage Notes

How To Report Summary Statistics and Detailed Statistics

HELP STATISTICS returns only summary interval statistics for columns and indexes that have had statistics collected on the specified table.

To report detailed interval statistics, you must use the SHOW STATISTICS statement.

Examples

Example: Summary Statistics on a Database

If statistics have been collected for the employee table in the personnel database, you can use the following request to obtain a list of the columns of the employee table that currently have statistics.

```
HELP STATISTICS personnel.employee;
```

The result shows that statistics were last collected for *YrsExp* on 02/06/03, when there were 17 unique values for the column.

Date	Time	Unique Values	Column Names
-----	-----	-----	-----
02/06/03	13:12:45	17	YrsExp

Suppose you have collected statistics on the following single and multicolumn sets from the orders table.

```
COLLECT STATISTICS COLUMN (o_orderdate),
                        COLUMN (o_orderkey),
                        COLUMN (o_orderdate, o_orderkey),
                        COLUMN (o_orderkey, o_orderdate)
ON orders;
```

The following request displays a summary of statistical information for the orders table.

```
HELP STATISTICS orders;
```

Example: HELP STATISTICS on Multiple Individual Columns

Assume that you have collected the following statistics on an *orders* table.

```
COLLECT STATISTICS COLUMN o_orderdatetime, COLUMN o_orderkey
ON orders;
```

The following request displays summary statistics for the *orders* table. The statistics include the table cardinality, indicated by the report row with the column name *.

```
HELP STATISTICS orders;
*** Help information returned. 3 rows.
```

```

*** Total elapsed time was 1 second.
Date      Time      Unique Values      Column Names
-----
10/08/16 18:14:36      100 *
10/08/16 18:10:48      100 o_orderdatetime
10/08/16 18:05:44      100 o_orderkey

```

Example: Summary Multicolumn Statistics

Consider the following collect statistics request.

```

COLLECT SUMMARY STATISTICS
  COLUMN CAST(o_orderdatetime AS DATE AT 0) AS OrderDate,
  COLUMN (CAST(o_orderdatetime AS DATE AT 0), o_orderkey)
    AS OrderDateAndKey
ON orders;

```

The following request displays a summary of statistical information for the multicolumn statistics using their names, *order_date* and *order_date_and_key*, respectively.

```

HELP STATISTICS orders;
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.
Date      Time      Unique Values      Column Names
-----
10/08/16 18:14:36      100 *
10/08/16 18:10:48      15 order_date
10/08/16 18:05:44      100 order_date_and_key

```

Example: HELP STATISTICS for a Geospatial Column

Consider the following COLLECT STATISTICS request, where *o_spatial_location* is a geospatial column.

```

COLLECT STATISTICS
  COLUMN o_orderdatetime,
  COLUMN o_orderkey,
  COLUMN o_spatial_location
ON orders;

```

The following HELP STATISTICS request returns summary statistical information for *orders* that reports information for the geospatial column. This information includes the table cardinality known to the Optimizer indicated by the row with column value ***.

Because the concept Unique Values is meaningless for Geospatial data types, a HELP STATISTICS request always returns the total number of base table rows in the position taken by a geospatial column, which in this case is `o_spatial_location`.

```
HELP STATISTICS orders;
*** Help information returned. 3 rows.
*** Total elapsed time was 1 second.
```

Date	Time	Unique Values	Column Names
10/08/16	18:14:36	100	*
10/08/16	18:10:48	100	o_orderdatetime
10/08/16	18:05:44	100	o_orderkey
10/08/16	18:19:08	100	o_spatial_location

Related Information

Also see *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142 and *Teradata Vantage™ - Database Design*, B035-1094 for information about statistics collection and recollection.

HELP STATISTICS (QCD Form)

Displays summary or detailed attributes for the statistics that have been collected in the TableStatistics table of a specified query capture database (QCD) for a specified table.

Submit a SHOW STATISTICS request (see [SHOW STATISTICS](#)) and specify the VALUES option to report detailed attributes for the QCD statistics that have been collected for a table, hash index, or join index. You can also use SHOW STATISTICS to report SUMMARY QCD statistics for a table, join index, or hash index.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have one of the following privileges to perform the QCD form of HELP STATISTICS.

- Any privilege on `table_name`.
- Own `table_name` and have SELECT privilege on the TableStatistics table in `QCD_name`.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against the QCD statistics collected on a specified database object.

HELP STATISTICS Syntax (QCD Form)

```
HELP [CURRENT] STATISTICS
  [ON] [ database_name. | user_name. ] object_name
```

```

FROM QCD_name
[ FOR QUERY query_ID ]

[ SAMPLID statistics_ID ]
[ USING MODIFIED ] [;]

```

HELP STATISTICS Syntax Elements (QCD Form)

CURRENT

Reports extrapolated table-level summary QCD statistics.

database_name

user_name

Name of the containing database or user for *object_name* if something other than the current database or user.

object_name

Name of a table, join index, or hash index for which QCD statistics information is required.

QCD_name

Specifies that the statistics to be retrieved are found in the TableStatistics table of the specified QCD database.

query_ID

Value inserted into QCD.TableStatistics.QueryId by the database.

Each unique composite of *query_ID* and *statistics_ID* identifies a separate set of statistics in QCD.TableStatistics for a given column or index.

The default value is 0.

statistics_ID

Value inserted into QCD.TableStatistics.StatisticsId by the database.

Each unique composite of *query_ID* and *statistics_ID* identifies a separate set of statistics in

QCD.TableStatistics for a given column or index.
The default value is 1.

USING MODIFIED

For reporting the modified statistics stored in QCD.TableStatistics.ModifiedStats.
If you do not specify the USING MODIFIED option, the system reports the unmodified statistics stored in QCD.TableStatistics.StatisticsInfo.

Usage Notes

How To Report Summary Statistics and Detailed Statistics

HELP STATISTICS returns only summary interval statistics for columns and indexes that have had statistics collected on the specified table.

To report detailed interval statistics, you must use the SHOW STATISTICS statement.

Example: Summary Statistics

The report produced by the summary form of HELP STATISTICS (QCD Form) is identical to that produced by the summary form of HELP STATISTICS (Optimizer Form).

Assume you collected statistics on the Employee table at the same time using COLLECT STATISTICS (Optimizer Form) and COLLECT STATISTICS (QCD Form) with a 99 percent sample percentage, then the following request would report the columns and indexes of the Employee table that currently have sampled statistics in the *table_statistics* table of the *my_qcd* database.

```
HELP STATISTICS personnel.employee FROM my_qcd;
```

The result shows that statistics were last collected for YrsExp on 02/06/17, when there were 17 unique values for the column.

Date	Time	Unique Values	Column Names
02/06/17	13:12:45	17	yrs_exp

The report does not reflect sample size. Sample size is mentioned only to ensure that the comparison between the reports is accurate. A significantly smaller sample size could produce somewhat different statistics.

Related Information

- [HELP STATISTICS \(Optimizer Form\)](#)

- [SHOW STATISTICS](#)
- COLLECT STATISTICS (QCD Form) in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146
- DROP STATISTICS (QCD Form) in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146

Also see the information about the Query Capture Facility in *Teradata Vantage™ - SQL Request and Transaction Processing*, B035-1142.

Row-Level Security Constraint Statements

CREATE CONSTRAINT

Creates the SQL row-level security constraint definition and associates it with specific UDFs to enforce that constraint.

Before creating a constraint you must:

- Design the classification system that defines the set of name:value pairs that must be specified for the constraint.
- Create the UDFs that enforce the constraint, which you must specify for SYSLIB.*function_name*.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

There are no ANSI standards for row-level security-related SQL statements.

Required Privileges

You must have the CONSTRAINT DEFINITION privilege to create a row-level security constraint.

Privileges Granted Automatically

None.

CREATE CONSTRAINT Syntax

```
CREATE CONSTRAINT constraint_name data_type ,
  [ [NOT] NULL, ]
  VALUES ( name:value [,...] ) , constraint [,...] [;]
```

Syntax Elements

constraint

```
{ DELETE SYSLIB.delete_function_name |
  INSERT SYSLIB.insert_function_name |
  SELECT SYSLIB.select_function_name |
  UPDATE SYSLIB.update_function_name
}
```


CREATE CONSTRAINT Syntax Elements

constraint_name

The name of the row-level security constraint object. The constraint name becomes a column name if you include the constraint in the definition for a table, view, or index.

Constraint names must follow the system rules for database object names, and must be unique among:

- constraints defined in the system
- column names defined in any table that includes the constraint

For information on naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

data_type

The data type for *constraint_name*. One of the following:

- SMALLINT

Use the SMALLINT data type only for hierarchical (level) constraints.

The range of valid constraint values is from 1 to 10,000 inclusive.

- BYTE(*n*)

Use the BYTE(*n*) data type only for non-hierarchical (category) constraints.

n represents the number of bytes in the constraint value string for a table row. If you do not specify a value for *n*, the default is 1 byte. The maximum value for *n* is 32.

To allow for more categories than the column could otherwise contain, the system automatically expresses each non-hierarchical value as a unique bit position, which allows a table row to contain up to 256 distinct values for the constraint column.

The data type you specify becomes the data type of the associated system-created constraint column in any table to which the constraint is assigned.

For detailed information about the uses of the SMALLINT and BYTE data types when designing classification systems and security constraints, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

NOT

Specifies that a constraint value for a specific table row cannot be null.

NULL

A constraint value for a specific table row can be null.

The default is NULL.

VALUES

The string to follow is a set of *name:value* pairs, which defines either a hierarchical (level) or non-hierarchical (category) classification system.

The set of *name:value* pairs defines the set of classifications for the constraint.

The maximum number of *name:value* pairs depends on whether the constraint is hierarchical (10,000) or non-hierarchical (256).

name

The name of a member of the classification system being defined for the constraint.

One or more of the classification names for the constraint can be assigned to a user or profile to define their access privileges on tables that contain the constraint.

value

The value code for the corresponding *name* in a *name:value* pair.

Value codes appear for each row in the corresponding constraint column of tables protected by the constraint.

For details about name:value pairs and how the value is determined for a table row, see *Teradata Vantage™ - Resource Usage Macros and Tables*, B035-1099.

DELETE, INSERT, SELECT, UPDATE

The SQL operations that can be enforced by a constraint.

You must specify between 1 and 4 SQL operations per constraint definition. Each specified SQL operation must be followed by the *SYSLIB.function_name* that enforces the operation.

You cannot specify an SQL operation type more than once in a constraint.

SYSLIB.function_name

The name of the UDF that enforces the corresponding DELETE, INSERT, SELECT, or UPDATE operation for the constraint.

The specified UDF must be contained within the *SYSLIB* database at the time you specify it in the constraint definition.

Each UDF named in *SYSLIB.function_name* specifications for a constraint defines the minimum classification required to perform the associated SQL operation on each row of a table that includes the constraint.

For more information about implementing row-level security, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

Examples

Example: Creating a Constraint Object for a Hierarchical Classification

The following CREATE CONSTRAINT request defines values and enforcements for a hierarchical (level) classification system, within which the values are hierarchically related.

```
CREATE CONSTRAINT classification_level SMALLINT, NOT NULL,
VALUES (top_secret:4, secret:3, confidential:2, unclassified:1),
INSERT SYSLIB.insert_level,
UPDATE SYSLIB.update_level,
DELETE SYSLIB.delete_level,
SELECT SYSLIB.read_level;
```

Example: Creating a Constraint Object with a Non-Hierarchical Classification

The following CREATE CONSTRAINT request defines values and enforcements for a non-hierarchical (category) classification system, within which the values are independent categories.

```
CREATE CONSTRAINT classification_category BYTE(8),
VALUES (nato:1, united_states:2, canada:3, united_kingdom:4,
       france:5, norway:6),
INSERT SYSLIB.insert_category,
UPDATE SYSLIB.update_category,
DELETE SYSLIB.delete_category,
SELECT SYSLIB.read_category;
```

Related Information

- [HELP SESSION](#)
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100 (a complete description of implementing row-level security, including creation of constraints)

ALTER CONSTRAINT

Modifies the definition of an existing row-level security constraint object.

Note:

You can only ADD, DROP, or REPLACE one UDF per ALTER CONSTRAINT statement.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the CONSTRAINT DEFINITION privilege to alter a row-level security constraint object.

There are no privileges granted automatically.

ALTER CONSTRAINT Syntax

```
ALTER CONSTRAINT constraint_name AS
{ VALUES (name:value [,...]) |
  FUNCTION { { ADD | REPLACE } action SYSLIB.function_name | DROP action }
} [;]
```

action

```
{ DELETE | INSERT | SELECT | UPDATE }
```

ALTER CONSTRAINT Syntax Elements

constraint_name

The name of the row-level security constraint object to be altered.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

VALUES

The string to follow is a set of *name:value* pairs that make up the classification system enforced by the *constraint_name*.

If you specify any *name:value* pairs, the system deletes any previously existing *name:value* pairs for the *constraint_name*.

name

The name of a member of the classification system defined in the *constraint_name*. Must be either:

- a hierarchical level name
- a non-hierarchical category name

value

The value code for the corresponding name in a *name:value* pair.

For details about *name:value* pairs, see [CREATE CONSTRAINT](#).

FUNCTION

This option changes the list of currently existing UDFs for the *constraint_name*.

ADD

To add a new DELETE, INSERT, SELECT, or UPDATE function to *constraint_name*.

There must not be any UDF currently defined for the specified statement type in the constraint object, and the specified UDF must reside in the SYSLIB database.

REPLACE

To replace an existing DELETE, INSERT, SELECT, or UPDATE function in *constraint_name*.

The new UDF replaces the existing function for that statement type and must reside in the SYSLIB database.

action

One of the following:

- **DELETE**
The UDF *function_name* being added, dropped, or replaced enforces row-level security restrictions on the DELETE operation.
- **INSERT**
The UDF *function_name* being added, dropped, or replaced enforces row-level security restrictions on the INSERT operation.
- **SELECT**
The UDF *function_name* being added, dropped, or replaced enforces row-level security restrictions on the SELECT operation.
- **UPDATE**
The UDF *function_name* being added, dropped, or replaced enforces row-level security restrictions on the UPDATE operation.

SYSLIB.function_name

The name of the constraint UDF that is subject to the ADD or REPLACE request.

You cannot specify SYSLIB.*function_name* for the DROP option.

If you specify *function_name*, the specified function must currently exist in the SYSLIB database.

DROP

To drop an existing DELETE, INSERT, SELECT, or UPDATE function from *constraint_name*.

Specify only the statement type for the DROP option, but do not specify a *function_name*. If you specify a *function_name*, the database returns an error to the requestor.

The DROP option only removes the reference to the related UDF from *constraint_name* and does not affect the actual UDF.

ALTER CONSTRAINT Examples

Example: Adding a DELETE Security Policy to a Constraint Object

This ALTER CONSTRAINT example adds a DELETE security policy UDF to the *classification_level* constraint object.

```
ALTER CONSTRAINT classification_level
AS FUNCTION ADD DELETE SYSLIB.deletelevel;
```

Example: Adding a New Set of Name:Value Pairs to a Constraint Object

This ALTER CONSTRAINT example adds the specified set of *name:value* pairs to the *classification_level* constraint object and deletes all of the old *name:value* pairs.

```
ALTER CONSTRAINT classification_level
AS VALUES (topsecret:4, secret:3, confidential:2, unclassified:1);
```

Related Information

- [HELP SESSION](#)
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100

DROP CONSTRAINT

Drops the definition of a row-level security constraint object from the data dictionary.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the CONSTRAINT DEFINITION privilege to drop a row-level security constraint.

Privileges Granted Automatically

None.

DROP CONSTRAINT Syntax

```
DROP CONSTRAINT constraint_name [;]
```

DROP CONSTRAINT Syntax Elements***constraint_name***

Name of row-level security constraint object whose definition to drop from the data dictionary.

Usage Notes**Restrictions on Dropping a Row-Level Security Constraint**

Before you can drop a row-level security constraint, you must remove the constraint from all objects that reference the constraint.

You can use the following example SQL requests to determine which database objects have constraint assignments.

Finding Tables and Indexes with a Security Constraint

1. Query the DBC.Dependency table to find tables that have a security constraint column:

```
SELECT database1name, object1name from dbc.dependency where  
object2name='constraint_name ';
```
2. Run the SHOW TABLE command for each table returned by the query in step 1. The system displays the standard CREATE TABLE for the table, including any indexes that are defined on the table.

Determining Whether a Column is a Security Constraint

You can execute a HELP COLUMN statement against a column to find out whether the column is a security constraint. See [HELP COLUMN](#).

Finding Views that Include a Security Constraint

```
SELECT databasename, tvmlname from dbc.tvml, dbc.dbase  
where
```

```
tablekind='V' and
tvm.databaseid=dbase.databaseid and
tvmid in (select tableid from dbc.tvfields where constraintid in
(select constraintid from dbc.secconstraints
where constraintname='constraint_name'));
```

Finding Users or Profiles with an Assigned Constraint

For users:

```
SELECT UserName from DBC.UsrAsgdSecConstraintsV
where
ConstraintName='constraint_name'
```

For profiles:

```
SELECT ProfileName from DBC.ProfileAsgdSecConstraintsV
where
ConstraintName='constraint_name'
```

Example: Dropping a Row-Level Security Constraint

This request drops a row-level security constraint named *levels*.

```
DROP CONSTRAINT levels;
```

Related Information

- [HELP SESSION](#)
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100

Triggers Statements

ALTER TRIGGER

Enables or disables a trigger or changes its creation timestamp.

Some utilities, such as FastLoad and MultiLoad, cannot operate on tables with enabled triggers. Disabling the triggers enables bulk loading of data into the tables. Although the Teradata Parallel Data Pump utility operates on tables with enabled triggers, performance issues can occur.

Some triggers coordinate actions on a set of tables. If these triggers are disabled for some period of time, the application must ensure that the relationship among the set of tables and the trigger is not lost.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Other SQL dialects support similar non-ANSI standard statements with names such as the following:

- DISABLE TRIGGER
- ENABLE TRIGGER

Required Privileges

To execute ALTER TRIGGER, you must have the DROP TRIGGER privilege on the table on which the trigger is defined, or on the database containing it.

There are no privileges granted automatically.

ALTER TRIGGER Syntax

```
ALTER TRIGGER [ database_name. | user_name. ]  
  { trigger_name { ENABLED | DISABLED | TIMESTAMP } |  
    table_name { ENABLED | DISABLED }  
  } [;]
```

ALTER TRIGGER Syntax Elements

database_name

Name of the database containing the trigger definition if not the default database for the *trigger_name* or *table_name*.

user_name

Name of the user containing the trigger definition if not the default database for the *trigger_name* or *table_name*.

trigger_name

The name of the trigger to be altered.

If you specify *trigger_name* you can alter only one trigger at a time.

ENABLED

The named trigger or all triggers on the named table are to be enabled.

Enabled triggers function as active database objects and follow normal trigger protocol.

DISABLED

The named trigger or all triggers on the named table are to be disabled. Disabled triggers continue to exist as database objects, but cannot execute. For a trigger to execute, it must first be enabled.

TIMESTAMP

The creation timestamp of the named trigger is to be replaced with the current timestamp.

Altering the creation timestamp of a trigger changes its position in the default order of execution, when multiple triggers are defined on a table.

The TIMESTAMP option does not apply to a table.

You can use the TIMESTAMP option to change the order of execution of triggers, particularly triggers that were created without the ORDER clause. If you do not specify a TIMESTAMP, then the existing timestamp for the trigger is not changed.

ALTER TRIGGER uses the current time when adding a timestamp. To change the timings for multiple order-sensitive triggers defined on the same table, you must submit ALTER TRIGGER statements that change the timestamps in the required order to ensure the correct execution precedence.

table_name

The name of the table on which triggers are to be enabled or disabled.

CREATE TRIGGER and REPLACE TRIGGER

Creates a new trigger or replaces a trigger definition.

REPLACE TRIGGER changes the definition for a trigger without having to drop and recreate it.

See *Teradata Vantage™ - Temporal Table Support*, B035-1182 for information about the temporal forms of CREATE TRIGGER and REPLACE TRIGGER.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

Required Privileges

To create a trigger, you must have the following privileges:

- CREATE TRIGGER on both of the following:
 - The database in which the trigger is created.
 - Either the subject table or its containing database.
- SELECT on any column referenced in a WHEN clause or a triggered SQL statement subquery.
- Depending on the triggered SQL statement, INSERT, UPDATE, or DELETE on the triggered SQL statement target table.
- The privileges that would normally be required to execute the individual triggered SQL statements.

Creating or replacing a trigger does not grant trigger-related privileges to either the creator or the immediate owner of that trigger.

To replace a trigger you must have the following privileges:

- DROP TRIGGER on the subject table or the database.

The exception is when you use REPLACE TRIGGER when no target trigger exists and you instead create a new trigger.

In that case, you need the CREATE TRIGGER privilege on both of the following:

- The database in which the trigger is created.
- Either the subject table or its containing database.
- SELECT on any column referenced in a WHEN clause or a triggered SQL statement subquery.
- Depending on the triggered SQL statement, INSERT, UPDATE, or DELETE on the triggered SQL statement target table.
- The privileges that would normally be required to execute the individual triggered SQL statements.

The following privilege requirements apply to any user, other than the immediate owner, who performs a triggering statement:

- You must have the privileges required to execute that triggering statement, and the immediate owner of the trigger must also hold all the privileges required to create the trigger.
- If you have the required privileges for the triggering statement, but the immediate owner of the trigger no longer has the privileges required to perform the triggered action statements, then neither you nor any other user can run the triggering statement.

Privileges Granted Automatically

None.

CREATE TRIGGER and REPLACE TRIGGER Syntax

```
{ CREATE | REPLACE } TRIGGER [ database_name_1. ] trigger_name
  [ ENABLED | DISABLED ]
  { BEFORE | AFTER } [ temporal_option ] triggering_event
  ON [ database_name_2. ] table_name
  [ ORDER integer ]
  REFERENCING reference [...]
  [ FOR EACH { ROW | STATEMENT } ]
  [ WHEN ( search_condition ) ]
  { trigger_action | BEGIN ATOMIC trigger_action END } [;]
```

triggering_event

```
{ INSERT |
  DELETE |
  UPDATE [ OF { column_name [,...] | ( column_name [,...] ) } ]
}
```

reference

```
{ OLD [ ROW ] [AS] old_transition_variable_name |
  NEW [ ROW ] [AS] new_transition_variable_name |
  { OLD_TABLE | OLD TABLE } [AS] old_transition_table_name |
  { NEW_TABLE | NEW TABLE } [AS] new_transition_table_name |
  OLD_NEW_TABLE [AS] old_new_table_name ( old_value, new_value )
}
```

trigger_action

```
{ SQL_procedure_statement [;...] |
  ( SQL_procedure_statement [;...] )
}
```

CREATE TRIGGER and REPLACE TRIGGER Syntax Elements

database_name_1

An optional qualifier if the trigger is being created in a database other than the default for the current user.

trigger_name

Name of the trigger being created or replaced.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

A *trigger_name* must be unique within the database in which it is created.

ENABLED

Keyword that enables a trigger to execute.

ENABLED is the default.

DISABLED

Keyword that disables a trigger from executing.

The definition of a disabled trigger is not dropped, but it must be enabled before it can execute (see [ALTER TRIGGER](#)).

BEFORE

Specifies that the trigger performs before the triggering event, or triggering statement, is executed.

- A BEFORE row trigger cannot have data-changing statements as triggered action SQL statements.
- A BEFORE statement trigger is not valid under any circumstances.

The database returns an error message in both cases.

AFTER

The trigger performs after the triggering event.

temporal_option

See *Teradata Vantage™ - Temporal Table Support*, B035-1182 for documentation of these options.

triggering_event

A triggering event, or triggering statement, can be an INSERT, DELETE, UPDATE, or MERGE request. See *Teradata Vantage™ - Temporal Table Support*, B035-1182 for documentation of the rules that apply to triggering statements for temporal tables.

INSERT

The triggering event for this trigger is one of the following:

- INSERT

- INSERT SELECT
- Atomic Upsert
- MERGE

DELETE

The triggering event for this trigger is a DELETE.

UPDATE

The triggering statement for this trigger is one of the following:

- UPDATE
- Atomic Upsert
- MERGE

Any number of rows, including none, can be updated.

column_name

Name of a column in a set of column names.

The trigger fires when any column in the list is updated.

You can optionally enclose the list in parentheses.

If you do not specify a column name list, the default is all columns.

The list cannot specify duplicate column names.

The *column_name* list does not apply to INSERT or DELETE triggering events.

database_name_2

Optional qualifier for the subject table.

If the subject table is in a database other than the default for the current user, this specification is mandatory.

table_name

Name of the subject table with which this trigger is associated.

table_name must be the name of an existing base table.

The object referenced by *table_name* cannot be any of the following:

- Global temporary table
- Hash index
- Join index
- Recursive view
- Queue table

- Trace table
- View
- Volatile table

ORDER

ORDER values determine the execution sequence when two or more triggers have the identical trigger action time and trigger event.

If triggers have the same ORDER value, trigger action time, and trigger event, then they run in order of their creation timestamp.

integer

Value assigned to ORDER.

The value must be any positive small integer not greater than 32,767.

If the ORDER clause is not specified, the system assigns a default value of 32,767 to the trigger.

reference

The REFERENCING clause allows the WHEN condition and the triggered actions of a trigger to reference the set of rows in one or more transition tables. The clause is optional and has no default.

The semantics of this clause are identical for temporal and nontemporal tables.

OLD [ROW] [AS] *old_transition_variable_name*

Specifies a correlation name for the current row (transition row) before modification.

Valid only for ROW triggers on DELETE, MERGE UPDATE, and UPDATE events.

You can specify a normalize Period column for OLD [ROW].

NEW [ROW] [AS] *new_transition_variable_name*

Specifies a correlation name for the current row (transition row) after modification.

Valid only for ROW triggers on INSERT, MERGE INSERT, MERGE UPDATE, and UPDATE events.

You cannot specify a normalize Period column for NEW [ROW].

OLD_TABLE

OLD TABLE

Equivalent introductions to a table correlation name for the old values transition table.

OLD TABLE is the form specified by the ANSI SQL-2011 standard.

OLD_TABLE is a Teradata extension retained for backward compatibility.

You can specify OLD_TABLE or OLD TABLE references in both statement triggers and row triggers.

You can specify a normalize Period column for OLD TABLE or OLD_TABLE.

OLD_TABLE and OLD TABLE are valid only for DELETE, MERGE UPDATE, and UPDATE triggering events.

old_transition_table_name

Table alias name specified with the OLD TABLE or OLD_TABLE options for referencing the transition table of old values.

NEW_TABLE

NEW TABLE

Equivalent introductions to a table correlation name for the new values transition table.

NEW TABLE is the form specified by the ANSI SQL:2011 standard.

NEW_TABLE is a Teradata extension retained for backward compatibility.

You can specify NEW_TABLE or NEW TABLE references in both statement triggers and row triggers.

You cannot specify a normalize Period column for NEW ROW TABLE or NEW_TABLE.

NEW_TABLE and NEW TABLE are valid only for INSERT, MERGE INSERT, MERGE UPDATE, and UPDATE triggering events.

new_transition_table_name

Table alias name specified with the NEW TABLE or NEW_TABLE options for referencing the transition table of new values.

OLD_NEW_TABLE

Introduction to a table correlation name for the old and new values transition table.

You can only specify an OLD_NEW_TABLE reference for an AFTER UPDATE trigger.

This clause provides two advantages for statement triggers over the NEW TABLE and OLD TABLE syntax:

- Enables a statement trigger to store all the old and new values in a single row.
- Eliminates the need for a self-join between the transition table and itself.
- Enables the capability to compare the old value with its corresponding new value in the affected row rather than comparing the old value with the new values in all affected new rows.

old_new_table_name

Table alias name specified with the OLD_NEW_TABLE option for referencing the transition table of old and new values in the same row.

old_value

An alias you can use to reference specific old value columns of the row in the old new table transition table.

new_value

An alias you can use to reference specific new value columns of the row in the old new table transition table.

FOR EACH ROW

The trigger is to fire for each qualified row. That is, each row that evaluates to TRUE for any WHEN condition specified for the trigger.

FOR EACH STATEMENT is the default.

FOR EACH STATEMENT

The trigger is to fire once per processed SQL statement in the request whenever a WHEN condition for the trigger evaluates to TRUE.

FOR EACH STATEMENT is the default.

WHEN (*search_condition*)

Logic that further refines the conditions for firing the trigger.

search_condition is a Boolean condition that represents a valid search condition based on comparisons of items within the scope of the trigger definition. The trigger fires only if the value of *search_condition* is TRUE.

You must qualify columns in *search_condition* with either *old_transition_variable_name* or *new_transition_variable_name* unless they are in a subquery. For example:

```
CREATE TRIGGER RaiseTrig
  AFTER UPDATE OF salary ON employee
  REFERENCING OLD AS OldRow NEW AS NewRow
  FOR EACH ROW
    WHEN ((NewRow.salary - OldRow.salary) / OldRow.salary >.10)
    INSERT INTO salary_log
    VALUES ('USER', NewRow.name, OldRow.salary, NewRow.salary);
```

search_condition can contain aggregates only in a subquery. When you reference an OLD TABLE or NEW TABLE transition table from a *search_condition*, you must express the

predicate as a subquery. Because a WHEN condition must provide a single result, the typical use of OLD TABLE and NEW TABLE column references is with aggregates.

search_condition can include UDT comparisons only if ordering has been defined for the UDTs. See [CREATE ORDERING and REPLACE ORDERING](#).

If the trigger is a row trigger, then *search_condition* is based on row correlation names for the current row.

search_condition is evaluated once for each:

- Execution of the triggering statement for statement triggers
- Row of the transition table of changed rows for row triggers

trigger_action

The ANSI SQL:2011 specification requires multiple statements to be enclosed within the BEGIN ATOMIC and END keywords. Vantage complies with this, and also allows you to specify multiple SQL procedure statements without the BEGIN ATOMIC and END keywords.

Each triggered action statement in an SQL procedure statement list must be terminated by a SEMICOLON (;) character.

You can specify the list of SQL procedure statements either enclosed or not enclosed by parentheses.

In BTEQ, if you do not enclose the list of SQL procedure statements in parentheses, the line immediately following each statement must follow these rules:

- It must begin with the SEMICOLON character that terminates the preceding line.
- It must specify the next triggered action statement in the list.

UDT expressions are valid in triggered SQL statements.

See *Teradata Vantage™ - Temporal Table Support*, B035-1182 for details and examples of using triggered SQL statements with temporal tables.

BEGIN ATOMIC

A keyword introducing multiple triggered action statements.

If you begin the triggered SQL statement clause with BEGIN ATOMIC, you must terminate it with the END keyword.

SQL_procedure_statement

One or more valid triggered action statements.

END

A keyword terminating the statement block introduced by the BEGIN ATOMIC keywords.

If you use BEGIN ATOMIC, you must specify END to terminate the triggered SQL statements clause.

Examples

Example: Ensuring That Parent Table Updates Propagate to its Child

Suppose you want to ensure that all UPDATES and DELETES to a parent table are propagated to a child table. This example, which is designed to enforce referential integrity procedurally, defines two AFTER statement triggers on the parent table to ensure that primary key updates to the parent table propagate to the appropriate foreign key column in a child table.

These are the table definitions:

```
CREATE TABLE parent_tab (
  prime_key INTEGER,
  column_2  INTEGER,
  column_3  INTEGER)
UNIQUE PRIMARY INDEX (prime_key);
CREATE TABLE child_tab (
  prime_key INTEGER,
  for_key   INTEGER,
  column_3  INTEGER
FOREIGN KEY (for_key) REFERENCES WITH NO CHECK OPTION
  parent_tab(prime_key));
```

Now define the triggers on Parent_Tab, the subject table:

```
CREATE TRIGGER UpdateForKey
  AFTER UPDATE OF prime_key ON parent_tab
  REFERENCING OLD TABLE AS OldTable NEW TABLE AS NewTable
FOR EACH STATEMENT (
  UPDATE child_tab
  SET for_key=NewTable.prime_key
  WHERE child_tab.for_key=OldTable.prime_key);
CREATE TRIGGER DelForKey
  AFTER DELETE ON parent_tab
  REFERENCING OLD TABLE AS OldTable
FOR EACH STATEMENT (
  UPDATE child_tab
  SET for_key=NULL
  WHERE for_key=OldTable.prime_key);
```

Example: Audit Log for Large Pay Raises

Triggers are particularly useful for audits of all kinds. This example shows an AFTER row trigger that inserts a log record whenever an employee gets a raise larger than ten percent.

These are the table definitions:

```
CREATE TABLE employee (
  name      CHARACTER(30),
  dept_Id   INTEGER,
  salary     DECIMAL(10,2),
  comments  CHARACTER(30));
CREATE TABLE salary_log (
  user_name CHARACTER(30),
  emp_name  CHARACTER(30),
  old_salary DECIMAL(10,2),
  new_salary DECIMAL(10,2));
```

Now define the trigger on the *employee* table:

```
CREATE TRIGGER RaiseTrig
  AFTER UPDATE OF salary ON employee
  REFERENCING OLD AS OldRow NEW AS NewRow
  FOR EACH ROW
  WHEN ((NewRow.salary-OldRow.salary)/OldRow.salary >.10)
  INSERT INTO salary_log
  VALUES ('USER', NewRow.name, OldRow.salary, NewRow.salary);
```

When the following requests are processed, two inserts are made to *salary_log*. The third update does not meet the WHEN condition of the trigger, so no corresponding row is inserted in *salary_log*.

```
UPDATE employee
SET salary = salary*1.5, comments = 'Employee of the Year'
WHERE name = 'John Smith';
UPDATE employee
SET salary = salary*2, comments = 'Employee of the Decade'
WHERE name = 'Min Chan';

UPDATE employee
SET salary = salary*1.05, comments = 'Normal midrange raise'
WHERE name = 'Lev Ulyanov';
```

Example: Using a SET Clause

This example uses a SET clause as a triggered action statement in a BEFORE row trigger.

This is the subject table definition. Assume that values are loaded into the amount column of the table from a source file employing a USING INSERT request.

```
CREATE TABLE subject_table (
  entry_date DATE,
  amount      INTEGER);
```

This is the trigger definition:

```
CREATE TRIGGER set_trig
  BEFORE INSERT ON subject_table
  REFERENCING NEW AS curr_value
  FOR EACH ROW
  SET curr_value.entry_date = DATE;
  -- Adds the current system date to the entry_date column
```

Because of the SET clause assignment, the following triggering statement:

```
USING (amount INTEGER)
INSERT INTO subject_table VALUES (NULL,:amount);
```

executes as if it were this statement:

```
USING (thisdate DATE, amount INTEGER)
INSERT INTO subject_table VALUES (:thisdate, :amount);
```

Example: Cascaded Triggers

This example demonstrates how one triggered action statement can cause another trigger to fire.

These are the table definitions.

```
CREATE TABLE tab1 (
  a INTEGER,
  b INTEGER,
  c INTEGER);
CREATE TABLE tab2 (
  d INTEGER,
```

```

    e INTEGER,
    f INTEGER);
CREATE TABLE tab3 (
    g INTEGER,
    h INTEGER,
    i INTEGER);

```

These are the trigger definitions.

```

CREATE TRIGGER trig1
  AFTER INSERT ON tab1
  REFERENCING NEW AS NewRow
FOR EACH ROW (
  INSERT INTO tab2
  VALUES (NewRow.a + 10, NewRow.b + 10, NewRow.c));
CREATE TRIGGER trig2
  AFTER INSERT ON tab2
  REFERENCING NEW AS NewRow
FOR EACH ROW (
  INSERT INTO tab3
  VALUES (NewRow.d + 100, NewRow.e + 100, NewRow.f));

```

Now, suppose the following INSERT request is submitted:

```

INSERT INTO tab1
VALUES (1,2,3);

```

This triggering event fires a trigger to insert into *tab2*. This operation is equivalent to the following INSERT request:

```

INSERT INTO tab2
VALUES (11,12,3);

```

This triggering event fires a trigger to insert into *tab3*. This operation is equivalent to the following INSERT request:

```

INSERT INTO tab3
VALUES (111,112,3);

```

Example: Valid WHEN Clause

The following WHEN clause is valid because an aggregate appears on the right-hand side of the search condition, and the left-hand side of the inequality condition is a constant.

```
CREATE TRIGGER TrigWhen
  AFTER INSERT ON t1
  REFERENCING NEW AS NewRow
FOR EACH ROW
  WHEN (10 > (
    SELECT SUM(b)
    FROM t2
    WHERE t2.c < 5))
ABORT;
```

If you insert values into *t1* and the WHEN condition is satisfied, then the triggered action statement, ABORT, performs and a failure message is returned.

```
INSERT INTO t1 (1, 1, 1);
*** Failure 3514 User-generated transaction ABORT.
```

Example: Non-Valid WHEN Clause

The following CREATE TRIGGER request fails because the WHEN condition contains an aggregate function, but is not specified in a subquery:

```
CREATE TRIGGER trigwhen2
  AFTER INSERT ON t1
  REFERENCING NEW_TABLE AS NewTab
FOR EACH STATEMENT
  WHEN (10 < MAX(NewTab.a))
ABORT;
*** Failure 5430 The Trigger WHEN clause cannot contain an aggregate or
table reference.
```

Example: Valid Use of Subquery in WHEN Clause

The following CREATE TRIGGER request succeeds because its WHEN clause aggregate function is specified as part of a subquery:

Note that the aggregate subquery is on the right hand side of the inequality condition, as it must be.

```
CREATE TRIGGER trigwhen3
  BEFORE INSERT ON t1
  REFERENCING OLD AS previous
FOR EACH ROW
  WHEN (previous.a <= (
    SELECT SUM(b)
    FROM t2, t3
    WHERE t2.c = t3.c))
ABORT;
```

Example: Non-Valid Use of Subquery in WHEN Clause

The following WHEN clause is not valid because the subquery in the search condition returns multiple values and therefore cannot evaluate to either TRUE or FALSE:

```
CREATE TRIGGER trigwhen4
  AFTER INSERT ON t1
  REFERENCING NEW AS NewRow
FOR EACH ROW
  WHEN (t2.a > (
    SELECT b
    FROM t2
    WHERE t2.c < 5))
ABORT;
```

If you insert multiple values into *t1*, the SELECT request returns an “unknown” response, which the WHEN condition cannot evaluate, and the transaction fails.

```
*** Failure 3669 More than one value was returned by a subquery.
```

Example: OLD_NEW_TABLE AFTER UPDATE Trigger

The following OLD_NEW_TABLE trigger definition is equivalent to the OLD_TABLE, NEW_TABLE trigger definition that follows it in terms of functionality; however, the OLD_NEW_TABLE syntax eliminates a join between OLD_TABLE and NEW_TABLE in the subquery, which makes it more high-performing.

```
CREATE TRIGGER inventory_trigger
  AFTER UPDATE ON inventory
  REFERENCING OLD_NEW_TABLE AS OldNewTab (OldValue, NewValue)
  (INSERT INTO InventoryLogTbl
    SELECT OldValue.ProductKey,
```



```

OldValue.AvailQty,                NewValue.AvailQty
    FROM OldNewTab));
CREATE TRIGGER inventory_trigger AFTER UPDATE ON inventory
    REFERENCING OLD_TABLE AS OldTab
                NEW_TABLE AS NewTab
    (INSERT INTO InventoryLogTbl
        SELECT OldTab.ProductKey,
OldTab.AvailQty,                NewTab.AvailQty
        FROM OldTab, NewTab
        WHERE OldTab.ProductKey = NewTab.ProductKey));

```

Example: Calling an SQL Procedure From Within a Trigger

The following trigger is used to track changes to documents. There is a document table and a journal table. As changes are made to the document table, the trigger changes the journal table. A separate application processes and removes rows from the journal table. The trigger ensures that there is at most one row in the journal table for each corresponding row in the document table. So, if a document is changed more than once, then the trigger maintains a single journal row for that document.

```

CREATE TABLE document_table (
    docnum    INTEGER,
    document  BLOB)
UNIQUE PRIMARY INDEX(docnum);
CREATE TABLE document_journal_table (
    docnum    INTEGER,
    action    CHARACTER(1))
UNIQUE PRIMARY INDEX(docnum);
CREATE TRIGGER log_inserts_to_doc AFTER INSERT ON document_table
    REFERENCING new AS new_row
    FOR EACH ROW
    BEGIN ATOMIC
        CALL insert_doc_journal( new_row.docnum );
    END;
CREATE PROCEDURE insert_doc_journal(IN docnum INTEGER)
    BEGIN
        DECLARE num_delete_journal_records INTEGER;
        SELECT COUNT(docnum) INTO :num_delete_journal_records
        FROM document_journal_table
        WHERE docnum=:docnum
        AND    action = 'D';
        IF num_delete_journal_records > 0 THEN
            -- We have a delete journal record.  So, this document
            -- was previously deleted and now they are inserting it.

```

```

-- A delete followed by an insert is really an update so
-- remove the delete journal record and add an update
-- change the journal record
UPDATE document_journal_table
SET action = 'U'
WHERE docnum = :docnum;
ELSE
-- We either have no journal records or we have an insert or
-- update record. An insert followed by an insert is still an
-- insert. An insert followed by an update is really an insert
-- so remove any journal record and add an insert journal      -- record.
UPDATE document_journal_table
SET action = 'I'
WHERE docnum = :docnum;
ELSE INSERT INTO document_journal_table VALUES (:docnum,'I');
END IF;
END;

```

Example: Calling a UDF From a Trigger

The following example calls a user-defined function (internal form) named *firstlerr* from a trigger.

The relevant CREATE TABLE definitions are as follows:

```

CREATE TABLE trg_udf007_04 (
  a INTEGER,
  b FLOAT);
CREATE TABLE trg_udf007_03 (
  a INTEGER,
  b FLOAT);
CREATE TABLE trg_udf007_01 (
  a INTEGER,
  b FLOAT);

```

Insert a row into *trg_udf007_01*:

```

INSERT INTO trg_udf007_01
VALUES (4, 2.5);

```

Create the following UDF body:

```

#define SQL_TEXT Latin_Text
#include <udfdefs.h>

```

```

/* Select statement:
CREATE FUNCTION first1err(integer, float)
RETURNS float
LANGUAGE C
NO SQL
EXTERNAL NAME 'sc!first1err!first1err.c';
*/
void first1err(INTEGER *a,
FLOAT *b,
FLOAT *result,
int *indc_a,
int *indc_b,
int *indc_result,
char sqlstate[6],
SQL_TEXT extname[129],
SQL_TEXT specific_name[129],
SQL_TEXT error_message[257])
{
if (*indc_a == -1 || *indc_b == -1)
{
*indc_result = -1;
return;
}
*result = *a + *b;
*indc_result = 0;
/* handle warning */
if (*a == -1 )
{
strcpy(sqlstate, "01H01");
strcpy((char *) error_message, "You have been warned no nulls");
return;
}

/* create a divide fault */
if (*a == -2 )
{
int f2 = 2;
int f1 = 0;
volatile int f3 = 99999;
f3 = f2/f1;
if (f3 < 0)
f3 = 5;
return;
}
if ( *result < 0.0 )

```

```

{
    strcpy(sqlstate, "22H01");
    strcpy((char *) error_message, "This is a user created error.");
    return;
}
}

```

Create the following trigger:

```

CREATE TRIGGER trg_udf007_01_trigger
  AFTER INSERT ON trg_udf007_03
  REFERENCING NEW AS cur
  FOR EACH ROW
  WHEN ( 11 > firstterr(cur.a, cur.b))
  (INSERT INTO trg_udf007_04 (cur.a, cur.b);
  );

```

Note that the WHEN clause of this trigger definition calls the UDF named *firstterr*.

Perform the following INSERT request:

```

INSERT INTO trg_udf007_03
  SELECT *
  FROM trg_udf007_01;

```

The output produces one row in *trg_udf007_03* and one row in *trg_udf007_04*.

Example: Adding to a Mailing List Using UDTs

The following example adds new California customers to a mailing list using UDTs.

```

CREATE TRIGGER CA_Mailing_list
  AFTER INSERT ON customer
  REFERENCING NEW AS NewRow
  FOR EACH ROW
  WHEN (NewRow.address.state() = 'CA')
  INSERT INTO MailingAddresses
  VALUES (NewRow.Name.Last(), NewRow.Name.First(), NewRow.address);

```

Example: Setting a Default Address to Replace an Inserted Null

The following example sets a UDT employee address to a default if the inserted address is null.

```
CREATE TRIGGER DefaultEmployeeAddress
  BEFORE INSERT ON Employee
  REFERENCING NEW AS NewRow
  FOR EACH ROW
  WHEN (NewRow.address IS NULL)
  SET NewRow.address =
    address().street('17095 Via Del Campo').zip('92127');
```

Example: Dynamic UDT Input From a Trigger

The following example creates an AFTER INSERT row trigger that specifies a NEW VARIANT_TYPE constructor expression in its WHEN clause.

First, create the trigger.

```
CREATE TRIGGER Dyn_TestTrigger02
  AFTER INSERT ON Source_DynTriggerTest2
  REFERENCING NEW AS NewRow
  FOR EACH ROW
  WHEN (scalar001dynRtnint_1p(NEW VARIANT_TYPE(NewRow.a AS a,
                                                NewRow.b AS b))<8)

  INSERT INTO Target_DynTriggerTest2
  VALUES(1,NewRow.a,NewRow.b);
```

Assume that there are no rows in *Target_DynTriggerTest2*. Now insert a row into *Source_DynTriggerTest2*, which invokes the newly created trigger, *Dyn_TestTrigger02*, to insert a row into *Target_DynTriggerTest2* if the evaluation of the scalar UDF *scalar001dynRtnint_1p* satisfies the specified WHEN clause condition.

```
INSERT INTO Source_DynTriggerTest2
VALUES (3,3,3);
```

Select all columns from *Target_DynTriggerTest2*:

```
SELECT *
FROM Target_DynTriggerTest2;
*** Query completed. One row found. 3 columns returned.
*** Total elapsed time was 1 second.
  Integer1      NewColA      NewColB
-----
          1          3          3
```

As expected given the definition for trigger *DynTestTrigger02* , the newly inserted row in *Target_DynTriggerTest2* contains the values 1, 3, and 3 for its three columns.

Example: Using a Trigger to Update a NoPI Table

The following CREATE TRIGGER request creates a statement trigger that does an INSERT SELECT of the values represented by *a* and *b* from the OLD_TABLE transition table (aliased as *myold*) into NoPI table *nopi008_t999* whenever NoPI table *nopi008_ti* is updated.

```
CREATE TRIGGER nopi008_trig4
  AFTER UPDATE ON nopi008_t1
  REFERENCING OLD_TABLE AS myold
  FOR EACH STATEMENT (
    INSERT INTO nopi008_t999(a,b)
      SELECT a,b
      FROM myold;);
```

Example: Invoking an SQL UDF Within a Trigger Definition

This example invokes the SQL UDF *common_value_expression* from the WHEN clause of the definition of the row trigger *trig_insert_after_update*.

```
CREATE TRIGGER trig_insert_after_update
  AFTER UPDATE OF b1 ON t1
  REFERENCING OLD AS OldRow NEW AS NewRow
  FOR EACH ROW
    WHEN (test.common_value_expression(OldRow.a1, NewRow.b1) > .10)
    INSERT INTO t2
      VALUES (NewRow.a1, NewRow.b1, NewRow.c1);
```

The next example invokes the SQL UDF *common_value_expression* in the triggered statement of the row trigger *trig_insert*.

```
CREATE TRIGGER trig_insert
  AFTER INSERT ON t1
  REFERENCING NEW AS n
  FOR EACH ROW (
    INSERT INTO t2
      VALUES (n.a1, test.common_value_expression(n.b2, 1), n.c1); );
```

Related Information

[DROP MACRO](#)

DROP TRIGGER

Drops the definition for the specified trigger from its subject table.

When you drop a trigger, the system:

- Sets an EXCLUSIVE lock on the trigger.
- Frees the disk space used by the dropped trigger and any fallback copy.
- Removes any explicit access privileges on the object.
- Removes the metadata for the dropped object from the data dictionary.

ANSI Compliance

DROP TRIGGER is ANSI SQL:2011-compliant.

Required Privileges

To drop a trigger, you must have DROP TRIGGER privilege on either the subject table of the trigger, or the database containing that table.

DROP TRIGGER Syntax

```
DROP TRIGGER [ database_name. | user_name. ] trigger_name [;]
```

DROP TRIGGER Syntax Elements

database_name

user_name

Name of the containing database or user for the trigger to be dropped.

This specification is required only if the trigger to be dropped is contained in a different database than the current database.

trigger_name

Name of the trigger to drop.

Example: Dropping a Trigger

This request drops a trigger named *parent_tab*.

```
DROP TRIGGER parent_tab;
```

Related Information

- CREATE TRIGGER and REPLACE TRIGGER in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- HELP TRIGGER in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184

HELP TRIGGER

Displays the attributes for a specified trigger.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must either own the specified trigger or table or have at least one privilege on its containing database or, for a trigger, its subject table.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against the specified trigger.

HELP TRIGGER Syntax

```
HELP TRIGGER [ database_name. | user_name. ] { trigger_name | table_name } [;]
```

HELP TRIGGER Syntax Elements

database_name

user_name

Name of the containing database or user for the trigger or table, if something other than the current database or user.

trigger_name

Name of the trigger for which information is being requested.

table_name

Information for all triggers defined on the named table are to be displayed.

HELP TRIGGER Examples

Example: HELP TRIGGER for a Table Key

The following example reports the information for a trigger that updates a table key.

```
HELP TRIGGER UpdateForKey;
*** Help information returned. 1 row.
*** Total elapsed time was 1 second.
      Name          UpdateForKey
      -----
      ActionTime    B
Decimal Order Value 10
      Creation TimeStamp 2003-02-12 15:55:02
      Event         I
      Kind          S
      Enabled       Y
      Comment       ?
      ValidTime Type C
      TransactionTime Type C
```

Example: HELP TRIGGER for a Bitemporal Table

Assume you have created the following row trigger definition for a bitemporal table.

```
CREATE TRIGGER db1.trig_1
  AFTER CURRENT VALIDTIME INSERT ON db1.table_1
  REFERENCING NEW ROW AS NewRow_1
  FOR EACH ROW
  BEGIN ATOMIC
    (ABORT 'aborted' WHERE NewRow_1.c2 > 100;)
  END;
```

A HELP TRIGGER request on this trigger reports the following information for *trig1*.

```
HELP TRIGGER db1.trig1;
      Name trig1
      ActionTime A
Decimal Order Value 32,767
      Creation TimeStamp 2010-12-09 23:11:57
      Event I
      Kind R
      Enabled Y
      Comment ?
```

```
ValidTime Type C
TransactionTime Type C
```

Related Information

- [SHOW object](#)
- *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184
- *Teradata Vantage™ - Temporal Table Support*, B035-1182

RENAME TRIGGER

Renames an existing trigger.

To rename a trigger that is not contained in the same database or user as your current default database setting, you must qualify the old and new names with the name of the containing database or user for the object.

Note:

You cannot change the immediate owner of the renamed object using this statement.

ANSI Compliance

RENAME TRIGGER is ANSI SQL:2011-compliant.

Required Privileges

You must have DROP privileges on the trigger to be renamed and the appropriate CREATE privileges on its containing database or user.

RENAME TRIGGER Syntax

```
RENAME TRIGGER [ database_name_1. | user_name_1. ] old_name
{ TO | AS } [ database_name_2. | user_name_2. ] new_name [ ; ]
```

RENAME TRIGGER Syntax Elements

database_name_1

user_name_1

Optional name of the containing database or user for the trigger to be renamed if other than the current database or user.

You cannot use this statement to change the database or user qualifier for the trigger.

old_name

Existing name for the trigger.

database_name_2***user_name_2***

Optional name of the containing database or user for the renamed trigger if other than the current database or user.

new_name

New name for the trigger.

For information about naming database objects, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

Example: Rename a Trigger

This statement renames the *parent_tab* trigger to *parent_table*.

```
RENAME TRIGGER parent_tab TO parent_table;
```

Related Information

- [CREATE TRIGGER and REPLACE TRIGGER](#)
- [DROP TRIGGER](#)

Comment, Help, and Show Statements

COMMENT (Comment Placing Form)

Creates a user-defined description of a user-defined database object or definition in the data dictionary.

When you add a comment for an object, the new comment is added to the data dictionary, replacing any existing comment.

When you display a comment for an object, the comment text in the data dictionary is returned. Otherwise, null is returned if the object does not have a comment. For information on the comment returning form of the COMMENT statement, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

The following privileges are required to place a COMMENT.

Create or Replace Object Comment	Privilege Required
Object or definition other than a UDT or method	DROP on the object. If the object is a column, then you must have the DROP privilege on the object in which the column is defined.
Comment on a method	UDTMETHOD on the SYSUDTLIB database.
User-defined data type	UDTTYPE on the SYSUDTLIB database.
GLOP	DROP GLOP.

Privileges Granted Automatically

None.

COMMENT Syntax (Comment-Placing Form)

```
COMMENT [ON] { object_kind_1 | object_kind_2 }
  [ database_name. | user_name. ] object_name
  [ [ AS | IS ] 'comment' ] [;]
```

COMMENT Syntax Elements (Comment-Placing Form)

object_kind_1

A mandatory database object kind specification.

The valid specifications for *object_kind_1* are not all database objects. Triggers and views, for example, are definitions of actions rather than database objects.

You must specify the following database object kinds to place a comment for the kind of object they represent:

- COLUMN
- FUNCTION
- GLOP SET
- MACRO
- MAP
- METHOD
- PROCEDURE
- PROFILE
- ROLE
- TRIGGER
- TYPE
- VIEW

object_kind_2

An optional database object kind specification.

For a join index, do not specify an object kind.

You can specify the following database object kinds to retrieve a comment for the kind of object, but these keywords are optional.

- DATABASE
- FILE

If you do not specify FILE for a user-installed file (UIF), you must specify the database name.

- TABLE
- USER

database_name

The containing database for *object_name* if it is not contained by the current database or user.

user_name

The containing user for *object_name* if it is not contained by the current database or user.

comment

A description of the object.

Maximum string length is 255 characters from any supported client character sets.

An existing string can be changed by specifying a new string.

If you do not specify a string, any string previously stored is returned.

object_name

The name of the object to which you want to add a comment. You can add a comment to the following objects:

- Parameter in a macro, SQL procedure, or user-defined function.
- Column in a user base table or view.
- Specific function, macro, file, profile, role, SQL procedure, base table, trigger, or view name contained by a database or user.
- Database or user.
- UDT.

You can add a comment to a particular attribute of a structured UDT by specifying *database_name.udt_name.attribute_name*.

The maximum size for *attribute_name* is 128 Unicode characters.

- Method.

If you specify a method, you must use its specific method name.

- GLOP set.

If no object kind keyword precedes the object name, then the database attempts to deduce the object from the level of qualification in the name. Use the fully qualified name if there is any ambiguity.

Let x.y.z indicate the qualification hierarchy, with x being the highest, such as database or user, and z the lowest, such as parameter or attribute.

The following table shows the hierarchies for *object_name*.

Hierarchy level	Object implied
x	<ul style="list-style-type: none"> • database • user
x.y	<ul style="list-style-type: none"> • base table • error table • GLOP set

Hierarchy level	Object implied
	<ul style="list-style-type: none"> • hash index • join index • macro • profile • role • procedure • trigger • UDF • view within database x or user x.
x.y.z	<ul style="list-style-type: none"> • macro parameter • procedure parameter • structured UDT attribute • table column • UDF parameter • view column within base table, error table, GLOP set, hash index, join index, macro, profile, role, procedure, trigger, UDF, or view y which, in turn, is within database x or user x.

Usage Notes

Maps and Comments

You can specify an existing contiguous or sparse map, including TD_DataDictionaryMap or TD_GlobalMap.

You must be in the same secure zone as the sparse map to place or retrieve the comment for the sparse map.

Japanese Characters in Comments

On Japanese language sites, comments can contain single byte characters, multibyte characters, or both from KanjiEBCDIC, KanjiEUC, or KanjiShift-JIS character sets.

For example, this comment contains single byte and multibyte characters.

```
THIS COMMENT IS USED FOR TABLE TAB 日本語のコメント
```

COMMENT Examples (Comment Placing Form)

Example: Placing a Comment

This statement specifies a description of the name column in the employee table, which is assumed to be in the current database by default because employee is not qualified with a database name. You must specify the keyword COLUMN.

```
COMMENT ON COLUMN employee.name
IS 'Employee name, last name followed by first initial';
```

Example: Commenting on a UDF

Function Definition for Examples

Following is the UDF definition for the examples below:

```
CREATE FUNCTION find_text
( searched_string VARCHAR(500), pattern VARCHAR(500) )
RETURNS CHAR
LANGUAGE C
NO SQL
EXTERNAL
PARAMETER STYLE SQL;
```

Example: Commenting on the Parameter Pattern of a UDF

This statement places a comment on the parameter pattern of the UDF find_text :

```
COMMENT ON COLUMN find_text.pattern
IS 'The pattern matching string';
```

Because the object kind for this request is COLUMN, you must specify the keyword COLUMN for the object kind when you code it.

Example: Commenting on the Definition of an External UDF

This statement places a comment on the definition of the external UDF find_text :

```
COMMENT ON FUNCTION find_text
IS 'Text search function';
```


Because the object kind for this request is FUNCTION, you must specify the keyword FUNCTION for the object kind when you code it.

Example: Commenting on an SQL Function

This statement defines a comment on the SQL function SpecificUDF1. Note that you must specify a specific UDF name.

```
COMMENT ON FUNCTION SpecificUDF1
AS 'This is an SQL user-defined function';
```

Because the object kind for this request is FUNCTION, you must specify the keyword FUNCTION for the object kind when you code it.

Examples: Commenting on a Map

You can add a comment for an existing contiguous or sparse map.

```
COMMENT ON MAP TD_Map2 AS 'All-AMP map following 2016 expansion.';

COMMENT MAP TD_DataDictionaryMap IS 'Map for data dictionary tables.';

COMMENT MAP OneAMP_Map 'One-AMP map based on contiguous map TD_Map5.';
```

Example: Commenting on a Hash Index

Suppose you have a hash index named OrdHIIdx defined on the orders table in the accounting database. You could type the following request to define a comment on the hash index OrdHIIdx :

```
COMMENT accounting.OrdHIIdx AS 'hash index on Orders';
```

Because the object kind for this request is TABLE, representing a hash index, you are not required to specify TABLE for the object kind in the request.

Commenting on UDT and Method Definitions

The following example set shows how to place comments on a UDT or method:

UDT and Method Definitions for Examples

Suppose you define the following UDT and method.

```

CREATE TYPE address AS (
    street VARCHAR(20),
    zip CHARACTER(5) )
NOT FINAL
CONSTRUCTOR METHOD address( a VARCHAR(20),
                           b CHARACTER(5) )

RETURNS address
SPECIFIC address_constructor_1
SELF AS RESULT
LANGUAGE C
PARAMETER STYLE SQL
DETERMINISTIC
NO SQL;
CREATE CONSTRUCTOR METHOD address( VARCHAR(20),
                                  CHARACTER(5) )

RETURNS address
FOR address
EXTERNAL NAME
'SO!C:\structured_lib\addr_cons.obj!F!addr_constructor';

```

Example: Commenting on a UDT Definition

The following request places a comment on the UDT named address:

```

COMMENT ON TYPE address
AS 'Type containing the Street Address and Zipcode' ;

```

Because the object kind for this request is TYPE, you must specify the keyword TYPE for the object kind when you code it.

Example: Commenting on a Method Definition

The following request places a comment on the constructor method named address_constructor_1:

```

COMMENT ON METHOD address_constructor_1
AS 'Constructor for the Address UDT' ;

```

Because the object kind for this request is METHOD, you must specify the keyword METHOD for the object kind when you code it.

Also note that the method name must specify the specific name of the method rather than its calling name. See [CREATE METHOD](#) for further information about the calling name and the specific name for a method.

Related Information

See *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146 for information about the comment returning form of the COMMENT statement.

HELP ONLINE

Displays syntax help for any SQL statement or client utility command.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

None.

HELP ONLINE Syntax

```
HELP { 'HELP' |
      '{ SQL |
        ARCHIVE |
        DUMP |
        FASTEXPORT |
        FASTLOAD |
        MULTILOAD |
        PMPC |
        TPCCONS |
        SPL
      } [ command_name ]'
} [;]
```

HELP ONLINE Syntax Elements

SQL

A help request for the syntax of an SQL statement. If this topic is not followed by a specific statement, a list of commands is displayed for that topic.

ARCHIVE
DUMP
FASTEXPORT
FASTLOAD
MULTILOAD
PMPC
TPCCONS
SPL

A help request for one of the client utilities. These are the topics about which **HELP** is requested.

If these topics are not followed by a specific utility name, a list of commands is displayed for that topic.

command_name

An optional specific SQL statement or client utility command name.

Examples: **HELP ONLINE**

Example	Description
HELP 'SQL';	List of SQL statements and their syntax.
HELP 'MULTILOAD';	List of MultiLoad commands and their syntax.
HELP 'SQL UPDATE';	Syntax for the SQL UPDATE statement.
HELP 'SPL';	List of all SQL procedure DDL and control statements and their syntax.
HELP 'SPL IF';	Syntax, semantics, and usage of the SQL procedure IF control statement.

SHOW *object*

For tables, macros and views, displays the SQL data definition text for the original create text from DBC.TVM.RequestText.

For information on displaying statistics information, see [SHOW STATISTICS](#).

ANSI Compliance

The following statements and associated options are Teradata extensions to the ANSI SQL:2011 standard:

- SHOW AUTHORIZATION
- SHOW CAST
- SHOW CONSTRAINT
- SHOW ERROR TABLE FOR
- SHOW FUNCTION

- SHOW GLOP SET
- SHOW HASH INDEX
- SHOW JOIN INDEX
- SHOW MACRO
- SHOW METHOD
- SHOW PROCEDURE
- SHOW SPECIFIC FUNCTION
- SHOW SPECIFIC METHOD
- SHOW TABLE *error_table_name*
- SHOW TABLE *table_name*
- SHOW TRIGGER
- SHOW TYPE
- SHOW VIEW

Required Privileges

The following privileges are required to run a SHOW request against the specified objects.

To use SHOW ERROR TABLE FOR, SHOW FUNCTION, SHOW HASH INDEX, SHOW JOIN INDEX, SHOW MACRO, SHOW PROCEDURE, SHOW TABLE, or SHOW VIEW, you must have one of the following privileges:

- Any privilege on the user-defined function, hash index, join index, macro, SQL procedure, GLOP set, table, or view, or any privilege on the database containing it.
- At least one privilege on the DBC.TVM table.
- SHOW FUNCTION for an SQL function requires any privilege on the function or its containing database or the SELECT privilege on the DBC.TVM table.

To use SHOW FUNCTION to show the C source code text for an external UDF, you must also have the DROP privilege on the function; otherwise, you can only display the CREATE FUNCTION text.

You do not need the DROP privilege on the function to show the SQL source code text for an SQL UDF.

To use SHOW CAST, SHOW TYPE, SHOW METHOD, or SHOW SPECIFIC METHOD, you must have one of the following privileges.

- At least one privilege on the SYSUDTLIB database.
- The UDTUSAGE privilege on the UDT.
- The SELECT privilege on the DBC.TVM table.

To use SHOW CONSTRAINT, you must have either the CONSTRAINT DEFINITION or CONSTRAINT ASSIGNMENT privilege.

To use SHOW AUTHORIZATION, you must have the DROP AUTHORIZATION privilege on the authorization object.

Use the SHOW privilege to enable a user to perform HELP or SHOW requests only against the specified database object.

SHOW *object* Syntax

```

SHOW {
  [ IN XML ] HASH INDEX [ database_name. | user_name. ] hash_index_name |

  [ IN XML ] JOIN INDEX [ database_name. | user_name. ] join_index_name |

  MACRO [ database_name. | user_name. ] macro_name |

  [ TEMPORARY ] TABLE [ database_name. | user_name. ] table_name |

  ERROR TABLE FOR [ database_name. | user_name. ] data_table_name |

  [ IN XML ] TABLE [ database_name. | user_name. ] error_table_name |

  TRIGGER [ database_name. | user_name. ] trigger_name |

  [ IN XML ] VIEW [ database_name. | user_name. ] view_name |

  PROCEDURE [ database_name. | user_name. ] procedure_name |

  SPECIFIC FUNCTION [ database_name. | user_name. ] specific_function_name |

  FUNCTION [ database_name. | user_name. ] function_name
    [ ( { data_type | UDT_name } [, ...] ) ] |

  SPECIFIC METHOD [SYSUDTLIB.] specific_method_name |

  method |

  CAST [SYSUDTLIB.] UDT_name |

  TYPE [SYSUDTLIB.] { UDT_name | ARRAY_name | VARRAY_name } |

  storage_format SCHEMA [SYSUDTLIB.] schema_name |

  FILE [ uif_database_name. | uif_user_name. ] uif_name |

  CONSTRAINT constraint_name |

  AUTHORIZATION authorization_name |

```

```
GLOP SET GLOP_set_name
}
```

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

  { TIME | TIMESTAMP } [ (fractional_seconds_precision) ] [WITH TIME
ZONE] |

  INTERVAL YEAR [(precision)] [TO MONTH] |

  INTERVAL MONTH [(precision)] |

  INTERVAL DAY [(precision)]
    [TO { HOUR | MINUTE | SECOND [(fractional_seconds_precision)] }] |

  INTERVAL HOUR [(precision)]
    [TO { MINUTE | SECOND [(fractional_seconds_precision)] }] |

  INTERVAL MINUTE [(precision)] [TO SECOND
[ (fractional_seconds_precision) ] ] |

  INTERVAL SECOND [ ( precision [, fractional_seconds_precision ] ) ] |

  PERIOD (DATE) |

  PERIOD ( { TIME | TIMESTAMP } [(precision)] [WITH TIME ZONE] ) |

  REAL |

  DOUBLE PRECISION |

  FLOAT [ (integer) ] |

  NUMBER [ ( { integer | *} [, integer ] ) ] |

  { DECIMAL | NUMERIC } [ ( integer [, integer ] ) ] |

  { CHAR | BYTE | GRAPHIC } [ (integer) ] |

  { VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } (integer) |
```

```

LONG VARCHAR |

LONG VARGRAPHIC |

{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB }
  ( integer [ G | K | M ] ) |

[SYSUDTLIB.] { XML | XMLTYPE } [ ( integer [ G | K | M ] ) ]
  [ INLINE LENGTH integer ] |

[SYSUDTLIB.] JSON [ ( integer [ K | M ] ) ] [ INLINE LENGTH integer ]
  [ CHARACTER SET { UNICODE | LATIN } | STORAGE FORMAT { BSON |
UBJSON } ] |

[SYSUDTLIB.] ST_GEOMETRY [ ( integer [ K | M ] ) ] [ INLINE LENGTH
integer ] |

[SYSUDTLIB.] DATASET [ ( integer [ K | M ] ) ] [ INLINE LENGTH
integer ]
  storage_format |

[SYSUDTLIB.] { UDT_name | MBR | ARRAY_name | VARRAY_name }
}

```

method

```

[ INSTANCE | CONSTRUCTOR ] METHOD [SYSUDTLIB.] method_name
  [ ( { data_type | UDT_name } [, ...] ) ]
  FOR UDT_name

```

storage_format

```

STORAGE FORMAT { Avro | CSV [ CHARACTER SET { UNICODE | LATIN } ] }
  [ WITH SCHEMA [ database. ] schema_name ]

```

SHOW object Syntax Elements**IN XML**

Returns the report in XML format.

This option only applies to SHOW HASH INDEX, SHOW JOIN INDEX, SHOW TABLE, and SHOW VIEW. You can also report on error tables by using the SHOW IN XML TABLE *error_table_name* syntax.

For information on the usage limitations of the output of the IN XML option for join indexes and views, see *join_index_name* or *view_name*, respectively, later in this table.

The XML schema for the output produced by this option is maintained in:

<http://schemas.teradata.com/queryplan/queryplan.xsd>

database_name

The name of the containing database for the object.

For UDTs, methods, and UDFs related to UDTs, this is always SYSUDTLIB.

user_name

The name of the containing user for the object.

For UDTs, methods, and UDFs related to UDTs, this is always SYSUDTLIB.

hash_index_name

The name of the hash index whose most recent SQL create text is to be reported.

join_index_name

The name of the join index whose most recent SQL create text is to be reported.

If the join index is defined on a temporal table, the SQL text returned by SHOW JOIN INDEX includes the temporal qualifier that was associated with the embedded SELECT request at the time the join index was created, whether it was explicitly specified or implicitly derived from the default session temporal qualifier.

If the join index is system-defined, the SQL text returned by SHOW JOIN INDEX displays the non-reserved keyword SYSTEM_DEFINED between the words CREATE and JOIN.

If you submit this SQL text as a CREATE JOIN INDEX request, the database returns an error to the requestor because SYSTEM_DEFINED is not valid SQL text.

A SHOW JOIN INDEX IN XML request does not report all of the definition constructs for join indexes, so it is not possible to decompose and reconstruct their definitions from their reported XML format definitions.

However, the XML text for join index definitions is helpful because it includes the following useful information:

- The names and data types of the columns in the join index definition.
- A list of all of the referenced database objects in the join index definition.

For further information, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

For a column-partitioned join index, the SQL text returned by SHOW JOIN INDEX includes a PARTITION BY clause with a COLUMN clause. Grouping, if any, is included in the COLUMN clause, not in the select expression list.

See *table_name* later in this table for a list of exceptions to the rules for what the database reports for a SHOW JOIN INDEX request. These rules apply equally to SHOW JOIN INDEX and SHOW TABLE requests.

macro_name

The name of the macro whose most recent SQL create text is to be reported.

There is an upper limit of 12,500 characters that SHOW MACRO can display.

table_name

Display the most recent SQL create text. See [SHOW TABLE](#).

data_table_name

The name of the data table for which the error table whose most recent SQL create text is to be reported.

This syntax is useful if you did not define an explicit name for the error table and you do not know the system-assigned default name assigned to it.

error_table_name

The name of the error table whose most recent SQL create text is to be reported.

trigger_name

The name of the trigger whose most recent SQL create text is to be reported. There is an upper limit of 12,500 characters that SHOW TRIGGER can display.

view_name

The name of the view whose most recent SQL create text is to be reported. There is an upper limit of 12,500 characters that SHOW VIEW can display.

A SHOW VIEW IN XML request does not report all of the definition constructs for views, so it is not possible to decompose and reconstruct their definitions from their reported XML format definitions.

Despite this, the XML text for view definitions is helpful because it includes the following useful information.

- The names and data types of the columns in the view definition.
- A list of all of the referenced database objects in the view definition.

For further information, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

procedure_name

The name of the SQL procedure whose most recent SQL create text is to be reported.

SHOW PROCEDURE displays procedure text as it was sent to the database. You must type line breaks in the procedure using the Enter or Return key to properly display the text.

specific_function_name

The specific function name for the UDF whose most recent SQL create text is to be reported.

function_name

The function name for the UDF whose most recent SQL create text is to be reported.

data_type

UDT_name

The data type parameters, including UDTs, that uniquely identify an overloaded function name. For a list of data types, see [Data Types Syntax](#).

specific_method_name

The specific name for the method whose most recent SQL create text is to be reported.

method_name

The name of the method whose most recent SQL create text is to be reported.

UDT_name

The name of the UDT for which the most recent SQL create text, including that for its CREATE ORDERING and CREATE TRANSFORM statements, are to be reported. This specification applies to both the SHOW CAST and the SHOW TYPE statements.

Similarly, if the only cast or ordering or transform functionality for a UDT is system-generated, the system does not report that DDL because it was not created by SQL CREATE requests, so there is no create text to return.

ARRAY_name

VARRAY_name

The name of the ARRAY or VARRAY type for which the most recent SQL create text is to be reported. The SQL create text is reported in standard Teradata syntax using the ARRAY keyword.

The ordering and transform functionality for the ARRAY type are not displayed because they are system-generated, so there is no create text to return.

storage_format

Storage format of the DATASET type for which to show the schema.

SYSUDTLIB

Name of the target database containing the schema.

schema_name

Name of schema.

uif_name

Name of user-installed file.

Display the text associated with the installed file and display the DDL used to install the file.

Note:

Binary user-installed files (UIF) are not displayed.

uif_database_name

Name of database containing the user-installed file (UIF).

uif_user_name

Name of user containing the user-installed file (UIF).

constraint_name

The name of the constraint for which the most recent SQL create text is to be reported.

authorization_name

The name used to create the authorization object with a CREATE AUTHORIZATION request.

GLOP_set_name

The name of the GLOP set whose definition is to be reported.

The request reports the DDL for the CREATE GLOP SET request that created *GLOP_set_name*.

SHOW *object* Examples

Example: SHOW HASH INDEX

This example uses the following CREATE HASH INDEX statement to create ord_cust_hidx:

```
CREATE HASH INDEX ord_cust_hidx
(o_date) ON orders
BY (o_date)

ORDER BY VALUES;
```

The SHOW HASH INDEX report differs from the text of the CREATE HASH INDEX DDL as follows:

- All names are qualified
- Fallback protection and checksum options display
- Map assignment

When you do not specify a BY or ORDER clause in the CREATE HASH INDEX DDL, the default values for those clauses are not reported.

```
SHOW HASH INDEX ord_cust_hidx;
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
-----
CREATE HASH INDEX YourDB.ord_cust_hidx ,NO FALLBACK ,CHECKSUM = DEFAULT,
MAP = TD_MAP1
(o_date )
ON YourDB.orders
BY (o_date )
ORDER BY VALUES
;
```

Example: SHOW HASH INDEX in XML Format

Assume you alter the hash index to use a sparse map:

```
ALTER HASH INDEX ord_cust_hidx, MAP=SmallTableMap;
```

This example returns the create text for hash index ord_cust_hidx in XML format:

```
SHOW IN XML HASH INDEX ord_cust_hidx;
```

The XML document generated reports all of the information required to recreate the hash index:

```
<?xml version="1.0" encoding="UTF-8"
standalone="no" ?><TeradataDBObjectSet version="1.0" xmlns="http://
schemas.teradata.com/dbobject" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://schemas.teradata.com/dbobject http://
schemas.teradata.com/dbobject/DBObject.xsd">
<HashIndex baseDBName="YourDB" baseTableName="orders" checkSumLevel="Default"
colocateName="YourDB_ord_cust_hidx" dbName="YourDB" fallback="false"
map="SmallTableMap" map_kind="sparse" name="ord_cust_hidx"
objId="0:3655" objVer="2"><IndexColumnList><Column name="o_date" order="1"/></
IndexColumnList><DistributeColumnList><Column name="o_date" order="1"/></
DistributeColumnList><OrderBy mode="Value"/>
<SQLText><![CDATA[CREATE HASH INDEX YourDB.ord_cust_hidx ,NO FALLBACK
,CHECKSUM = DEFAULT,
MAP = SmallTableMap COLOCATE USING YourDB_ord_cust_hidx
(o_date )
ON YourDB.orders
BY (o_date )
ORDER BY VALUES
;]]></SQLText></HashIndex><Environment><Server dbRelease="16.10"
dbVersion="16.10" hostName="localhost"/><User userId="00000704"
userName="User"/><Session charset="UTF8" dateTime="2017-03-24T17:22:48"/></
Environment></TeradataDBObjectSet>
```

Example: SHOW JOIN INDEX

This example illustrates a SHOW JOIN INDEX statement for the join index ord_cust_idx.

```
SHOW JOIN INDEX ord_cust_idx;
```

The statement reports the DDL for the join index:

```
CREATE JOIN INDEX YourDB.ord_cust_idx ,NO FALLBACK
,CHECKSUM = DEFAULT, MAP = SmallTableMap COLOCATE USING YourDB_ord_cust_idx AS
SELECT (YourDB.orders.o_custkey ,YourDB.customer.c_name )
,(YourDB.orders.o_status ,
YourDB.orders.o_date ,YourDB.orders.o_comment )
FROM YourDB.orders ,YourDB.customer
```

```
WHERE YourDB.orders.o_custkey = YourDB.customer.c_custkey
PRIMARY INDEX ( o_custkey )
```

Example: SHOW JOIN INDEX in XML Format

This example provides an example of the output for the following statement:

```
SHOW IN XML JOIN INDEX ord_cust_idx;
```

The XML document generated by SHOW IN XML JOIN INDEX does not contain the parsed query for the join index, only the column definitions of the index and the database objects it references. As a result, this XML document cannot be used to recreate the join index. However, it can be used to create a table having the same column definition of the original join index. The statement returns the following report:

```
<?xml version="1.0" encoding="UTF-8"
standalone="no" ?><TeradataDBObjectSet version="1.0" xmlns="http://
schemas.teradata.com/dbobject" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://schemas.teradata.com/dbobject http://
schemas.teradata.com/dbobject/DBObject.xsd"><JoinIndex checkSumLevel="Default"
colocateName="YourDB_ord_cust_idx" dbName="YourDB" fallback="false"
map="SmallTableMap" map_kind="sparse" name="ord_cust_idx"
objId="0:3652" objVer="1">
<RepeatComlumnList><Column format="X(1)" name="o_status" nullable="true"
order="1"><DataType><Char casespecific="true" charset="LATIN" length="1"
uppercase="false" varying="false"/></DataType></Column>
<Column format="yyyy-mm-dd" name="o_date" nullable="true"
order="2"><DataType><Date/></DataType></Column>
<Column format="X(79)" name="o_comment" nullable="true"
order="3"><DataType><Char casespecific="true" charset="LATIN" length="79"
uppercase="false" varying="true"/></DataType></Column></RepeatComlumnList>
<FixedColumn
List><Column format="-(10)9" name="o_custkey" nullable="true"
order="1"><DataType><Integer/></DataType></Column><Column format="X(26)"
name="c_name" nullable="false" order="2"><DataType><Char casespecific="true"
charset="LATIN" length="26" uppercase="false" varying="false"/></DataType></
Column></FixedColumnList>
<RefList><Ref dbName="YourDB" name="orders" type="Table"/><Ref dbName="YourDB"
name="customer" type="Table"/></RefList>
<SQLText><![CDATA[CREATE JOIN INDEX YourDB.ord_cust_idx ,NO FALLBACK
,CHECKSUM = DEFAULT, MAP = SmallTableMap COLOCATE USING YourDB_ord_cust_idx AS
SELECT (YourDB.orders.o_custkey ,YourDB.customer.c_name )
,(YourDB.orders.o_status ,
YourDB.orders.o_date ,YourDB.orders.o_comment )
```

```

FROM YourDB.orders ,YourDB.customer
WHERE YourDB.orders.o_custkey = YourDB.customer.c_custkey
PRIMARY INDEX ( o_custkey )
INDEX ( o_custkey ,c_name )
INDEX ( o_status ,o_date ,o_comment );]]></SQLText></
JoinIndex><Environment><Server dbRelease="16.10" dbVersion="16.10"
hostName="localhost"/><User userId="00000704" userName="YourDB"/
><Session charset="UTF8" dateTime="2017-03-24T16:54:56"/></
Environment></TeradataDBObjectSet>

```

Example: SHOW MACRO

The following request displays the most recently executed definition of the new_emp macro.

```
SHOW MACRO new_emp;
```

The database system reports the following macro definition.

```

REPLACE MACRO new_emp(
  name  CHARACTER(20) NOT NULL,
  street CHARACTER(30),
  city  CHARACTER(20),
  number INTEGER,
  dept  SMALLINT DEFAULT 999)
AS (INSERT INTO employee (name, street, city,
                          emp_no, dept_no)
    VALUES (:name, :street, :city, :number, :dept);
  UPDATE department
    SET emp_count = emp_count + 1
  WHERE dept_no = :dept);

```

Example: SHOW VIEW

This example begins with the creation of the view, then progresses to the SHOW VIEW request and finally presents the result of the report request.

```

CREATE VIEW staff_info
(number, name, position, department, sex, dob) AS
SELECT employee.empno, name, jobtitle, deptno, sex, dob
FROM employee

```



```
WHERE jobtitle NOT IN ('Vice Pres', 'Manager')
WITH CHECK OPTION;
```

When you submit the following SHOW VIEW request, the database returns the following view definition.

```
SHOW VIEW staff_info;
CREATE VIEW staff_info
  (number, name, position, department, sex, dob) AS
SELECT employee.empno, name, jobtitle, deptno, sex, dob
FROM employee
WHERE jobtitle NOT IN ('Vice Pres', 'Manager')
WITH CHECK OPTION;
```

Example: SHOW VIEW in XML Format

This example returns the CREATE VIEW text for table *user_name.v1* in XML format. Unlike a SHOW IN XML TABLE request, the XML document generated by a SHOW IN XML VIEW request does not report the parsed query for the view, and only contains the column definitions of the view and the database objects it references. As a result, this XML document cannot be used to recreate the view. However, it can be used to create a table having the same column definition of the original view.

The table that *v1* is defined on looks like this.

```
CREATE SET TABLE TESTDB.vt1, NO FALLBACK, NO BEFORE JOURNAL,
  NO AFTER JOURNAL, CHECKSUM = DEFAULT, DEFAULT MERGEBLOCKRATIO (
  a1 INTEGER NOT NULL,
  b1 INTEGER,
  c1 CHARACTER(20) CHARACTER SET UNICODE NOT CASESPECIFIC,
  d1 DECIMAL(15,2));
```

The create text for *v1* looks like this.

```
CREATE VIEW v1 AS
  SELECT a1, b1, c1
  FROM vt1
  WHERE d1 > 10000;
```

When you submit the following SHOW VIEW IN XML request, the database returns the report that follows the request.

```
SHOW VIEW v1 IN XML;
<?xml version="1.0" encoding="utf-8"?>
<TeradataDBObjectSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

xsi:schemaLocation="http://schemas.teradata.com/TeradataDBObject
TeradataDBObject.xsd"
xmlns="http://schemas.teradata.com/TeradataDBObject" version="1.0">
<View name="v1" dbName="testdb">
<ColumnList>
<Column name="a1" order="1">
<DataType><Integer/></DataType>
<Constraint nullable="false"/>
</Column>
<Column name="b1" order="2">
<DataType><Integer/></DataType>
</Column>
<Column name="c1" order="3">
<DataType>
<Char length="20" varying="false" uppercase="false" casespecific="false"
charset="unicode"/>
</DataType>
</Column>
</ColumnList>
<RefList>
<Ref name="vt1" dbName="testdb" type="table"/>
</RefList>
<SQLText>
<![CDATA[
CREATE VIEW V1 AS
SELECT A1, B1, C1
FROM VT1
WHERE D1 > 10000;
</SQLText>
</View>
<Environment>
<Server dbRelease="14.10.00.00" dbVersion="14.10.00.06" hostName="testhost"/>
<User userId="0" userName="dbc"/>
<Session charset="utf8" dateTime="2012-01-01T14:00:00-08:00"/>
</Environment>
</TeradataDBObjectSet>

```

Example: SHOW PROCEDURE

Following is an example shows the DDL for a procedure named spSample1:

```

SHOW PROCEDURE spSample1;
*** Text of DDL statement returned.

```

```

*** Total elapsed time was 1 second.
-----
CREATE PROCEDURE spSample1 (IN ip INTEGER,
                           OUT op INTEGER)
BEGIN DECLARE var_1 INTEGER;
SELECT col1 INTO var_1
FROM tab_1
WHERE col_2 = ip;
SET op = var1 * 10;
END;

```

Example: Showing a Java External Stored Procedure with an XML Data Type

This example shows the DDL for a Java external stored procedure with an XML data type as an input parameter:

```

SHOW PROCEDURE get_XML;
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
-----
REPLACE PROCEDURE get_XML(IN X1 XML, INOUT A1 INTEGER)
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
  EXTERNAL NAME
  'UDF_JAR:UserDefinedFunctions.get_XML(java.sql.SQLXML, int[])';

```

Example: SHOW FUNCTION (External Form)

This example reports the current DDL and function body for the specific external function name addnum.

```

SHOW FUNCTION addnum;
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
-----
REPLACE FUNCTION rgs.add_num (p1 INTEGER, p2 FLOAT)
  RETURNS FLOAT
  SPECIFIC add_num
  LANGUAGE C
  NO SQL
  PARAMETER STYLE SQL

```

```

NOT DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'cs!first1!udftest/first1.c!F!first1'
*** Text of DDL statement returned.
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
/* add integer and float */
void first1(INTEGER  *a,
            FLOAT    *b,
            FLOAT    *result,
            INT       *indc_a,
            INT       *indc_b,
            INT       *indc_result,
            CHAR      sqlstate[6],
            SQL_TEXT  extname[129],
            SQL_TEXT  specific_name[129],
            SQL_TEXT  error_message[257])
{
    if (*indc_a == -1 || *indc_b == -1)
    {
        *indc_result = -1;
        return;
    }
    *result = *a + *b;
    *indc_result = 0;
}

```

Example: SHOW FUNCTION (SQL Form)

SHOW FUNCTION (SQL Form) requests display the DDL create text for the SQL UDF. The output displayed by a SHOW FUNCTION request returns the actual create text that was used to create the SQL UDF and does not return the default values for some of the optional clauses that were not specified explicitly during the creation of the SQL UDF.

```

SHOW SPECIFIC FUNCTION udf_1;
CREATE FUNCTION udf_1 (a INTEGER,
                     b INTEGER)

RETURNS INTEGER
CONTAINS SQL
RETURN a + b;

```

Note that the output of this example does not display the default values for some options like the LANGUAGE clause, DETERMINISTIC clause, the null call clause, and so on because the creator of udf1

did not type these clauses explicitly when she created the SQL UDF, so the SHOW FUNCTION request does not return the default values for those clauses.

In the following example, the SHOW FUNCTION request does return the default values for the LANGUAGE clause, the DETERMINISTIC clause, and the null call clause because the creator typed those clauses explicitly when he created udf_2.

```
SHOW FUNCTION udf2 (INTEGER, INTEGER);
CREATE FUNCTION udf2 (a INTEGER, b INTEGER)
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
CALLED ON NULL INPUT
RETURN a - b;
```

Example: SHOW METHOD

The following examples show valid cases of SHOW METHOD.

- This example reports the SQL create text for a method that has the specific method name polygon_mbr.

```
SHOW SPECIFIC METHOD SYSUDTLIB.polygon_mbr;
```

- This example reports the SQL create text for a method that has the name polygon_mbr. Note that the method has no parameters, so no parameter data type list is specified.

```
SHOW METHOD SYSUDTLIB.area() FOR circle;
```

- This example reports the SQL create text for a method that has the name in_state and is associated with the UDT named circle. To be distinguished from another method definition on circle named in_state, the specification includes the parameter data type list for the method.

```
SHOW METHOD in_state(CHAR(2)) FOR address;
```

Example: Show User-Installed File (UIF)

This is example of SHOW FILE for a user installed file (UIF).

```
SHOW FILE mapper;
```

```
*** Text of DDL statement returned.
```

```
-----
```

```
--
CALL SYSUIF.INSTALL_FILE('mapper','mapper.py','cz!mapper.py!/tmp/mapper.py')

*** Text of DDL statement returned.
-----
--
#!/usr/bin/python

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

Example: SHOW CAST

The following example shows an instance of the SHOW CAST report where two different casts have been defined for the distinct UDT named *distinct_udt_1*.

```
SHOW CAST sysudtlib.DistinctUdt1;
*** Text of DDL statement returned.
*** Total elapsed time was 2 seconds.
-----
CREATE CAST (distinct_udt_1 AS VARBYTE(9012))
  WITH SPECIFIC FUNCTION d2f;
CREATE CAST (INTEGER AS distinct_udt_1)
  WITH SPECIFIC FUNCTION i2d;
```

Example: SHOW TYPE

The following examples show two instances of the SHOW TYPE report for a UDT.

Note:

To report cast information for a UDT, you must use the SHOW CAST statement.

```

SHOW TYPE UdtInt;
*** Text of DDL statement returned.
*** Total elapsed time was 2 seconds.
-----
CREATE TYPE SYSUDTLIB.udt_int AS INTEGER FINAL
INSTANCE METHOD intordering ( )
RETURNS INTEGER
SPECIFIC udt_int_int_ordering_24a8_r
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
RETURNS NULL ON NULL INPUT,
INSTANCE METHOD int_fromsql ( )
RETURNS INTEGER
SPECIFIC udt_int_int_fromsql_24a9_r
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
RETURNS NULL ON NULL INPUT;
SHOW TYPE udtint;
*** Text of DDL statement returned.
*** Total elapsed time was 5 seconds.
-----
CREATE TYPE SYSUDTLIB.udt_int AS INTEGER FINAL
INSTANCE METHOD int_ordering ( )
RETURNS INTEGER
SPECIFIC udt_int_int_ordering_24a8_r
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
RETURNS NULL ON NULL INPUT,
INSTANCE METHOD int_fromsql ( )
RETURNS INTEGER
SPECIFIC udt_int_int_fromsql_24a9_r
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
RETURNS NULL ON NULL INPUT;

```

Example: SHOW TYPE for ARRAY Types

Create the following ARRAY types.

```
CREATE TYPE myarray_1 AS
VARRAY(5) OF CHARACTER(10);
CREATE TYPE myarray_2 AS
CHARACTER(10) ARRAY[5];
```

A SHOW TYPE request for the ARRAY type *myarray_1* returns the following information. Note that the SQL text returned for CREATE TYPE *myarray_1* is reported using ARRAY format rather than the VARRAY format that was used to create the type. Because of this, the CREATE TYPE SQL text returned for *myarray_1* is identical to that returned for *myarray_2*.

```
SHOW TYPE myarray_1;
CREATE TYPE myarray_1 AS CHARACTER(10) ARRAY[5];
```

A SHOW TYPE request for the ARRAY type *myarray_2* returns the following information.

```
SHOW TYPE myarray_2;
CREATE TYPE myarray_2 AS CHARACTER(10) ARRAY[5];
```

Example: SHOW Avro SCHEMA

This example shows the schema for the CREATE *storage_format* SCHEMA example:

```
SHOW Avro SCHEMA chemDatasetSchema;
>
CREATE Avro SCHEMA chemDatasetSchema AS
'{
  "namespace": "example.avro",
  "type": "record",
  "name": "User",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "favorite_number", "type": ["int", "null"]},
    {"name": "favorite_color", "type": ["string", "null"]}
  ]
}';
```


Example: SHOW CONSTRAINT

This example shows a valid case of SHOW CONSTRAINT for a row-level security constraint.

The constraint is defined using the following row-level security constraint UDF.

```
CREATE FUNCTION SYSLIB.add_func (
  CURRENT_SESSION SMALLINT)
  RETURNS SMALLINT
  LANGUAGE C
  NO SQL
  PARAMETER STYLE TD_GENERAL
  EXTERNAL NAME 'CS!add_func!/home/xiaoy/test/showtab/constraint/
    add_func.c';
```

The constraint is defined as follows.

```
CREATE CONSTRAINT c1 SMALLINT, NOT NULL,
VALUES (a:1),
INSERT SYSLIB.add_func;
```

A SHOW CONSTRAINT request made for row-level security constraint *c1* returns the DDL for its definition as follows.

```
SHOW CONSTRAINT c1;
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
-----
CREATE CONSTRAINT c1 SMALLINT, NOT NULL,
VALUES (a:1),
INSERT SYSLIB.add_func;
```

SHOW request

Displays the DDL for all database objects referenced by a specified DML request.

You can return the report in XML format.

For information on displaying query logging information, see [SHOW QUERY LOGGING](#).

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Required Privileges

You must have the following privileges to execute a SHOW or SHOW IN XML request:

- The privileges necessary to perform the specified DML request.
- Any privilege on the database objects to show.

The SHOW privilege enables you to perform HELP or SHOW requests for the specified DML request.

SHOW *request* Syntax

```
SHOW [IN XML] [QUALIFIED] DML_request [;]
```

SHOW request Syntax Elements

IN XML

Return the report in XML format.

The XML schema for the output produced by this option is maintained in:

<http://schemas.teradata.com/queryplan/queryplan.xsd>

QUALIFIED

Any view DDL returned is to be qualified by the appropriate database and table names. This qualified text is taken from the DBC.TVM.CreateText column.

All qualified objects in the report are enclosed within QUOTATION MARK characters.

If you do not specify QUALIFIED, then any returned view DDL is taken from the DBC.TVM.RequestText column and is not qualified by the names of the appropriate database and table names.

DML_request

An SQL DML request for which the full text of the DDL for all the database objects it references is to be returned.

System-defined join index definitions on a temporal table are not returned by a SHOW DML request. The base table definition text seen in the SHOW TABLE output on which the system-defined join index is created returns the temporal constraints that are implemented as system-defined single-table join indexes, and it is only when this CREATE TABLE SQL text is issued to the database that the system-defined join indexes, if any, are created.

Examples

Example: SHOW For a SELECT Request

The following example reports the DDL definitions supporting a simple SELECT request involving one view and one table. Both objects were created in database DAFDB, so qualification is unnecessary.

```
SHOW
SELECT *
FROM View_1, Table_2
WHERE View1.column_1=Table2.column_3;
```

The report looks like this.

```
*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
```

```
-----
CREATE SET TABLE DAFDB.Table_2
  NO FALLBACK,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
  (column_1 INTEGER,
   column_2 INTEGER)
  PRIMARY INDEX(column_1);
CREATE SET TABLE DAFDB.Table_1
  NO FALLBACK,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
  (column_1 INTEGER)
  PRIMARY INDEX (column_1);
CREATE VIEW View_1 AS
  SELECT column_1 FROM Table_1;
```

Example: SHOW For a SELECT Request That Accesses Multiple Views

You create a view using two other views, each having the same name, but created in different databases.

The database names are US and International. The session is in Teradata mode.

Here is the DDL used to create the objects used for this example.

```

DATABASE us;
CREATE TABLE table_1 (
  eno INTEGER);
CREATE VIEW view_1 AS
  SELECT *
  FROM table_1;
DATABASE international;
CREATE TABLE table_1 (
  zno INTEGER);
CREATE VIEW view_1 AS
  SELECT *
  FROM table_1;
CREATE VIEW view_3 AS
  SELECT *
  FROM international.view_1, us.view_1;

```

If you perform a SHOW request on a query that selects data from view_3 , the report it returns will be ambiguous with respect to the two view_1 elements of view_3 unless you specify the QUALIFIED option.

For example, suppose you perform SHOW on the following query:

```

SHOW SELECT *
  FROM view_3;

```

The report looks like this:

```

*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
-----
CREATE SET TABLE international.table_1, NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL
(zno INTEGER)
PRIMARY INDEX (a);
*** Text of DDL statement returned.
-----
CREATE SET TABLE us.table_1, NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL
(enno INTEGER)
PRIMARY INDEX (enno);
*** Text of DDL statement returned.
-----
CREATE VIEW view_1 AS SELECT * FROM table_1;

```

```
*** Text of DDL statement returned.
```

```
-----  
CREATE VIEW view_1 AS SELECT * FROM table_1;
```

```
*** Text of DDL statement returned.
```

```
-----  
  
CREATE VIEW view_3 AS SELECT * FROM international.view_1,us.view_1;
```

Example: SHOW For a SELECT Request Using the QUALIFIED Option

Notice that none of the views in [Example: SHOW For a SELECT Request That Accesses Multiple Views](#) is qualified by the name of its containing database.

Use the QUALIFIED option to produce a report that resolves the ambiguous dependencies.

Consider the same scenario used in [Example: SHOW For a SELECT Request That Accesses Multiple Views](#), but this time specify that the report must be qualified.

```
SHOW QUALIFIED SELECT *  
FROM view_3;
```

The report looks like this.

```
*** Text of DDL statement returned.
```

```
*** Total elapsed time was 1 second.
```

```
-----  
CREATE SET TABLE international.table_1, NO FALLBACK,  
NO BEFORE JOURNAL,  
NO AFTER JOURNAL  
(zno INTEGER)  
PRIMARY INDEX (a);
```

```
*** Text of DDL statement returned.
```

```
-----  
CREATE SET TABLE us.table_1, NO FALLBACK,  
NO BEFORE JOURNAL,  
NO AFTER JOURNAL  
(eno INTEGER)  
PRIMARY INDEX (eno);
```

```
*** Text of DDL statement returned.
```

```
-----  
CREATE VIEW "international"."view_1" AS SELECT *  
FROM      "international"."table_1";
```

```
*** Text of DDL statement returned.
```

```
-----
CREATE VIEW "us"."view_1" AS SELECT * FROM "us"."table_1";
*** Text of DDL statement returned.
```

```
-----

CREATE VIEW "international"."view_3" AS SELECT * FROM
"international"."view_1"."zno", "us"."view_1"."eno";
```

Each of the view definitions in this report is fully qualified, removing the ambiguity in [Example: SHOW For a SELECT Request That Accesses Multiple Views](#).

Example: SHOW For a SELECT Request on a Table With Multiple Referential Integrity Constraints

Suppose you have three base tables, two of which are parents to the third by means of referential integrity relationships.

In the following set of DDL table definitions, tables t1 and t3 are both parents to table t2.

```
CREATE SET TABLE t1 (
  a1 INTEGER,
  b1 INTEGER,
  FOREIGN KEY (b1) REFERENCES WITH CHECK OPTION t3(a3))
  UNIQUE PRIMARY INDEX (a1);
CREATE SET TABLE t3 (
  a3 INTEGER,
  b3 INTEGER,
  FOREIGN KEY (b3) REFERENCES WITH CHECK OPTION t1(a1))
  UNIQUE PRIMARY INDEX (a3);
CREATE SET TABLE t2 (
  a2 INTEGER,
  b2 INTEGER,
  b3 INTEGER,
  FOREIGN KEY (a2) REFERENCES WITH CHECK OPTION t1 (a1),
  FOREIGN KEY (b3) REFERENCES WITH CHECK OPTION t3 (a3))
  PRIMARY INDEX (a2);
```

You decide to run a SHOW report on the following query. Notice that even though the query only touches table t2 directly, the report also displays the DDL for the two parent tables of t2. t1 and t3.

```
SHOW SELECT *
FROM t2;
```

```

*** Text of DDL statement returned.
*** Total elapsed time was 1 second.
-----
CREATE SET TABLE TEST.T2 ,NO FALLBACK ,
      NO BEFORE JOURNAL,
      NO AFTER JOURNAL
      (a2 INTEGER,
       b2 INTEGER,
       b3 INTEGER,
FOREIGN KEY ( a2 ) REFERENCES WITH CHECK OPTION TEST.T1 ( a1 ),
FOREIGN KEY ( b3 ) REFERENCES WITH CHECK OPTION TEST.T3 ( a3 ))
PRIMARY INDEX ( a2 );
*** Text of DDL statement returned.
-----
CREATE SET TABLE TEST.T1, NO FALLBACK,
      NO BEFORE JOURNAL,
      NO AFTER JOURNAL
      (a1 INTEGER,
       b1 INTEGER
FOREIGN KEY (b1) REFERENCES WITH CHECK OPTION t3(a3))
UNIQUE PRIMARY INDEX (a1);
*** Text of DDL statement returned.
-----
CREATE SET TABLE TEST.T3, NO FALLBACK,
      NO BEFORE JOURNAL,
      NO AFTER JOURNAL
      (a3 INTEGER,
       b3 INTEGER
FOREIGN KEY (b3) REFERENCES WITH CHECK OPTION t1(a1))
UNIQUE PRIMARY INDEX (a3);

```

Example: SHOW for an Updatable DATE PPI Table

Suppose you create the following insurance customer table that is partitioned into historical (expired) policies and current policies.

```

CREATE TABLE customer (
  cust_name          CHARACTER(8),
  policy_number      INTEGER,
  policy_expiration_date DATE FORMAT 'YYYY/MM/DD')
PRIMARY INDEX (cust_name, policy_number)
PARTITION BY CASE_N(policy_expiration_date>=CURRENT_DATE, NO CASE);

```

You then submit a SHOW DML request for the following SELECT request against customer.

```
SHOW SELECT *
FROM customer;
```

The request returns the following DDL.

```
CREATE SET TABLE MOVEDATE.customer ,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT
    (
        cust_name CHARACTER(8) CHARACTER SET LATIN NOT CASESPECIFIC,
        policy_number INTEGER,
        policy_expiration_date DATE FORMAT 'YYYY/MM/DD')
PRIMARY INDEX ( cust_name ,policy_number )
PARTITION BY CASE_N(policy_expiration_date >= DATE, NO CASE);
```

The output of this report displays the original CURRENT_DATE expression as a DATE expression.

You then submit a qualified SHOW DML request for the following SELECT request against customer.

```
SHOW QUALIFIED SELECT *
FROM customer;
```

The request returns the following DDL.

```
CREATE SET TABLE MOVEDATE.customer ,NO FALLBACK,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT
    (
        cust_name CHAR(8) CHARACTER SET LATIN NOT CASESPECIFIC,
        policy_number INTEGER,
        policy_expiration_date DATE FORMAT 'YYYY/MM/DD')
PRIMARY INDEX ( cust_name ,policy_number )
PARTITION BY CASE_N(
    policy_expiration_date >= DATE '2007-04-17', NO CASE);
```

The output of this report displays the partitioning expression with the original CURRENT_DATE expression replaced by the resolved date.

Example: SHOW for an SQL Function

```
SHOW SELECT *
      FROM t1
      WHERE udf1(1,2) = 3;
```

Example: SHOW in XML Format

The SHOW IN XML statement returns the definitions of database objects referenced either directly or indirectly by a DML request in XML format. All database object names are qualified.

The XML result includes the definitions for all of the parent tables of the tables referenced in the DML request as well as the definitions for all of the base tables and base views for the views referenced in the DML request.

The definition for table *t1* is as follows.

```
CREATE SET TABLE user.t1, NO FALLBACK, NO BEFORE JOURNAL,
                      NO AFTER JOURNAL, CHECKSUM = DEFAULT,
                      DEFAULT MERGEBLOCKRATIO (
a1 INTEGER NOT NULL,
b1 CHAR(20) CHARACTER SET UNICODE NOT CASESPECIFIC,
c1 DATE FORMAT 'YY/MM/DD',
PRIMARY KEY (a1));
```

The definition for table *t2* is as follows.

```
CREATE SET TABLE user.t2, NO FALLBACK, NO BEFORE JOURNAL,
                      NO AFTER JOURNAL, CHECKSUM = DEFAULT,
                      DEFAULT MERGEBLOCKRATIO (
a2 INTEGER NOT NULL,
b2 DECIMAL(6,4) NOT NULL,
c2 VARCHAR(20) CHARACTER SET UNICODE NOT CASESPECIFIC)
UNIQUE PRIMARY INDEX (a2);
```

The request whose database object definitions are to be displayed is the following.

```
SHOW SELECT a1, b1, b2 FROM t1, t2 WHERE a1=a2;
XML Document for SHOW IN XML SELECT a1, b1, b2 FROM t1, t2 WHERE a1=a2:
1.      <?xml version="1.0" encoding="UTF-8"?>
<TeradataDBObjectSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://schemas.teradata.com/TeradataDBObjectSet"
```

```

TeradataDBObject.xsd"
xmlns="http://schemas.teradata.com/TeradataDBObject" version="1.0">
<Table objId="0:331" objVer="1" kind="set" baseClass="Table" name="t1"
dbName="testdb" fallback="false" beforeJournal="no" afterJournal="no"
checksumLevel="default" mergeBlockRatio="default">
<ColumnList>
<Column name="a1" order="1">
<DataType><Integer/></DataType>
<Constraint nullable="false"/>
</Column>
<Column name="b1" order="2">
<DataType><Char length="20" varying="false" uppercase="false"
casespecific="false" charset="unicode"/></DataType>
</Column>
<Column name="c1" order="3" format="YY/MM/DD">
<DataType><Date/></DataType>
</Column>
</ColumnList>
<TableConstraint>
<PrimaryKey>
<ColumnList>
<Column name="a1" order="1"/>
</ColumnList>
</PrimaryKey>
</TableConstraint>
<SQLText>
<![CDATA[
CREATE SET TABLE user.t1 ,NO FALLBACK ,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT,
DEFAULT MERGEBLOCKRATIO
(
a1 INTEGER NOT NULL,
b1 CHAR(20) CHARACTER SET UNICODE NOT CASESPECIFIC,
c1 DATE FORMAT 'YY/MM/DD',
PRIMARY KEY (a1));
</SQLText>
</Table>
<Table objVer="1" objId="0:422" kind="set" baseClass="Table" name="t2"
dbName="testdb" fallback="false" beforeJournal="no" afterJournal="no"
checksumLevel="default" mergeBlockRatio="default">
<ColumnList>
<Column name="a2" order="1">

```

```

<DataType><Integer/></DataType>
<Constraint nullable="false"/>
</Column>
<Column name="b2" order="2">
<DataType><Decimal precision="6" scale="4"/></DataType>
<Constraint nullable="false"/>
</Column>
<Column name="c2" order="3">
<DataType><Char length="20" varying="true" uppercase="false"
casespecific="false" charset="unicode"/></DataType>
</Column>
</ColumnList>
<Index>
<PrimaryIndex unique="true">
<ColumnList>
<Column name="a2" order="1"/>
</ColumnList>
</PrimaryIndex>
</Index>
<SQLText>
<![CDATA[
CREATE SET TABLE TESTDB2.t2 ,NO FALLBACK ,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT,
DEFAULT MERGEBLOCKRATIO
(
a2 INTEGER NOT NULL,
b2 DECIMAL(6,4) NOT NULL,
c2 VARCHAR(20) CHARACTER SET UNICODE NOT CASESPECIFIC)
UNIQUE PRIMARY INDEX ( a2 );
</SQLText>
</Table>
<Environment>
  <Server dbRelease="14.10.00.00" dbVersion="14.10.00.06"
hostName="testhost"/>
  <User userId="0" userName="dbc"/>
  <Session charset="utf8" dateTime="2012-01-01T14:00:00-08:00"/>
</Environment>
</TeradataDBObjectSet>

```

Example: SHOW for a SELECT Request with a Function Mapping

In this example, the SHOW for a SELECT request statement includes a function mapping definition and a foreign server definition that is part of the function mapping definition.

Below is the function mapping definition.

```
CREATE FUNCTION MAPPING appl_view_db.sessionize
  FOR sessionize SERVER coprocessor
  USING
    Timecolumn, timeout(100), clicklag(20), emitnull;
```

Following is the SHOW SELECT request.

```
SHOW SELECT * from test_db.sessionize (
  ON test_user.t1 as InputTable PARTITION BY 1
  USING
    TimeColumn('periodcol'),
    clicklag(200)
) as dt;
```

The SHOW request statement returns the definitions of the table, the function mapping, and the foreign server.

```
*** Text of DDL statement returned.
-----
CREATE SET TABLE TEST_USER.t1 ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT,
  DEFAULT MERGEBLOCKRATIO,
  MAP = TD_MAP1
  (
    userid INTEGER,
    periodcol TIMESTAMP(6),
    page VARCHAR(1000) CHARACTER SET LATIN NOT CASESPECIFIC)
PRIMARY INDEX ( userid );

*** Text of DDL statement returned.
-----
CREATE FUNCTION MAPPING appl_view_db.sessionize
  FOR sessionize SERVER coprocessor
```

```
USING  
Timecolumn, timeout(100), clicklag(20), emitnull;
```

```
*** Text of DDL statement returned.
```

```
-----  
CREATE FOREIGN SERVER TD_SERVER_DB.opt_srv1  
EXTERNAL SECURITY DEFINER TRUSTED TESTAUTH USING  
link ('sdlc4673sdlc4675')  
version ('active')  
DO IMPORT WITH TD_SYSFNLIB.QGINITIATORIMPORT ,  
DO EXPORT WITH TD_SYSFNLIB.QGINITIATOREXPORT ;
```

Notation Conventions

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 [<i>delimiter</i> ...]	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

Character Shorthand Notation Used in This Document

This document uses the Unicode naming convention for characters. For example, the lowercase character 'a' is more formally specified as either LATIN CAPITAL LETTER A or U+0041. The U+xxxx notation refers to a particular code point in the Unicode standard, where xxxx stands for the hexadecimal representation of the 16-bit value defined in the standard.

In parts of the document, it is convenient to use a symbol to represent a special character, or a particular class of characters. This is particularly true in discussion of the following Japanese character encodings:

- KanjiEBCDIC
- KanjiEUC
- KanjiShift-JIS

These encodings are further defined in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Character Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

Symbol	Encoding	Meaning
a-z A-Z 0-9	Any	Any single byte Latin letter or digit.
<u>a-z</u> <u>A-Z</u> <u>0-9</u>	Any	Any fullwidth Latin letter or digit.

Symbol	Encoding	Meaning
<	KanjiEBCDIC	Shift Out [SO] (0x0E). Indicates transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEBCDIC	Shift In [SI] (0x0F). Indicates transition from multibyte to single byte KanjiEBCDIC.
T	Any	Any multibyte character. The encoding depends on the current character set. For KanjiEUC, code set 3 characters are always preceded by ss3.
!	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by ss2, forming an individual multibyte character.
△	Any	Represents the graphic pad character.
Δ	Any	Represents a single or multibyte pad character, depending on context.
ss 2	KanjiEUC	Represents the EUC code set 2 introducer (0x8E).
ss 3	KanjiEUC	Represents the EUC code set 3 introducer (0x8F).

For example, string “TEST”, where each letter is intended to be a fullwidth character, is written as **TEST**. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set:

LMN<TEST>QRS

is represented as:

D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2

Pad Characters

The following table lists the pad characters for the various character data types.

Server Character Set	Pad Character Name	Pad Character Value
LATIN	SPACE	0x20
UNICODE	SPACE	U+0020
GRAPHIC	IDEOGRAPHIC SPACE	U+3000
KANJISJIS	ASCII SPACE	0x20
KANJI1	ASCII SPACE	0x20

Object Data Types

Data Types Syntax

data_type

```
{ INTEGER | SMALLINT | BIGINT | BYTEINT | DATE |

{ TIME | TIMESTAMP } [ (fractional_seconds_precision) ] [WITH TIME ZONE] |

INTERVAL YEAR [(precision)] [TO MONTH] |

INTERVAL MONTH [(precision)] |

INTERVAL DAY [(precision)] [TO { HOUR | MINUTE | SECOND
[(fractional_seconds_precision)] }] |

INTERVAL HOUR [(precision)] [TO { MINUTE | SECOND
[(fractional_seconds_precision)] }] |

INTERVAL MINUTE [(precision)] [TO SECOND [(fractional_seconds_precision)]] |

INTERVAL SECOND [(precision) [, fractional_seconds_precision]] |

PERIOD (DATE) |

PERIOD ( { TIME | TIMESTAMP } [(precision)] [WITH TIME ZONE] ) |

REAL |

DOUBLE PRECISION |

FLOAT [ (integer) ] |

NUMBER [ ( { integer | *} [, integer]... ) ] |

{ DECIMAL | NUMERIC } [ ( integer [, integer]... ) ] |

{ CHAR | BYTE | GRAPHIC } [ (integer) ] |
```

```

{ VARCHAR | CHAR VARYING | VARBYTE | VARGRAPHIC } [ ( integer ) ] |

LONG VARCHAR |

LONG VARGRAPHIC |

{ BINARY LARGE OBJECT | BLOB | CHARACTER LARGE OBJECT | CLOB } ( integer [ G | K
| M ] ) |

[SYSUDTLIB.] { XML | XMLTYPE } [ ( integer [ G | K | M ] ) ] [ INLINE LENGTH
integer ] |

[SYSUDTLIB.] JSON [ ( integer [ K | M ] ) ] [ INLINE LENGTH integer ]
[ CHARACTER SET { UNICODE | LATIN } | STORAGE FORMAT { BSON | UBJSON } ] |

[SYSUDTLIB.] ST_GEOMETRY [ ( integer [ K | M ] ) ] [ INLINE LENGTH integer ] |

[SYSUDTLIB.] DATASET [ ( integer [ K | M ] ) ] [ INLINE LENGTH integer ]
STORAGE FORMAT { Avro | CSV [ CHARACTER SET { UNICODE | LATIN } ] }
[ WITH SCHEMA [database.] schema_name ] |

[SYSUDTLIB.] { UDT_name | MBR | ARRAY_name | VARRAY_name }
}

```

Usage Notes

The JSON data type STORAGE FORMAT option can only be specified as:

- An attribute of a table column, since it only specifies a storage format.
- Part of a JSON data type declaration in a CAST statement.
- An optional parameter to the JSON constructor.

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